VAX/VMS
Guide to Writing a Device Driver

Order No. AA-H499C-TE

May 1982

This document explains how to write device drivers for devices that are not supported by VAX/VMS, and how to load these drivers into the VAX/VMS operating system.

REVISION/UPDATE INFORMATION: This document supersedes the VAX/VMS Guide to Writing a Device Driver (Order No. AA-H499B-TE), including Update Notice No. 1 (Order No. AD-H499B-T1).

SOFTWARE VERSION: VAX/VMS Version 3.0

digital equipment corporation · maynard, massachusetts

First Printing, February 1979
Revised, March 1980
Updated, January 1981
Revised, May 1982

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Printed in U.S.A.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	RSX
DEC/CMS	EduSystem	UNIBUS
DECnet	IAS	VAX
DECsystem-10	MASSBUS	VMS
DECSYSTEM-20	PDP	VT
DECUS	PDT	digital
DECwriter	RSTS	

ZK2140

HOW TO ORDER ADDITIONAL DOCUMENTATION

In Continental USA and Puerto Rico call 800-258-1710

In New Hampshire, Alaska, and Hawaii call 603-884-6660

In Canada call 613-234-7726 (Ottawa-Hull)
800-267-6146 (all other Canadian)

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire 03061

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575)

DIRECT MAIL ORDERS (CANADA)

Digital Equipment of Canada Ltd.
940 Belfast Road
Ottawa, Ontario K1G 4C2
Attn: A&SG Business Manager

DIRECT MAIL ORDERS (INTERNATIONAL)

Digital Equipment Corporation
A&SG Business Manager
c/o Digital's local subsidiary or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

CONTENTS

Page

Page

Page

Page

FIGURES

TABLES

PREFACE

The VAX/VMS Guide to Writing a Device Driver provides the information needed to write a device driver that runs under VAX/VMS Version 3.0 and to load that driver into the operating system. VAX/VMS makes no guarantee that drivers written for VAX/VMS Versions 1.0, through 1.6 and 2.0 through 2.5 will execute without modification on subsequent versions of the operating system. While the intent is to maintain the existing interface, some unavoidable changes may occur as new features are added. The use of internal executive interfaces other than those described in this manual is discouraged.

INTENDED AUDIENCE

This manual is intended for system programmers who are already familiar with the VAX-11 processor and the VAX/VMS operating system. The manual focuses on writing drivers for devices attached to the UNIBUS; however, Appendix F provides the additional information needed to write a driver for a device attached to the MASSBUS.

STRUCTURE OF THIS DOCUMENT

This manual is organized into two parts. The first part consists of the following chapters, which introduce VAX/VMS device drivers and those aspects of the VAX-11 processor and the VAX/VMS system that are essential to drivers:

- Chapter 1 introduces the main concepts associated with drivers on VAX/VMS.
- Chapter 2 describes an example of a line printer driver handling a data transfer.
- Chapter 3 discusses synchronization mechanisms: interrupt priority levels, fork processes and fork queues, and resource wait queues.
- Chapter 4 discusses UNIBUS considerations for direct memory access (DMA) transfers.
- Chapter 5 provides an overview of I/O processing and discusses the interaction between device drivers and VAX/VMS.

The second part of this document is a series of "how to" chapters that provide a sample approach to coding a device driver:

- Chapter 6 contains a template for writing a device driver.
- Chapter 7 details the macros that drivers invoke to create necessary tables.

- Chapter 8 describes the writing of function decision routines.
- Chapter 9 describes the writing of a start I/O routine.
- Chapter 10 describes the UNIBUS considerations for a start I/O routine.
- Chapter 11 describes the writing of an interrupt service routine.
- Chapter 12 describes the writing of I/O completion and device timeout routines.
- Chapter 13 describes the writing of unit and controller initialization routines, I/O cancellation routines, and error-logging routines.
- Chapter 14 describes the loading of a driver into the system.
- Chapter 15 describes the debugging tool XDELTA that you can use to debug a device driver.
- Appendix A describes the I/O data base in detail. This is an important appendix for the programmer of a device driver.
- Appendix B describes the VAX/VMS macros that drivers can invoke.
- Appendix C describes the VAX/VMS routines that device drivers can call.
- Appendix D contains a sample driver for an analog-to-digital converter.
- Appendix E contains a sample driver for two connected DR11s.
- Appendix F contains information needed to write a device driver for a device attached to the MASSBUS.
- Appendix G lists the starting physical addresses for the UNIBUS memory address space associated with the VAX-11 processors.
- The glossary at the end of this manual defines I/O-related and driver-related terms.

ASSOCIATED DOCUMENTS

This document has the following prerequisites:

- VAX Hardware Handbook
- VAX/VMS Summary Description and Glossary
- I/O-related portions of the VAX/VMS System Services Reference Manual
- The appendix on naming conventions in the VAX-11 Guide to Creating Modular Library Procedures
- VAX/VMS I/O User's Guide

The following documents are associated with this manual:

- VAX/VMS System Dump Analyzer Reference Manual
- VAX/VMS System Management and Operations Guide
- VAX/VMS Internals and Data Structures

CONVENTIONS USED IN THIS DOCUMENT

This manual describes code transfer operations in three ways:

1. The phrase "issues a system service call" implies the use of a CALL instruction.
2. The phrase "calls a routine" implies the use of a JSB or BSB instruction.
3. The phrase "transfers control to" implies the use of a BRB, BRW, or JMP instruction.

SUMMARY OF TECHNICAL CHANGES

This manual applies to Version 3.0 of VAX/VMS. The following list summarizes the major technical changes from the Version 2.2 manual:

1. DPTAB macro arguments -- The DPTAB macro has two optional arguments that allow drivers to control the SYSGEN utility's automatic configuration of the devices they operate:

   - The DEFUNITS argument -- Specifies to AUTOCONFIGURE a default number of units to be configured for a given controller.

   - The DELIVER argument -- Specifies the address of a driver-specific unit delivery action routine. See Chapter 7 and Chapter 14 for a discussion of these DPTAB arguments.

2. DDTAB macro argument -- The DDTAB macro has an optional argument, MNTVER, that specifies the address of a routine called at the start and end of a mount verification operation. See Chapter 7 for details.

3. SHOW command qualifiers -- The SYSGEN SHOW command has additional qualifiers that display values within the system configuration:

   - SHOW/ADAPTER -- Displays adapter nexus values

   - SHOW/CONFIGURATION -- Displays device CSR addresses, vector addresses, and associated adapter nexus values

   The SHOW command qualifiers are briefly described in Chapter 14.

4. Configuration Control Block (ACF) -- The SYSGEN autoconfiguration facility uses this data structure to describe the device it is currently adding to the configuration. Appendix A contains a description of the fields within the ACF.

5. The VAX-11/730 -- The following chapters describe features specific to the VAX-11/730:

   - Chapter 4 describes the VAX-11/730 UNIBUS adapter.

   - Chapter 14 gives VAX-11/730 adapter nexus values.

   - Chapter 15 describes how to bootstrap a VAX-11/730 with XDELTA.

   - Appendix G gives starting physical addresses for VAX-11/730 UNIBUS memory address space.

PART I
Introduction

CHAPTER 1

INTRODUCTION TO DEVICE DRIVERS

Under the VAX/VMS operating system, a device driver is a set of routines and tables that the system uses to process an I/O request for a particular device type. In order to understand how drivers are used by the VAX/VMS system, you must become familiar with the following basic concepts:

- Machine dependence and independence
- Asynchronous nature of a device driver
- Fork processes
- Process and interrupt context
- Device dependence and device independence
- I/O data base
- Synchronization mechanisms

The beginning sections of this chapter describe the concepts listed above. The later sections describe the more concrete aspects of drivers, such as the functions they perform.

1.1 MACHINE DEPENDENCE AND MACHINE INDEPENDENCE

The VAX/VMS operating system can run on any of three VAX-11 processors: the VAX-11/780, VAX-11/750, or VAX-11/730. Although these processors conform to the VAX-11 architecture, there are some differences in design among the three machines. To achieve machine-independence, follow the conventions outlined in this manual when you write a device driver. The driver will then operate on any processor without modification.

To aid in driver debugging, sections of this manual discuss certain internal differences among the VAX-11/780, VAX-11/750, and the VAX-11/730. This section defines several terms that describe the hardware configuration of each of these processors in a machine-independent manner:

- Backplane interconnect -- An internal processor bus that UNIBUS and MASSBUS adapters use to communicate with main memory and the central processor.
- UNIBUS adapter -- An interface device between the backplane interconnect and a UNIBUS.

UNIBUS adapters may be direct vector or nondirect vector devices. On a direct vector UNIBUS adapter, UNIBUS device interrupts cause a direct processor interrupt that jumps to vectors in page two (or three) of the system control block (SCB). On nondirect vector UNIBUS adapters, UNIBUS device interrupts cause a UNIBUS adapter interrupt and are dispatched by the UNIBUS adapter interrupt service routine. Chapters 3 and 4 discuss these adapters in more detail.

- MASSBUS adapter -- An interface device between the backplane interconnect and a MASSBUS.

- Interrupt dispatcher -- A combination of hardware and software that routes UNIBUS and MASSBUS device interrupts to the appropriate device driver interrupt service routine. The interrupt dispatcher's routing mechanism works differently depending upon whether the VAX-11 processor in use accepts direct vector or nondirect vector UNIBUS interrupts and whether the adapter in use is a MASSBUS or UNIBUS adapter.

- Physical address -- The physical memory that UNIBUS and MASSBUS adapters address through the backplane interconnect. The different VAX-11 processors have different amounts of physical address space. Physical addresses of device registers also vary with processor type.

1.2 COMPONENTS OF A DEVICE DRIVER

Normally, a device driver module consists of the following routines and tables:

- An I/O preprocessing routine or routines that validate device-specific parameters of an I/O request, format data, allocate system buffers, and lock pages in memory

- A start I/O routine that activates the device

- An interrupt service routine that responds to interrupts from the device unit

- An error recovery routine that retries I/O operations and performs other error handling

- An error-logging routine that writes the contents of device registers and other data into an error buffer for the system

- A cancel I/O routine that prevents further processing of an I/O request

- An initialization routine that readies a device or controller for operation when the system is bootstrapped or during recovery from a power failure

- A driver prologue table that describes the driver and the device type to the VAX/VMS procedure that loads drivers into the system

- A driver dispatch table that lists the entry point addresses of standard driver routines and records the size of diagnostic and error-logging buffers for the device type

- A function decision table that lists all valid function codes for the device and lists the addresses of I/O preprocessing routines associated with each valid function

With a few exceptions, which are noted in Chapter 7, the order of the various routines and tables within the driver module is not important.

1.3 ASYNCHRONOUS NATURE OF A DEVICE DRIVER

Using the driver tables and other information maintained by the driver and the operating system, the system determines which routines to activate and when they should be activated, as illustrated in Figure 1-1. For example, when a user process issues a Queue I/O Request system service, the system service calls various driver routines to perform preprocessing of the I/O request. Likewise, if a user process issues a Cancel I/O on Channel system service, the system service activates the driver's cancel I/O routine.

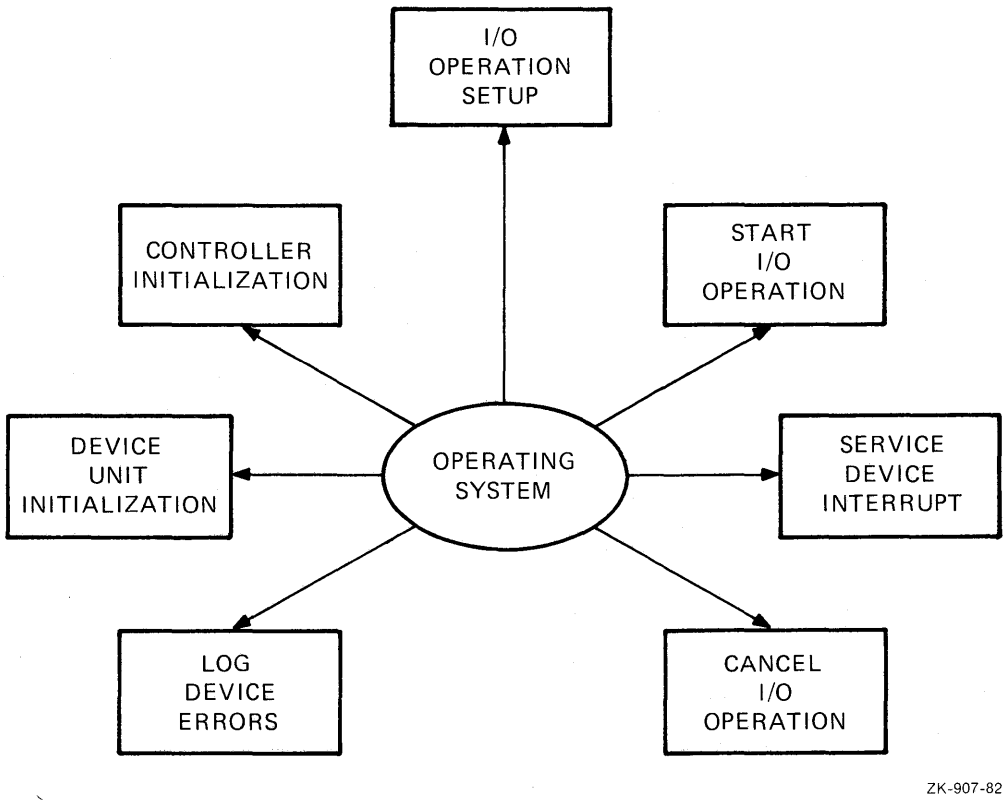


Figure 1-1: VAX/VMS Calls to Driver Routines

A device driver does not run from start to end. The system calls driver routines and suspends and resumes them; the central processor interrupts and reactivates driver routines. Because little sequential processing of driver code occurs, VAX/VMS must assume the responsibility for synchronizing the execution of the various driver routines and synchronizing the execution of all drivers in the system. The VAX/VMS operating system synchronizes driver execution using fork processes, interrupt priority levels, fork queues, and resource wait queues, described in the following sections.

1.4 FORK PROCESSES

A fork process is a process that is created dynamically and has minimal context. Fork processes execute entirely within the system address space. The VAX/VMS operating system creates and schedules a fork process by constructing a specialized control block called a fork block, inserting the fork block in a fork queue, and requesting a software interrupt. Fork queues and fork process dispatching are described further in Section 1.7.3.

A driver fork process has the following context:

- Three general registers
- Program counter (PC)
- A unit control block in the I/O data base that describes the target device of the I/O request

The unit control block also contains the driver's fork block. Section 1.8 describes the unit control block and other control blocks in the I/O data base.

Like other processes, fork processes can be suspended and interrupted. VAX/VMS places a driver fork process in a wait state when the process requests an unavailable resource, for example, a controller data channel. The processor interrupts a fork process when the processor receives a request for an interrupt at a higher priority level.

Driver fork processes execute at raised interrupt priority levels to minimize the number of interruptions. Fork processes can raise the priority level to 31 to block all other interrupts, if necessary.

The system automatically saves registers for interrupted fork processes and restores these registers when the process is reactivated. The operating system does not swap fork processes because the fork block and all data about the fork process reside in nonpaged system memory.

1.5 PROCESS CONTEXT AND INTERRUPT CONTEXT

Because a device driver consists of a number of routines that are activated by VAX/VMS, the operating system for the most part determines the context in which the routines execute. As an example, consider the following write request that occurs without error:

- A user process executing in user mode issues a write Queue I/O Request system service.
- The Queue I/O Request system service gains control in user process context but in kernel mode.
- The system service uses the driver's function decision table to call the appropriate preprocessing routines. These routines, called FDT routines, execute in full process context in kernel mode.
- When preprocessing is complete, a VAX/VMS routine creates a fork process to execute the driver's start I/O routine in kernel mode.

- The start I/O routine activates the device unit and suspends itself. At this point, VAX/VMS suspends the fork process executing the start I/O routine and saves sufficient context to reactivate the start I/O routine at the point of suspension.

- When the device completes the data transfer, it issues an interrupt. The interrupt causes the system to activate the driver's interrupt service routine.

- The interrupt service routine executes to handle the device interrupt. It then causes the start I/O routine to resume in interrupt context.

- The start I/O routine regains control in interrupt context but almost immediately issues a request to the operating system to transform its context to that of a fork process. This action dismisses the interrupt.

- When reactivated in fork process context, the start I/O routine performs device-specific I/O completion and passes control to the system for additional I/O postprocessing.

- VAX/VMS I/O postprocessing performs processing at a software interrupt priority level and then issues a kernel mode asynchronous system trap (AST) for the user process requesting I/O.

- When the kernel mode AST is delivered, the AST routine executes in full process context at kernel mode to deliver data and status to the process. If the original request specified a user mode AST, the kernel mode AST queues it.

- When the user process gains control, the user's AST routine executes in full process context in user mode.

It is essential, however, that the various driver routines not attempt to exceed the limitations of the context in which they execute. The majority of driver routines execute in fork process context.

1.6 DEVICE DEPENDENCE AND DEVICE INDEPENDENCE

The VAX/VMS approach to I/O is that the operating system should perform as much of the processing of an I/O request as possible and that drivers should restrict themselves to the device-specific aspects of I/O processing. To accomplish this, the VAX/VMS operating system provides drivers with the following services:

- The Queue I/O Request system service preprocesses an I/O request by performing those functions and checks that are common to all devices; for example, it validates the arguments in the I/O request that are not device specific. This type of preprocessing is called device-independent preprocessing.

- The VAX/VMS operating system includes a number of routines that drivers can call to perform I/O preprocessing, allocate and deallocate resources, and synchronize driver execution.

- VAX/VMS I/O postprocessing performs the device-independent I/O postprocessing for all I/O requests.

Thus, drivers can leave the device-independent I/O processing to the operating system and concentrate on the device-dependent aspects of a device unit; that is, those aspects that vary from device type to device type. In addition, drivers can call the VAX/VMS system to perform many functions that are device specific but common to several devices.

1.7 THE I/O DATA BASE

Because a driver and the operating system cooperate to process an I/O request, they must have a common I/O data base. Under VAX/VMS, the I/O data base consists of three main parts:

- Driver tables that allow the system to load drivers, validate device functions, and call drivers at their entry points
- Control blocks that describe every bus adapter, every device type, every device unit, every controller, and every logical path (channel) from a process to a device
- I/O request packets that define individual requests for I/O activity

The three driver tables are defined in every driver. Section 1.2 lists these tables. Appendix A describes each of the control blocks and the I/O request packet in detail.

Figure 1-2 illustrates some of the relationships among VAX/VMS I/O routines, the I/O data base, and a device driver.

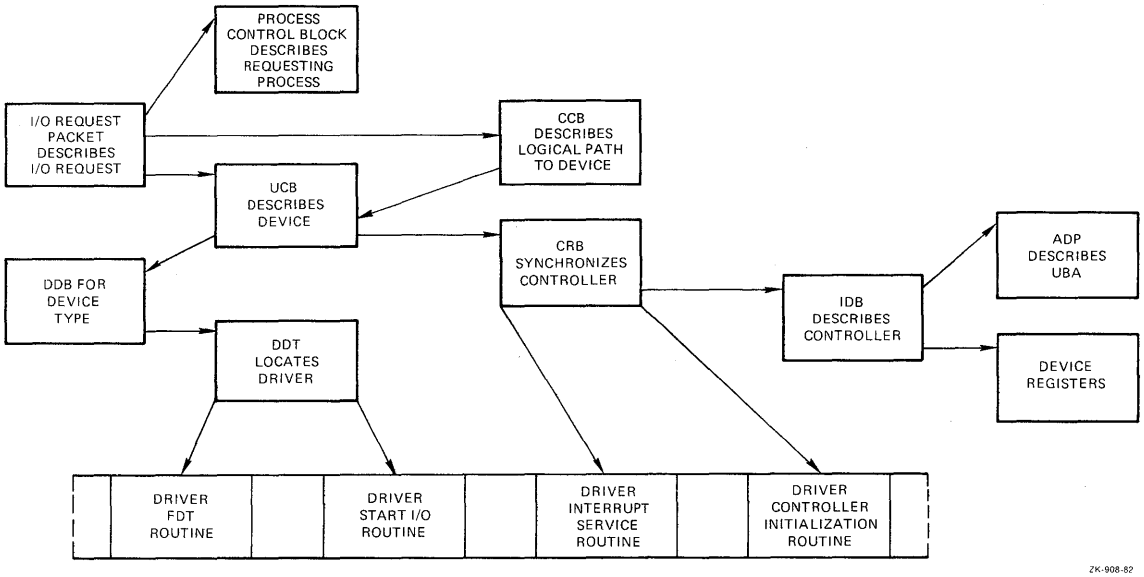


Figure 1-2: The I/O Data Base

1.7.1 Control Blocks In The I/O Data Base

Control blocks in the I/O data base permit access to and describe peripheral hardware. The VAX/VMS operating system creates these

control blocks either at system start-up or at the time a user-written driver is loaded into the system. Drivers refer to some or all of the following control blocks:

- Device data block (DDB)
- Unit control block (UCB)
- Channel request block (CRB)
- Interrupt dispatch block (IDB)
- Adapter control block (ADP)
- Channel control block (CCB)

1.7.1.1 Device Data Block - A device data block 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, and the driver name and location for those devices. In addition, the device data block contains a pointer to the first unit control block for the device units attached to the controller.

1.7.1.2 Unit Control Block - The system defines a unit control block for each device attached to the system. A unit control block defines the characteristics and current state of an individual device unit. In addition, it contains the fork block used by the unit's device driver and the listhead for the queue of pending I/O request packets for the unit. Because drivers execute as fork processes that are created for each I/O operation on a unit, the unit control blocks are the focal point of the I/O data base. When a driver is suspended or interrupted, the UCB fork block holds the driver's context.

1.7.1.3 Channel Request Block - The operating system creates a channel request block for each controller. A channel request block defines the current state of the controller and lists the devices waiting for the controller's data channel. In addition, it contains the code that dispatches a device interrupt to the interrupt service routine for that unit's driver.

1.7.1.4 Interrupt Dispatch Block - The system creates an interrupt dispatch block for each controller. An interrupt dispatch block lists the device units associated with a controller and points to the unit control block of the device unit that the controller is currently servicing. In addition, an interrupt dispatch block points to device registers and the controller's UNIBUS adapter.

1.7.1.5 Adapter Control Block - An adapter control block defines the characteristics and current state of a UNIBUS or MASSBUS adapter. An adapter control block for the UNIBUS adapter contains the queues and allocation bit maps necessary to allocate adapter resources. VAX/VMS provides routines that drivers can call to interface with their UNIBUS adapter.

1.7.1.6 Channel Control Block - A channel is a logical path between a process and the unit control block of a specific device unit. The channel control block describes this path. Each process owns a number of channel control blocks. When a process issues the Assign I/O Channel system service, the system writes a description of the assigned device to the channel control block. Unlike the data structures mentioned earlier, a channel control block is not located in nonpaged system space, but in the process's control region (P1 space).

1.7.2 I/O Request Packets

The third part of the I/O data base is a list of I/O request packets (IRPs). When a process requests I/O activity, the operating system constructs a packet of data, called an I/O request packet, that describes the I/O request in standard form.

The I/O request packet contains fields into which the system and driver I/O preprocessing routines can write information, such as device-dependent parameters specified in the call to the Queue I/O Request system service. Later, the system sends the I/O request packet to the device driver start I/O routine. The driver start I/O routine uses the packet as its source of detailed instructions about the operation to be performed. The packet includes buffer addresses, a pointer to the target device, I/O function code, and further pointers to the I/O data base.

1.8 SYNCHRONIZATION

The VAX/VMS operating system uses hardware and software interrupt priority levels (IPLs) with their associated interrupts, fork queues, and resource wait queues to synchronize the execution of all drivers within the system and to synchronize execution of various routines within a driver.

1.8.1 Interrupt Priority Levels

The VAX-11 processor defines 32 interrupt priority levels (0 through 31). The higher numbered IPLs are reserved for hardware interrupts, for example, device interrupts. The operating system uses the lower numbered IPLs. A higher IPL always takes precedence over a lower IPL. The VAX Hardware Handbook describes the VAX-11 processors' use of IPLs. The following IPLs are of particular interest to drivers:

- Hardware device IPLs (20 through 23); driver interrupt service routines execute at these IPLs.
- Driver fork processing IPLs (8 through 11); driver fork processes execute at these IPLs.
- I/O completion IPL (IPL 4); VAX/VMS gains control to begin its device-independent I/O postprocessing at this IPL.
- AST delivery IPL (IPL 2); VAX/VMS uses this IPL to coordinate the delivery of an AST to a user process. The Queue I/O Request system service also executes at this IPL.

1.8.2 Device Interrupts

When the processor grants a device interrupt, the processor and the VAX/VMS interrupt dispatcher save the current process context. The processor pushes the PC and PSL at the time of the interrupt onto the interrupt stack. In addition, the interrupt dispatcher saves R0 through R5 on the stack.

The interrupt service routine activated as a result of the interrupt follows conventions to preserve all other context of the interrupted process, as follows:

- Uses only R0 through R5
- Cleans up the stack after use

When the interrupt has been serviced, the driver interrupt service routine restores R0 through R5 from the stack. The processor restores the previous PC and PSL of the interrupted code. The interrupted process then resumes execution without any awareness of the interruption.

1.8.3 Fork Queues

When an interrupt service routine completes the handling of a device interrupt, it transfers control to the driver to complete device-dependent processing of the I/O request. When the driver regains control, it is executing at device IPL. Almost immediately, the driver should lower IPL to driver fork IPL so that it does not block other device interrupts. A driver lowers IPL by invoking a VAX/VMS macro that creates a fork process to execute at driver fork IPL.

Each driver fork IPL has an associated fork queue. A VAX/VMS macro queues the driver's fork block in the fork queue that corresponds to the driver's fork IPL and issues a software interrupt request for that IPL. When the software interrupt is granted, the VAX/VMS fork dispatcher dequeues fork blocks from the driver fork queues and reactivates the driver at the point following the macro invocation.

1.8.4 Resource Wait Queues

Drivers compete for the following shared resources:

- Central processor
- UNIBUS adapter mapping registers, if the device is a DMA device
- UNIBUS adapter buffered data paths, if the device is a DMA device
- The controller data channel if the device is attached to a multiunit controller

When a driver fork process needs an unavailable resource, VAX/VMS resource management routines perform the following steps:

- Save fork process context in the device's UCB fork block
- Insert the address of the UCB fork block in a resource wait queue
- Suspend the driver fork process

When another driver fork process frees the necessary resource, the VAX/VMS resource management routines take the following steps to reactivate the next driver fork process:

- Remove the next UCB fork block from the resource wait queue
- Restore fork process context into the registers
- Reactivate the suspended driver fork process

The VAX/VMS resource management routines allow the driver fork process to be unaware of its suspension and reactivation.

1.9 FUNCTIONS OF A DEVICE DRIVER

A VAX/VMS device driver controls I/O operations on a peripheral device by performing the following functions:

- Defines the peripheral device for the rest of VAX/VMS
- Defines the driver for the system procedure that loads the driver into system virtual address space and that creates the driver's associated data structures
- Readies the device and/or its controller for operation at system start-up and during recovery from a power failure
- Performs device-dependent I/O preprocessing
- Translates programmed requests for I/O operations into device-specific commands
- Activates the device
- Responds to hardware interrupts generated by the device
- Responds to device timeout conditions
- Responds to requests to cancel I/O on the device
- Reports device errors to an error-logging program
- Returns status from the device to the process that requested the I/O operation

The driver prologue table, described in Section 7.1, performs the first two functions listed above. Driver routines perform the remaining functions.

1.9.1 Initialization Routines

Most device controllers and device units require initialization when the VAX/VMS driver loading procedure loads the driver into memory and when the VAX/VMS system recovers from a power failure. The amount and

type of initialization varies from device type to device type. Section 13.1 provides additional information about device driver initialization routines.

1.9.2 FDT Routines

Every driver contains a function decision table (FDT) that indicates the I/O preprocessing routines that are to be executed for various functions on the device. When a user process issues a Queue I/O Request system service, the system service uses the I/O function code specified in the request to select one or more FDT routines for execution. FDT routines perform such functions as allocating buffers, locking pages in memory, and validating device-dependent parameters (P1 through P6) of the I/O request.

The driver contains FDT routines that are device-dependent. VAX/VMS provides additional FDT routines that perform processing common to many I/O functions, as described in Section 8.7. It is advisable for drivers to use FDT routines supplied by the operating system whenever possible.

Because FDT routines are called by the Queue I/O Request system service, they execute in full user process context. As a result, FDT routines have access to user-specified buffers located in the process address space; these buffers are not available to driver routines executing in fork context.

1.9.3 Start I/O Routine

The start I/O routine executes in a driver fork process to perform the following device-dependent functions:

- Translate the I/O function code into a device-specific command
- Transfer the details of the request from the I/O request packet to the device's unit control block
- Obtain access to the controller if it is a multiunit controller
- Obtain the necessary UNIBUS resources if the transfer is direct memory access (DMA)
- Modify the device registers to activate the device
- Perform device-dependent I/O postprocessing after the transfer occurs

The start I/O routine may be forced to wait for the controller or UNIBUS resources to become available. In either case, VAX/VMS suspends the routine and reactivates it when the resources are free. Section 1.8.4 describes the context that VAX/VMS saves for the suspended routine.

After activating the device, the start I/O routine invokes the VAX/VMS wait for interrupt macro. The wait for interrupt macro suspends the driver. The driver remains suspended until the driver's interrupt service routine handles the interrupt and returns control to the driver. At that point, the driver performs device-dependent I/O postprocessing and then transfers control to VAX/VMS for device-independent I/O postprocessing.

1.9.4 Interrupt Service Routine

When a device interrupt occurs, VAX/VMS transfers control to the device driver's interrupt service routine in interrupt context. The interrupt service routine determines whether the interrupt was expected or not and takes the appropriate action. Then the interrupt service routine reactivates the driver for I/O postprocessing.

1.9.5 Device Timeout Handler

As the result of an error condition or a device's being offline, it is possible for a device to fail to complete a transfer in a reasonable period of time. This condition is called device timeout. When a start I/O routine invokes the wait for interrupt macro, it specifies the time interval in which the device can complete a transfer without timing out and the name of a timeout handler that the system is to invoke in the case of a timeout. This information is recorded in the device's unit control block.

Once every second, the VAX/VMS system timer checks all devices in the system for device timeout. When it locates a device that has timed out, it calls the timeout handler.

1.9.6 Cancel I/O Routine

VAX/VMS provides the Cancel I/O on Channel system service that user processes can call to cancel I/O requests. The Cancel I/O on Channel system service, in turn, calls the driver's cancel I/O routine. VAX/VMS also calls the driver's cancel I/O routine when the device's reference count goes to zero; that is, when all users that assigned channels to the device have deassigned them.

1.9.7 Error-logging Routine

The driver's error-logging routine fills an error log buffer with information about the error, for example, the register contentsat the time of the error. VAX/VMS provides a routine that drivers can call to allocate an error log buffer and transfer control to the register dump routine.

1.10 AN EXAMPLE OF A UNIBUS I/O REQUEST

Figure 1-3 illustrates how the VAX/VMS operating system and the device driver process a user process request for a read I/O operation on a DMA UNIBUS device.

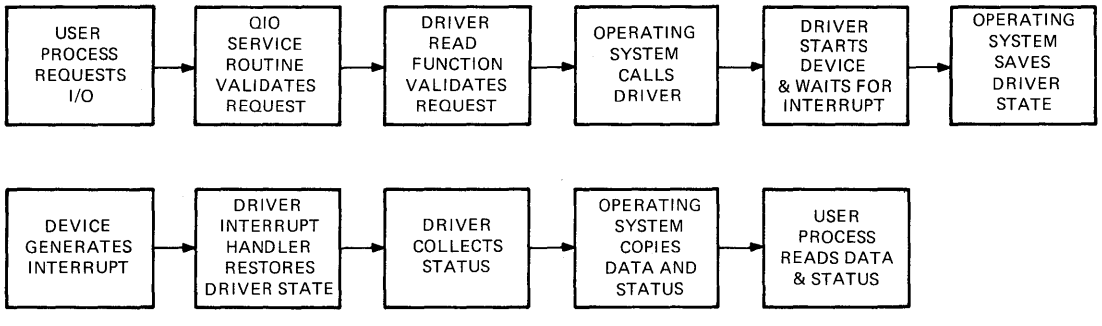


Figure 1-3: Processing a Sample I/O Operation

The processing of the sample I/O request illustrated in Figure 1-3 occurs in the following steps:

- **A process requests I/O operation.** A user process requests data from the device by issuing either of the following:

  - A VAX-11 RMS get record function call (which results in a Queue I/O request)

  - A Queue I/O Request system service

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

- **The operating system performs I/O preprocessing.** The Queue I/O Request system service validates the request and locates I/O data base control blocks that describe the device and its driver. The system service also allocates and initializes an I/O packet to contain a description of the I/O request. The system service then calls a read function routine in the driver.

- **The driver performs I/O preprocessing.** The driver function decision table 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.

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

- **The driver readies the UNIBUS adapter.** For DMA transfers, the driver fork process calls VAX/VMS routines that control the UNIBUS adapter hardware to map UNIBUS addresses into physical addresses for the transfer.

- **The driver activates the device.** The fork process activates the device by setting bits in device registers.

- **The driver waits for an interrupt.** A VAX/VMS routine saves the context of the driver fork process and relinquishes the processor until an interrupt occurs.

- **The device requests an interrupt.** When the data transfer is complete, the device requests a hardware interrupt that causes the system to dispatch to the driver's interrupt service routine.

- **The driver services the interrupt.** The driver's interrupt service routine handles the interrupt and reactivates the driver, which reads device registers to obtain status information about the transfer.

- **The operating system inserts the driver in a fork queue.** The driver requests that the process be reactivated at a lower software interrupt priority level.

- **The fork dispatcher reactivates the driver fork process.** When processor priority permits, the VAX/VMS fork dispatcher reactivates the driver as a fork process.

- **The driver completes the I/O operation.** The driver fork process completes device-dependent I/O processing of the I/O request and returns the I/O status to VAX/VMS.

- **VAX/VMS completes the I/O operation.** The VAX/VMS I/O postprocessing routines copy the I/O status into process address space and/or general registers and return control to the user process.

Of the thirteen steps listed above, only four describe driver I/O preprocessing and driver fork processing. The VAX/VMS I/O support routines perform all I/O processing common to many or all I/O requests. Even in device driver routines, driver writing is simplified by the use of VAX/VMS routines that handle device-independent functions.

The thirteen-step example condenses and simplifies the processing of an I/O operation by ignoring such issues as the following:

- Association of a device with a process; that is, device assignment

- Simultaneous I/O requests for one device

- Hardware interrupt priority levels

- Driver competition for shared system and UNIBUS adapter resources

- Driver competition for I/O activity through a multiunit controller

- Driver recovery from device errors or power failure

Later chapters discuss each of these issues in relation to device drivers.

1.11 THE UNIBUS

On a VAX-11 system, the backplane interconnect connects the central processor to memory. The backplane interconnect also connects the UNIBUS adapter and MASSBUS adapter to memory and to the central processor. Peripheral devices attach to either the UNIBUS, for UNIBUS devices, or the MASSBUS, for MASSBUS devices, as illustrated in Figure 1-4.

The VAX Hardware Handbook describes the hardware components diagrammed in Figure 4-1.

VAX/VMS provides device drivers for a number of standard devices supported by DIGITAL. These devices are connected to either the MASSBUS or the UNIBUS.

Nonstandard devices, that is, customer-supplied devices, normally are connected to the UNIBUS, but can also be attached to the MASSBUS or to the DR32 device interconnect. DIGITAL supplies a device driver and an application library for the DR32 device; see the chapter on the DR32 Interface Driver in the VAX/VMS I/O User's Guide for further information.

To activate a direct memory access (DMA) transfer on the UNIBUS, a driver must first obtain mapping registers, and, optionally, a buffered data path. The driver calls VAX/VMS routines that interface with the UNIBUS adapter to allocate these resources on behalf of the driver.

The direct data path maps each UNIBUS transfer to a backplane interconnect transfer. For each UNIBUS transfer, there is one backplane interconnect transfer. Each backplane interconnect operation transfers a single word or byte of data depending on the device. A buffered data path, on the other hand, allows multiple UNIBUS transfers to be assembled and transferred in one backplane interconnect operation.

Drivers performing other than DMA transfers are generally not concerned with UNIBUS adapter operation.

Instead of creating a complete device driver for a device that does not perform DMA transfers, you can connect the process to the device interrupt vector to program the device from a user process. For a description of how and when to connect to an interrupt vector, consult the VAX/VMS Real-Time User's Guide.

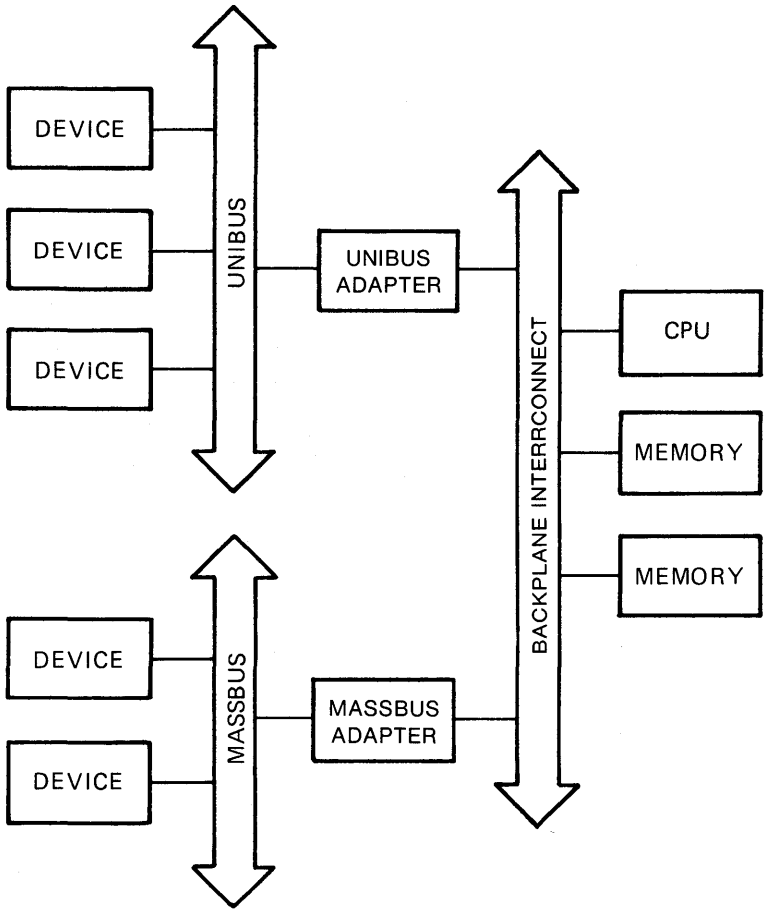


Figure 1-4: VAX-11 Hardware Configuration

1.12 PROGRAMMED I/O AND DIRECT MEMORY ACCESS I/O

Devices transfer data using one of the following methods:

- Programmed I/O
- Direct memory access (DMA) transfers

Devices that perform programmed I/O transfer data as single words or bytes using device registers. After each transfer completes, the device notifies the central processor.

Devices that perform DMA transfers do not require the central processor so frequently. 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. Normally, the driver of a DMA device allocates a UNIBUS buffered data path and UNIBUS map registers for I/O transfers.

1.13 BUFFERED I/O AND DIRECT I/O

Drivers can perform I/O transfers using either of the following methods:

- Buffered I/O
- Direct I/O

Buffered I/O allows data to be buffered in system address space. When the transfer is complete, the data is transferred to the user process's buffer. The driver can refer to the buffer in system space using system virtual addresses. Often, a driver uses buffered I/O for devices that perform programmed I/O, for example, line printers and card readers.

Direct I/O allows data to be placed directly in the user process's buffer. The driver must lock the pages containing the buffer in physical memory and refer to them using page frame numbers (PFNs). Normally, a driver uses direct I/O and a buffered data path for devices that perform DMA transfers.

The trade-off between buffered I/O and direct I/O is the time required to move the data into the user's buffer versus the time required to lock the buffer pages in memory. Chapter 8 provides additional information.

1.14 LOADABLE DRIVERS

The VAX/VMS operating system provides a procedure that allows a suitably privileged user to load drivers into a running VAX/VMS system. The System Generation Utility (SYSGEN) described in full in the VAX-11 Utilities Reference Manual, supports commands that invoke the driver loading procedure:

- LOAD -- to load a driver into the system
- CONNECT -- to create the I/O data base for additional devices of the same type
- RELOAD -- to load a previously loaded driver

The driver loading procedure uses information provided in the LOAD command and information contained in driver tables to load the driver into virtual memory and create the associated data base. The driver prologue table, which must be the first generated code in the driver module, contains the information that the loading procedure needs. Specifically, the driver prologue table contains the following:

- Address of the end of the driver; the loading procedure uses this to determine the size of the driver
- Driver loader flags that indicate whether the device needs a system page table entry and whether the driver can be reloaded
- Size of the unit control block
- Address of a routine to call if the driver is reloaded
- Name of the device driver module

The driver prologue table is followed by two lists of fields that require initialization:

- I/O data base fields to be initialized the first time the driver is loaded
- Fields to be initialized every time the driver is reloaded, that is, without an intervening bootstrap of the system

With the information provided in the driver prologue table and the two lists of fields, the driver loading procedure can both load and reload drivers and perform the I/O data base initialization that is appropriate to either situation.

CHAPTER 2

DISCUSSION OF A LINE PRINTER QUEUE I/O REQUEST

The LP11 is a buffered line printer. A user process can request the following functions for this printer:

- Write data to the line printer
- Read the line printer's device characteristics
- Alter the line printer's device characteristics

This chapter describes the following aspects of line printer I/O processing:

- The portions of the VAX/VMS device driver for an LP11 line printer that are used in servicing a write request
- The VAX/VMS components with which the driver interacts to process the write request

The LP11 was selected for this discussion because it is a simple driver but still illustrates many driver principles. Although the LP11 is usually spooled, for purposes of this discussion, assume that it is not spooled.

The first-time reader of this document may not understand all of the points made in this chapter; however, the chapter should provide some insight into driver flow and I/O processing.

Figure 2-1 illustrates the flow of execution through VAX/VMS routines and the line printer driver to satisfy this I/O request.

The double-sided boxes in Figure 2-1 indicate processing performed by driver subroutines. Boxes shown above the dotted line indicate processing in the context of the user process. Boxes below the dotted line indicate processing in fork or interrupt context.

2.1 DRIVER CODE FOR THE LP11 WRITE FUNCTION

The VAX/VMS device driver for an LP11 line printer implements a write function using the following parts of the driver:

- An FDT routine that reformats the user-supplied data
- A driver start I/O routine that writes data to the device print buffer until the printer enters a busy state to print the contents of the buffer

- Code that modifies a device register to enable interrupts from the line printer
- A driver interrupt service routine that returns control to the driver fork process after a hardware interrupt from the line printer
- Code that returns I/O status to a VAX/VMS I/O completion routine

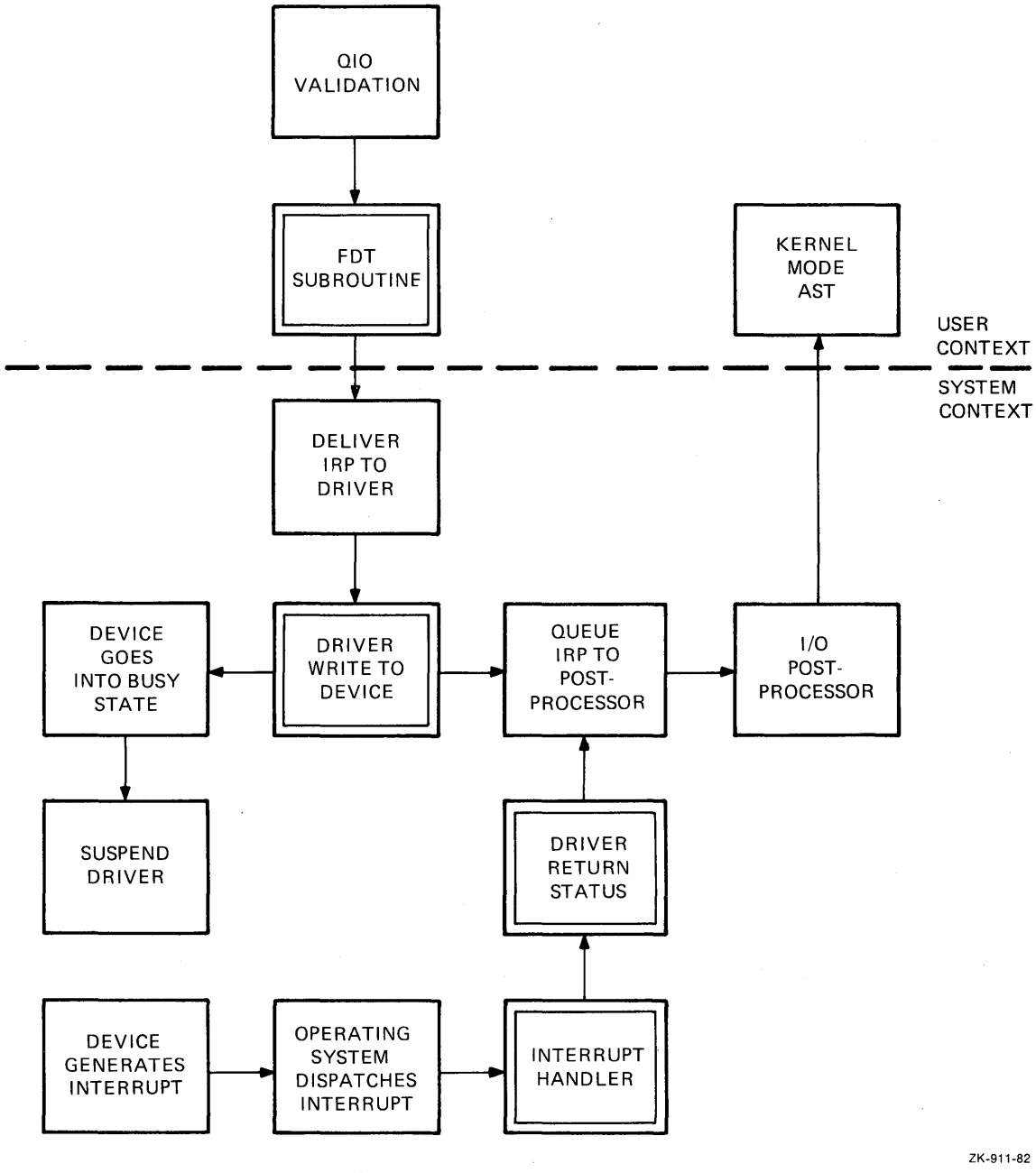


Figure 2-1: A Line Printer Write Function

2.2 A USER PROCESS'S I/O REQUEST

A user process writes a line to the printer by issuing a Queue I/O Request system service call that specifies the write virtual block function code, as follows:

```
$QIO_S      CHAN = CHANNEL_NUMBER,-
            FUNC = #IO$_WRITEVBLK,-
            EFN = #6,-
            IOSB = STATUS_BLOCK,-
            P1 = BUFFER_ADDRESS,-
            P2 = #BUFFER_SIZE,-
            P4 = #^X30
```

The parameters P1, P2, and P4 are device-dependent parameters.

2.3 I/O PREPROCESSING BY VAX/VMS

When called, the Queue I/O Request system service first validates that the I/O request is correctly specified; that is, the I/O request must meet the following criteria:

- The location CHANNEL_NUMBER must contain a channel number that serves as an index into the process I/O channel list. The process must have previously assigned the line printer device to this process channel using the Assign I/O Channel system service.

  During verification of the channel number, the Queue I/O Request system service obtains the address of the line printer driver's function decision table (FDT). Figure 2-2 illustrates the chain of pointers from the channel index number to the FDT address. As a result of chaining through the I/O data base, the Queue I/O Request system service also determines what device is the target of the request.

- The line printer FDT must list IO$_WRITEVBLK as a valid function for the device.

- The event flag number must be valid.

- The process buffered I/O request quota must permit the Queue I/O Request system service to perform a buffered I/O request without exceeding the process's quotas.

- The process must have write access to the user-specified location to be used as an I/O status block.

If all of the checks described above succeed, the Queue I/O Request system service creates an I/O request packet in nonpaged system address space. The service then writes all known details about the I/O request into the I/O request packet.

If the target device for the I/O request is not file-structured, the Queue I/O Request system service changes any virtual function code to its logical equivalent when it builds the I/O request packet. Thus, for a line printer device, IO$_WRITEVBLK is translated to IO$_WRITELBLK.

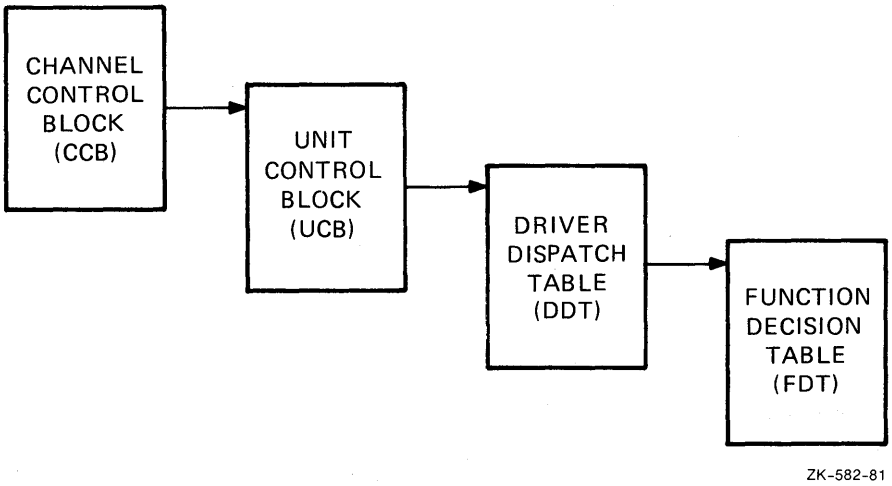


Figure 2-2: Locating a Function Decision Table

2.4 I/O PREPROCESSING BY THE DRIVER

Once it has validated the I/O request, the Queue I/O Request system service scans the function decision table for an entry that associates the IO$_WRITELBLK function code with an FDT routine. The system service calls the routine, which in the case of the line printer driver is a device-specific routine located in the line printer device driver.

The FDT routine confirms that the requesting process has read access to the buffer starting at BUFFER_ADDRESS. Then, the FDT routine buffers data from the process address space into system address space in the following steps:

- It calculates the length of the required system space buffer.
- If the process byte count quota for buffered I/O (BYTCNT) permits, the routine allocates a buffer from system address space, stores the address of the buffer in the I/O request packet, and decreases the current process byte count quota.
- It then synchronizes with other possible subprocesses[1] to read and write fields of the line printer's unit control block.
- It reads the description of the line printer's current line and page position from the device's unit control block.
- It reformats the data from the process buffer into the system buffer, adding carriage control characters, as specified in the I/O request argument P4, before and after the data.

  Formatting includes such functions as the replacement of horizontal tabs with multiple spaces and the replacement of lowercase characters with uppercase characters, if necessary.

---

1. For example, if a process allocates a printer, it is possible for the process and any of its subprocesses to issue write requests to the printer concurrently.

- It rewrites updated line and page positions into the device's UCB. This information indicates what the current location on the page being printed will be where the request completes.

- Finally, the routine transfers control to a VAX/VMS routine that queues the I/O packet to the device driver.

All of the I/O processing described to this point occurs in the context of the user's process. The user address space is mapped, and the processor's interrupt priority level (IPL) is still low enough to permit process scheduling and paging. Subsequent queuing of the transfer request to the driver and all resulting driver processing occur at higher interrupt priority levels that synchronize driver handling of the device, as described in Chapter 3.

2.5 QUEUING THE I/O PACKET TO THE DRIVER

Before queuing the I/O request packet to the proper driver, the VAX/VMS queuing routine raises the interrupt priority level to the driver fork level (UCB$B_FIPL) stored in the unit control block. Raising IPL to fork level synchronizes driver access to the unit control block.

If the device is idle, that is, if the busy bit (UCB$V_BSY) in the I/O status word of the unit control block is clear, VAX/VMS can transfer control to the driver. The driver dispatch table contains the entry point to the driver's start I/O routine. To find the proper entry point, the queuing routine chains through the I/O data base to the driver dispatch table, as follows:

UCB ⟶ DDT ⟶ Entry point to start I/O routine

If the device unit is busy with another transfer, VAX/VMS inserts the I/O request packet in a queue of packets waiting for the unit. The unit control block contains the head of the queue. The packet's position in the queue depends on the scheduling priority of the process issuing the request.

2.6 DRIVER DEVICE ACTIVATION

The LP11 line printer controller accepts data into a device data buffer until the print buffer is full or the driver writes a carriage control character into the print buffer. When either event occurs, the line printer sets a busy bit in the device's control/status register. Then a device driver sets the interrupt enable bit in the device's control/status register and waits for the printer to interrupt. When the line printer requests a hardware interrupt, the driver can resume putting characters in the print buffer.

The line printer driver routine writes to the line printer data buffer according to the following sequence:

1. The driver locates the LP11 device registers using a chain of pointers starting at the device's unit control block (UCB).

   UCB ⟶ CRB ⟶ IDB ⟶ CSR address

   The CSR address is always the address of the line printer control/status register, and all other device registers are at fixed offsets from this address. In contrast to many

other devices, such as disks, the LP11 line printer does not share a controller with other devices; therefore, no arbitration for ownership of the controller is required.

2. The driver examines the device's control/status register to see if the device is ready to accept characters.

3. If the device is ready, the driver writes a byte of data into the line printer data buffer and decreases the count of bytes to transfer. It then repeats step 2.

4. If the device is not ready, that is, if the device's internal buffer is full, the driver raises IPL to 31 to block out all interrupts and sets the interrupt enable bit in the device's control/status register.

   After enabling interrupts, the driver invokes a VAX/VMS wait for interrupt macro to suspend driver processing until the line printer requests an interrupt or the device times out.

2.7 WAITING FOR A DEVICE INTERRUPT

The VAX/VMS wait for interrupt routine suspends the driver by performing the following functions:

- Saving driver context (R3, R4, and the address of the next instruction in the driver) in the device's unit control block
- Calculating the time at which the device will time out
- Setting bits in the device's unit control block to indicate that the driver expects a device interrupt within a specified time period

VAX/VMS then drops IPL back to driver fork level and returns control to the caller of the driver's start I/O routine.

The driver remains in a suspended state until one of two events occurs:

- The line printer requests a hardware interrupt.
- VAX/VMS reports a device timeout because the line printer did not request a hardware interrupt within a specified period of time.

Normally, the LP11 prints the contents of its data buffer and requests the interrupt.

2.8 INTERRUPT HANDLING

When the LP11 line printer requests a hardware interrupt, the interrupt dispatcher passes the interrupt to the LP11 driver interrupt service routine.

The driver's interrupt service routine restores control to the driver, as follows:

- Restores the address of the unit control block in R5
- Confirms that the interrupt was expected by examining bits in the device's unit control block

- Restores the saved registers (R3 and R4) from the device's unit control block
- Transfers control to the driver PC address stored in the device's unit control block

Rather than execute in interrupt context, the reactivated driver routine calls a VAX/VMS routine to create a driver fork process. VAX/VMS again suspends driver processing by performing the following steps:

- Saving driver context (R3, R4, and the driver PC address) in the device's unit control block
- Inserting the UCB address in the appropriate fork queue

The driver suspension allows the operating system to reschedule driver processing at a lower IPL. A VAX/VMS fork dispatcher reactivates the driver when IPL drops to driver fork level.

After creating the fork process, the system returns control to the driver's interrupt service routine. which performs the following steps:

- Restores registers saved at the time of the device interrupt
- Dismisses the interrupt

2.9 I/O COMPLETION PROCESSING BY THE DRIVER

When the VAX/VMS fork dispatcher reactivates the driver fork process, the driver code continues transferring characters into the line printer data buffer until the transfer is complete. The driver code performs the following steps to transfer characters:

- It obtains the number of characters left to transfer from the unit control block.
- It transfers characters until the LP11 again prints its data buffer or all characters have been transferred.
- When all characters have been transferred, the driver code branches to driver I/O completion code.

The driver's I/O completion code stores the following information in R0:

- A success status code
- The number of bytes transferred

Then, the driver code transfers control to VAX/VMS to complete the I/O request.

2.10 I/O COMPLETION PROCESSING BY THE VAX/VMS SYSTEM

The operating system inserts the I/O request packet into an I/O postprocessing queue. If another I/O request packet is in the wait queue for the device unit, VAX/VMS dequeues that packet and calls the driver start I/O routine to process it.

When IPL drops to IPL$_IOPOST, the processor grants the I/O postprocessing interrupt request. The I/O postprocessing dispatcher dequeues the packet for the line printer I/O request and performs the following steps:

- It increases the use count of the process's buffered I/O requests since the current operation is complete. The use count is maintained for accounting purposes.
- It deallocates the system buffer used for the reformatted user data.
- It increases the process's current byte count quota.
- It sets an event flag to indicate that the I/O operation is complete.
- It queues a kernel mode AST routine that will deallocate the I/O request packet and stores I/O status into the user's I/O status block.

The user process examines the event flag or issues a Wait for Single Event Flag system service call to determine that the I/O operation is complete.

CHAPTER 3

SYNCHRONIZATION OF I/O REQUEST PROCESSING

The VAX/VMS operating system uses three mechanisms to synchronize I/O processing:

- Hardware interrupt priority levels and interrupt service routines
- Driver fork processes, fork blocks, and fork queues
- Resource wait queues

When programming a driver, you must observe the VAX/VMS conventions that govern the use of interrupt priority levels and fork processes. The VAX/VMS routines that grant resources to drivers enforce the use of resource wait queues.

3.1 INTERRUPT PRIORITY LEVELS

The VAX-11 processor defines 32 levels of hardware priorities, called interrupt priority levels (IPLs). IPL 0 has the lowest priority, and IPL 31 has the highest. Interrupts can be requested either by software (software interrupts) or by the hardware (hardware interrupts). The system uses the various interrupt priority levels as follows:

- User mode software runs at IPL 0.
- Operating system routines and driver fork processes request software interrupts at IPLs 1 through 15.
- Devices and error conditions generate hardware interrupts at IPLs 16 through 31.

Many IPLs have an interrupt service routine associated with them. The processor responds to both software and hardware interrupts by transferring control to the appropriate interrupt service routine. The interrupt service routine processes the interrupt and, when finished, dismisses the interrupt with an REI instruction.

3.1.1 IPLs Defined by VAX/VMS

Table 3-1 describes the uses that VAX/VMS defines for IPLs 0 through 15.

Table 3-1: IPLs Defined by VAX/VMS

IPL	Symbolic Name	Use
0	--	User mode software
1	--	Reserved
2	IPL$_ASTDEL	AST delivery interrupt service routine
3	IPL$_SCHED	Scheduler interrupt service routine
4	IPL$_IOPOST	I/O postprocessing interrupt service routine
5	IPL$_XDELTA	XDELTA interrupt service routine
6	IPL$_QUEUEAST	Fork level processing for queuing ASTs
7	IPL$_SYNCH IPL$_TIMER	System data base access and software timer interrupt service routine
8 - 11	UCB$B_FIPL	Fork level for driver execution
12 - 15		Reserved

3.1.2 IPLs Defined for the Hardware

Hardware interrupt levels are used for device interrupts (IPLs 20 through 23) and urgent conditions including power failure and serious errors such as a machine check. The VAX Hardware Handbook provides additional information about hardware interrupt levels.

3.1.3 Interrupt Service Routines

The VAX/VMS operating system uses interrupt service routines that gain control at the preset IPLs described above. Using preset IPLs guarantees that interrupts are processed according to the following priorities:

- Device interrupts (highest priority)
- Device driver fork processes
- I/O postprocessing
- Process scheduling
- AST delivery (lowest priority)

For example, VAX/VMS completes the processing of an I/O request by placing the I/O request packet in the I/O postprocessing queue and requesting an interrupt at the I/O postprocessing IPL (IPL 4). When the interrupt priority level drops below 4, the processor grants the software interrupt by transferring control to the I/O postprocessing service routine.

Interrupt service routines run in a reduced context. The stack is a special stack used only during interrupt processing; it is the interrupt stack. Of the register set, usually only R0 through R5 are saved. The interrupt service routine must restore these registers before it returns from an interrupt. If the service routine uses any other registers, the routine must save the registers before use and restore them after use. Using registers other than R0 through R5 is not recommended.

When a hardware interrupt occurs, the system transfers control to the driver interrupt service routine with IPL set to the hardware device interrupt level. Since code executing at IPLs 20 through 23 blocks most other hardware interrupts and all software interrupts, driver code lowers its IPL as soon as possible.

The operating system allows the creation of a fork process so that a driver can continue execution without blocking other device interrupts. Section 3.2 discusses fork processes.

3.1.4 Raising IPL

Code running in kernel mode can raise its IPL to lock out context switching and block interrupts. VAX/VMS software interrupt service routines perform some of their processing at IPLs higher than the IPL at which the routines gain control. For example, the scheduler is an interrupt service routine that gains control at IPL 3; however, it raises IPL to 7 to read and modify the system data base. I/O drivers typically raise IPL to check for a power failure, send a message to a mailbox, and sometimes to access device registers. Driver code should not raise IPL for more than a few instructions because so doing blocks all interrupts at lower IPLs.

3.1.5 Lowering IPL

Once an interrupt service routine has received the interrupt, it transfers control to the main flow of driver code. At this point, the driver is executing in the context of an interrupt service routine and at device IPL.

When a driver gains control, it may execute a few instructions at the high IPL; however, almost immediately a driver lowers IPL to fork IPL. A driver lowers IPL by invoking the VAX/VMS macro that creates fork processes, IOFORK. As a result of invoking IOFORK, VAX/VMS performs the following functions for the driver:

- Consults the device's unit control block to determine fork IPL for the driver
- Creates a driver fork process and queues it for execution at the appropriate IPL
- Requests a software interrupt at that IPL

When the queued driver fork process is reactivated, it executes at the lower fork IPL. Section 3.2 describes fork process dispatching in greater detail.

Driver fork processes also can modify IPL by invoking certain VAX/VMS macros; Section 3.1.11 describes these macros. Normally, a driver uses these macros to raise IPL before initiating a transfer.

3.1.6 Dispatching Device Interrupts

VAX-11 peripheral devices request interrupts at IPLs 20 through 23. When a device requests an interrupt at one of these IPLs and the processor is executing at a lower IPL, the processor performs the following steps:

- Grants the interrupt
- Transfers control to an interrupt service routine for the device

If the processor is executing at a higher or equal IPL, the interrupt remains pending.

The dispatching of UNIBUS device interrupts differs depending upon the type of processor and UNIBUS adapter in the hardware configuration.

When an interrupt occurs on a configuration that uses the nondirect vector UNIBUS adapter, the processor transfers control to an interrupt service routine for the UNIBUS adapter of the device that requested the interrupt. The UNIBUS adapter interrupt service routine then carries out the following steps:

- Saves R0 through R5 on the interrupt stack
- Reads a UNIBUS adapter register to determine the vector address of the device requesting the interrupt
- Uses the vector address as in index into a vector jump table within the UNIBUS adapter control block. The vector jump table contains a list of channel request block addresses that point to the interrupt service routines for all the devices attached to that UNIBUS.
- Transfers control to the channel request block (CRB) address that corresponds to the vector address.

The CRB address contains a JSB instruction that passes control to the device's interrupt service routine. Figure 3-1 shows a flowchart that details interrupt dispatching of a nondirect vector interrupt.

On a configuration that supports direct vector interrupts, the UNIBUS adapter does not dispatch the interrupt. Instead, the processor locates the device's interrupt service routine by using the system control block (SCB). The system control block consists of two or three pages of addresses. Page one lists the exception vectors; pages two and three contain the list of CRB addresses that point to the interrupt service routines for devices attached to the first UNIBUS and an optional second UNIBUS, respectively. The SCB base register (SCBB), an internal processor register, marks the base of the system control block.

The processor obtains the vector address of the device that requested the interrupt and uses it as an index into page two (or page three) of the SCB. When it finds the corresponding CRB address, the processor transfers control to the interrupt dispatching code in the device's channel request block. On direct vector configurations, the interrupt dispatch code is a PUSHR instruction of R0 through R5 followed by the JSB to the device's interrupt service routine.

Figure 3-2 shows a flowchart of interrupt dispatching on a direct vector UNIBUS adapter.

To maintain machine-independent descriptions of interrupt handling, subsequent chapters in this manual refer to the combination of hardware and software that transfers device interrupts to the device's interrupt service routine as the interrupt dispatcher.

3.1.7 Transferring Control to the Driver Fork Process

When a device driver receives an expected interrupt from a device, the driver interrupt service routine executes in the context of an interrupt; it is not executing in driver fork process context at that point. Interrupt context has the following characteristics:

- IPL is elevated to the level at which the device requests hardware interrupts.
- The stack is the interrupt stack.
- The top of the stack contains a pointer to the address of the controller's interrupt dispatch block (IDB), which contains the address of the control/status register.
- The stack also contains saved R0 through R5 and the PC and PSL of the interrupted code.

The interrupt occurs either because the device has completed an I/O operation or because an error occurred during the I/O operation. Driver interrupt service routines generally determine whether to service the interrupt by examining the I/O data base. If the unit control block for the device that currently owns the controller indicates that the interrupt is expected, the service routine takes the following steps to transfer control to the driver's start I/O routine:

- Loads the UCB address into R5
- Restores the contents of two registers (R3 and R4) from the UCB fork block
- Returns control to the saved PC in that fork block

The driver may need to execute a few instructions in the context of the interrupt. For example, the driver may copy device status information from device registers into the device's unit control block. After executing these instructions at device IPL, the driver completes the I/O processing at a lower priority by creating a fork process, as described in Section 3.2.

3.1.8 IPL Use During I/O Processing

I/O processing occurs mainly at the following IPLs:

- IPL$_ASTDEL (IPL 2)
- IPL$_IOPOST (IPL 4)
- Driver fork processing IPLs (IPLs 8 through 11)
- Hardware device IPLs (IPLs 20 through 23)
- IPL$_POWER (IPL 31)

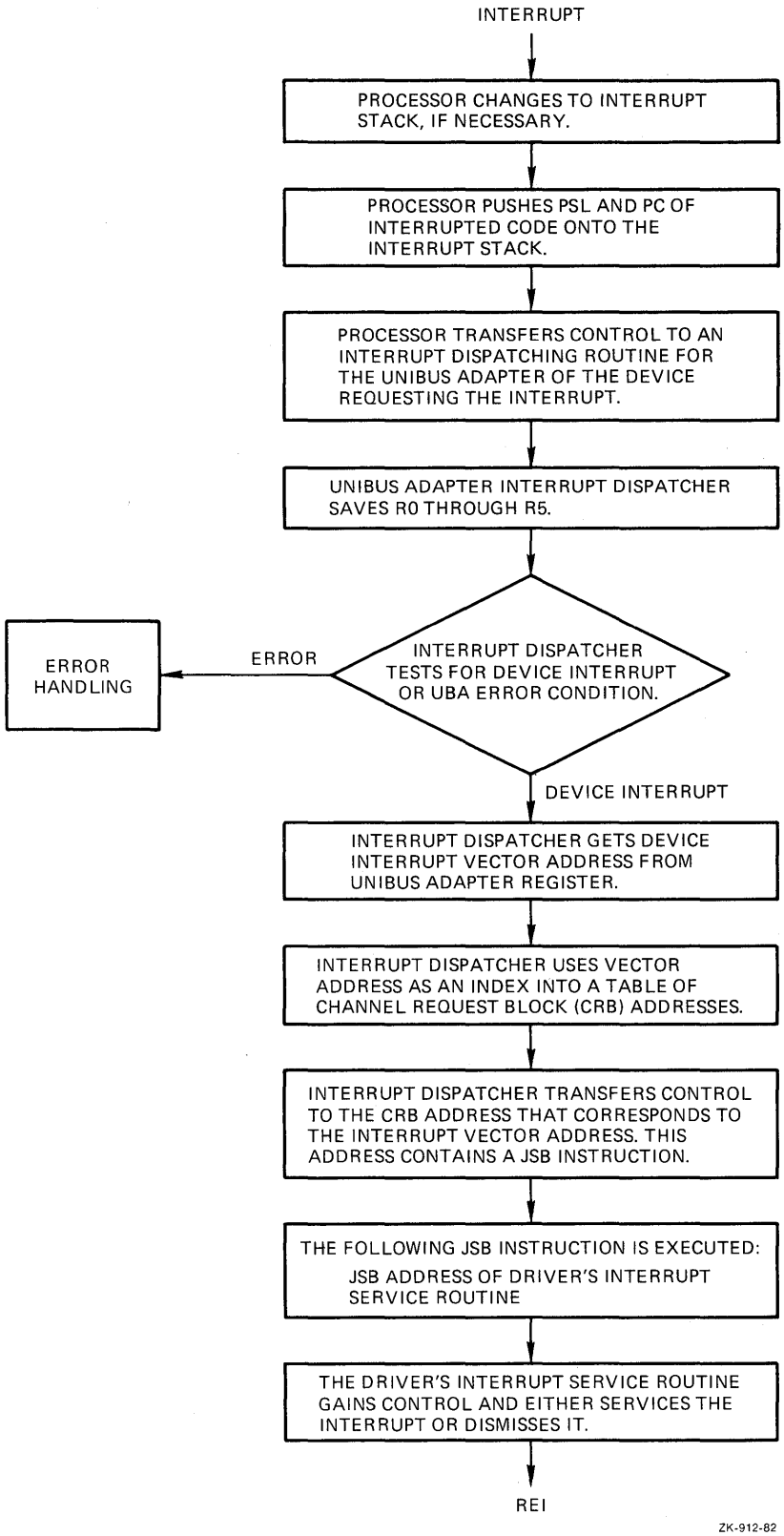


Figure 3-1: Interrupt Dispatching of a Nondirect Vector Interrupt

3.1.8.1 IPL$_ASTDEL (IPL 2) - IPL$_ASTDEL blocks the delivery of asynchronous system traps (ASTs). When a system service for which an AST was specified completes, the system service queues the AST and causes a software interrupt to be requested at IPL$_ASTDEL. The AST delivery interrupt service routine gains control when IPL drops below IPL$_ASTDEL. It delivers the AST to the process that is currently scheduled.

Any driver routine that allocates or deallocates dynamic system pool space while running in the context of a process (for example, an FDT routine) must do so at an IPL of IPL$_ASTDEL or higher. The VAX/VMS allocation routine records the address of the allocated system memory in a register. If an AST that aborts the process were to occur, the allocated memory would be lost from the pool. To block ASTs, I/O preprocessing from the time that the Queue I/O Request system service allocates an I/O request packet through the execution of the last FDT routine occurs at IPLs no lower than IPL$_ASTDEL.

A process cannot incur page faults when IPL is above IPL$_ASTDEL. Any code that executes at a higher IPL must refer only to nonpaged virtual memory or pages that have been locked in virtual memory. A fatal bugcheck occurs if a page fault is incurred above IPL$_ASTDEL.

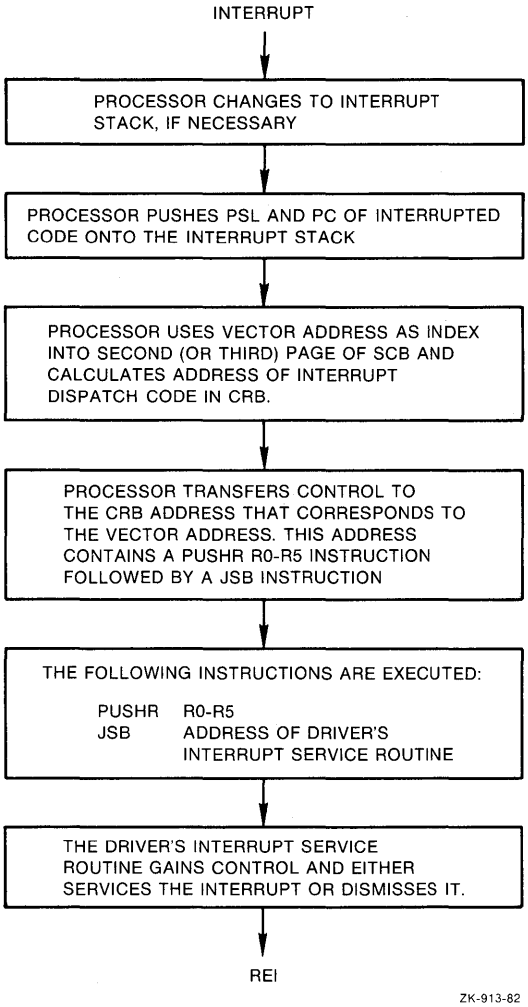


Figure 3-2: Interrupt Dispatching of a Direct Vector Interrupt

In addition, some I/O postprocessing occurs in a kernel mode AST service routine that also executes at IPL$_ASTDEL. Kernel mode ASTs, running in the context of a process whose I/O completed, write status information into I/O status blocks, copy buffered input into process space, and deallocate system buffers.

3.1.8.2 IPL$_IOPOST (IPL 4) - I/O postprocessing includes all I/O completion processing that can occur without reference to the device's unit control block and, thus, can occur at an IPL lower than driver fork IPL. To request I/O postprocessing, drivers call a VAX/VMS routine that inserts I/O request packets in the postprocessing queue and requests a software interrupt at IPL$_IOPOST.

I/O postprocessing runs at an IPL higher than IPL$_SCHED so that all pending I/O completion processing is finished before the scheduler looks for a new process to schedule. Whether a process is awaiting I/O completion affects its ability to execute. Since I/O postprocessing queues ASTs to processes, the scheduler may preferentially reschedule a waiting process because of a pending AST to the process.

The VAX/VMS operating system performs I/O postprocessing in the IPL 4 interrupt service routine. This routine adjusts process quota use, queues a kernel mode AST to write status and data into the process's address space, and deallocates system memory.

3.1.8.3 Driver Fork Processing (IPLs 8 through 11) - Driver fork processing occurs at an IPL in the range 8 through 11 depending on the contents of the unit control block field UCB$B_FIPL. UCB$B_FIPL contains a value that is used as that device's fork IPL. All driver routines, except for most FDT routines, execute at driver fork IPL or higher. Usually driver routines should not read or alter fields of the unit control block unless IPL is at fork level or higher.

A driver must never lower IPL below the IPL of the interrupt that caused the driver to be reentered unless the driver does so by creating a fork process at the lower IPL.

All devices on a single UNIBUS adapter share the same fork IPL if they actively compete for shared UNIBUS adapter resources such as map registers and data paths.

3.1.8.4 Hardware Device Interrupts - The UCB$B_DIPL field in the device's unit control block contains an IPL value at which the device requests hardware interrupts. This IPL is in the range 20 through 23 because device interrupts usually need to interrupt most user and VAX/VMS software functions. IPLs 20 through 23 correspond to UNIBUS bus request (BR) levels 4 through 7. Device drivers sometimes raise IPL to UCB$B_DIPL or higher before reading and writing certain device registers.

3.1.8.5 IPL$_POWER - The highest IPL, IPL$_POWER, locks out all other interrupts. Many VAX/VMS 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.

When a device driver needs to execute a series of instructions without interruption, the driver raises IPL to IPL$_POWER. The driver never should remain at IPL$_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 between the time that the driver writes set-up data into device registers and the time that the driver starts the device by writing into the device control register.

3.1.9 Additional IPLs

In addition to the IPLs described above, VAX/VMS defines the following:

- IPL$_SCHED (IPL 3); never used by drivers
- IPL$_QUEUEAST (IPL 6); very seldom used by drivers
- IPL$_SYNCH and IPL$_TIMER (IPL 7); very seldom used by drivers
- IPL$_MAILBOX (IPL 11); very seldom used by drivers

For debugging purposes, the VAX/VMS operating system defines the priority level IPL$_XDELTA (IPL 5); it is described in Section 3.1.9.5.

3.1.9.1 IPL$_SCHED - When the system wishes to reschedule processes, a VAX/VMS routine requests a software interrupt at IPL$_SCHED. The scheduler interrupt service routine gains control at this IPL.

If a process raises IPL to or above IPL$_SCHED, the scheduler cannot reschedule the processor. The process runs until an interrupt occurs at a higher IPL or the process reduces IPL below IPL$_SCHED.

3.1.9.2 IPL$_QUEUEAST - IPL$_QUEUEAST is a fork level IPL. That is, the interrupt service routine for IPL$_QUEUEAST is the fork dispatcher that dequeues fork blocks and restores control to fork processes needing to execute at IPL$_QUEUEAST.

To queue an AST, a driver creates a fork process at IPL$_QUEUEAST. When the fork dispatcher restores control to the fork process, the process can raise IPL to IPL$_SYNCH and queue the AST.

A driver that wishes to gain access to the system data base for any reason can also create a fork process at IPL$_QUEUEAST. The fork dispatcher restores control to the driver at IPL$_QUEUEAST, and the driver can then raise IPL to IPL$_SYNCH (a nonfork IPL) to gain access to the system data base.

3.1.9.3 IPL$_SYNCH and IPL$_TIMER - IPL$_SYNCH is the system data base synchronization level. When a VAX/VMS subroutine or a driver needs to modify or read a dynamic portion of the system data base, the routine always executes at IPL$_SYNCH to ensure that the data base does not change due to some interrupt service routine or process action.

A timer queue interrupt service routine fields interrupts requested at IPL$_TIMER, which is also IPL 7. The hardware clock interrupt service routine requests a software timer interrupt at IPL$_TIMER when the current process has exceeded its processor time quantum or when the first entry in the timer queue is due. The timer interrupt service routine dequeues the first timer queue entry and takes appropriate action.

3.1.9.4 IPL$_MAILBOX - When a VAX/VMS or driver routine writes into a mailbox, IPL must be at IPL$_MAILBOX to prevent other writers from modifying incomplete data in the mailbox, or readers from reading invalid data.

IPL$_MAILBOX is the highest fork level; drivers can raise IPL to IPL$_MAILBOX and write into a mailbox.

3.1.9.5 IPL$_XDELTA - To stop the operating system for debugging purposes, you can halt the operating system from the console terminal and request a software interrupt at IPL$_XDELTA. The processor must be executing below IPL 5 for the interrupt to have an effect. Chapter 15 describes the XDELTA debugging program.

3.1.10 Overview of IPL Use

Figure 3-3 illustrates the normal IPL flow during the processing of an I/O request.

The user program, executing at IPL 0, issues a Queue I/O Request system service call. I/O processing by the system service and FDT routines occurs mostly at IPL$_ASTDEL. Very rarely, an FDT routine raises IPL to driver fork level to read or modify the device's unit control block.

The start I/O routine executes as a fork process at fork IPL, but may raise to device interrupt IPL or IPL$_POWER for short periods of time. After the driver fork process activates the device, the driver calls a VAX/VMS routine that saves the driver fork context, suspends driver fork processing, and restores IPL to a previous level.

Figure 3-3 illustrates the completion of the I/O request from the point of the device interrupt to the delivery of ASTs to the user program. The device interrupts at a device IPL (in the range 20 through 23). VAX/VMS transfers control to the appropriate driver interrupt service routine. The service routine reactivates the driver fork process with IPL still at hardware device IPL.

The fork process briefly examines or saves the contents of device registers, but soon requests that VAX/VMS insert a fork block describing its context into one of the fork queues for driver fork IPLs (8 through 11). When the driver fork process regains control at driver fork IPL, the process analyzes the success of the I/O operation and writes status into R0 and R1. Then, still at driver fork IPL, VAX/VMS inserts the I/O request packet into the I/O postprocessing queue and starts the next I/O request.

The I/O postprocessing routine adjusts process quota usage and deallocates system buffers for write functions at IPL$_IOPOST. The routine also calls another VAX/VMS routine that raises IPL to IPL$_SYNCH to queue a kernel mode AST to the process that issued the

original QIO request. The AST routine executes at IPL$_ASTDEL, and may queue a user AST routine that eventually executes at an IPL of 0. I/O postprocessing continues at IPL$_IOPOST until all entries in the postprocessing queue have been serviced.

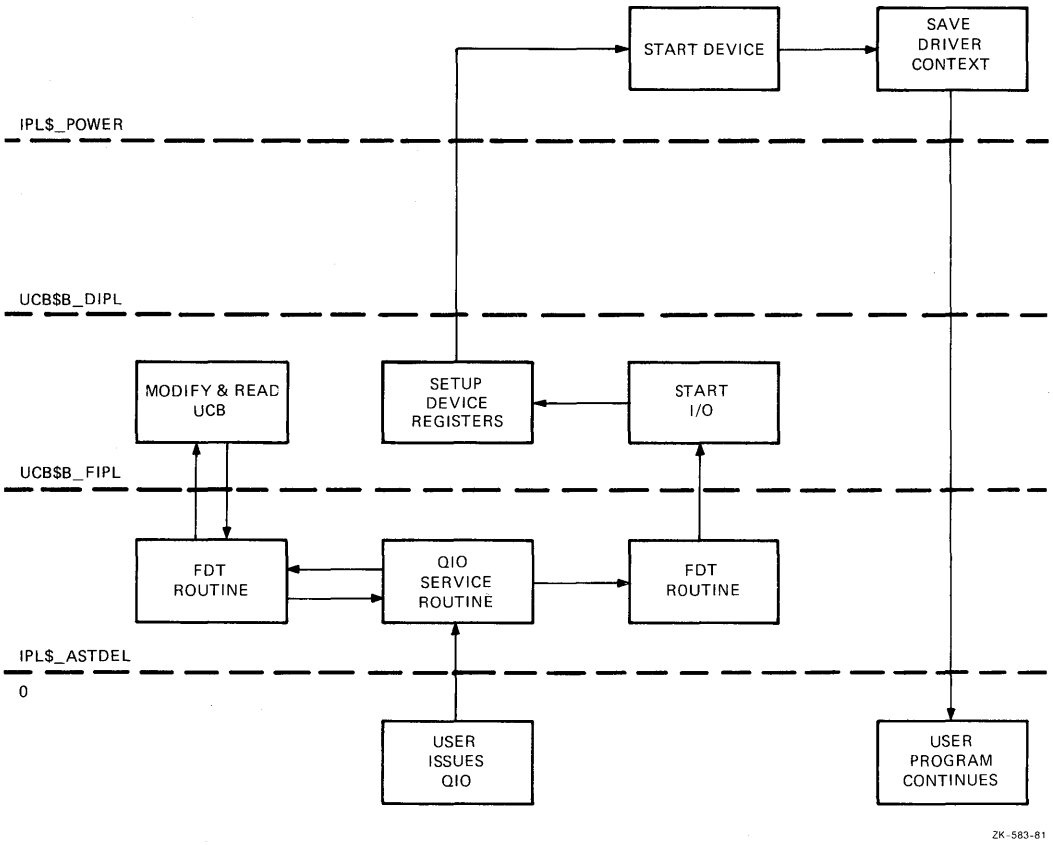


Figure 3-3: IPL Conventions During I/O Processing

3.1.11 Modifying IPL in Driver Code

The interrupt priority level at which driver code executes changes as a result of either of the following events:

- The driver's calling a VAX/VMS routine that raises or lowers IPL
- The driver's invoking a VAX/VMS macro to request explicitly a change in IPL

Subsequent chapters of this manual discuss the VAX/VMS routines that change IPL; discussions include their expectation of IPL at entry and their IPL setting at exit. The sections that follow describe the macros that drivers can call to change IPL:

- SETIPL
- DSBINT
- ENBINT
- SOFTINT

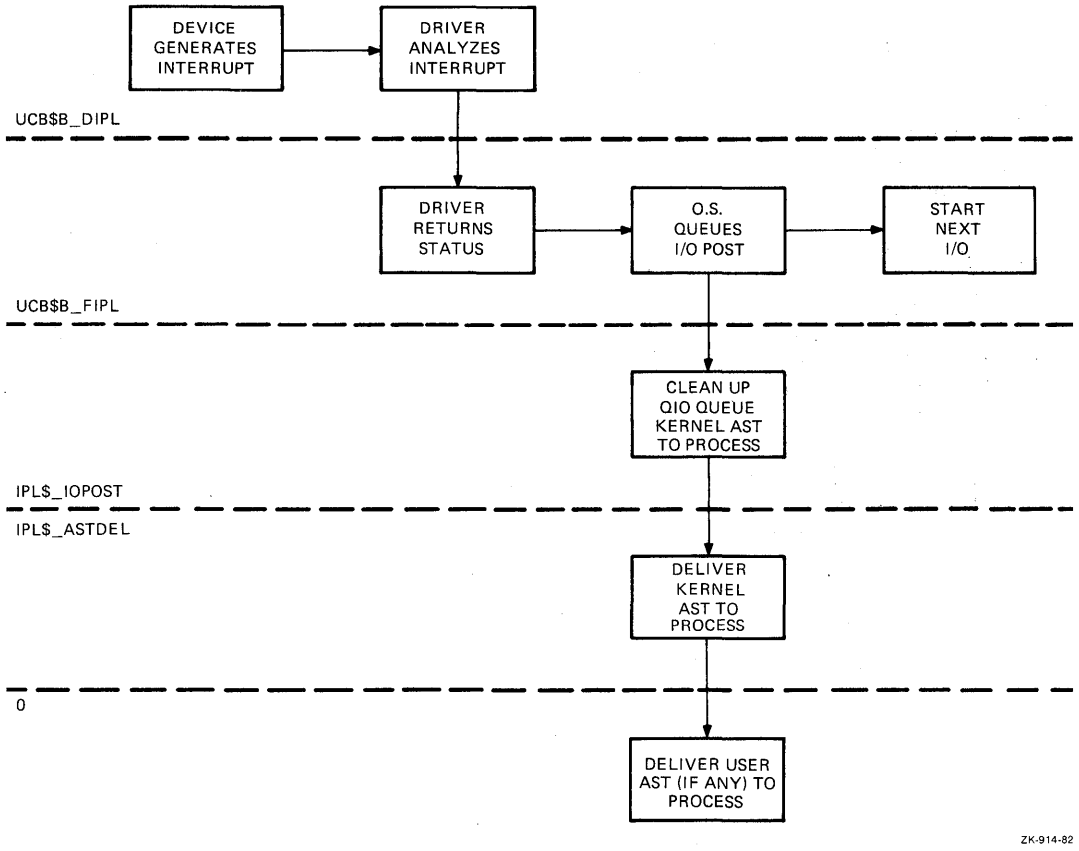


Figure 3-4: IPL Conventions During I/O Completion

3.1.11.1 Set Interrupt Priority Level Macro - The Set Interrupt Priority Level (SETIPL) macro moves the specified IPL into the IPL processor register.

Format

```
SETIPL [ipl]
```

ipl
: The interrupt priority level. If no priority level is specified, the macro moves the value 31 into the IPL register. Setting IPL to 31 blocks all interrupts.

3.1.11.2 Disable Interrupts Macro - The Disable Interrupts (DSBINT) macro saves the current IPL in the specified destination and moves the specified IPL into the IPL processor register. Procedures invoke this macro to raise IPL.

Format

```
DSBINT [ipl] [,dst]
```

ipl
: The interrupt priority level. The macro saves the current IPL on the top of the stack (default) or in the specified destination and moves the specified IPL into the IPL register. If IPL is not specified, the macro moves the value 31 into the IPL processor register; this blocks all interrupts.

dst
The location in which the current IPL is to be saved. If this argument is not specified, the current IPL is stored on the top of the stack by default.

3.1.11.3 Enable Interrupts Macro - The Enable Interrupts (ENBINT) macro restores an IPL value to the IPL processor register. Procedures invoke this macro to lower IPL to a previously saved level. If an interrupt is pending at an intermediate IPL (that is, one lower than the current IPL but higher than the specified IPL), restoring IPL causes immediate interruption of the current procedure.

Format

ENBINT [src]

src
The location containing the IPL to be restored. If this argument is not specified, the macro moves the IPL value contained on the top of the stack into the IPL register.

3.1.11.4 Software Interrupt Macro - The Software Interrupt (SOFTINT) macro moves the specified IPL into the software interrupt request processor register to request a software interrupt. If the processor is executing at a low IPL (for example, IPL 0) and detects a software interrupt request at a higher IPL (1 through 15), the processor immediately transfers control to a software interrupt service routine for the appropriate IPL. If the processor is executing at or above the specified IPL, the processor does not transfer control to the software interrupt service routine until IPL drops below the specified IPL.

Format

SOFTINT ipl

ipl
The interrupt priority level at which the software interrupt is being requested.

3.2 FORK BLOCKS AND FORK DISPATCHING

Device driver routines that activate a device and complete an I/O operation after a device interrupt execute for relatively short periods of time. Execution may be suspended to wait for a device interrupt or shared resources. To ensure that the resulting context switching is fast, VAX/VMS forces driver routines to execute in a minimal fork process context consisting of a device UCB, called a fork block, and a few registers.

Driver fork processes are created in either of the following situations:

- Once the preprocessing of an I/O packet has been performed, a VAX/VMS routine creates a fork process to execute the driver's start I/O routine. If the driver is already busy, the VAX/VMS routine queues the I/O packet for the driver to process later.

- Either the driver's interrupt service routine or the driver postprocessing routine creates a fork process to perform device-dependent I/O postprocessing.

When the system creates a driver fork process to execute the start I/O routine, the newly created fork process can execute immediately because the I/O packet has been preprocessed by the Queue I/O Request system service and driver FDT routines, and the device is idle.

When the driver interrupt service routine or the driver postprocessing routine creates a driver fork process, it does so to lower the IPL of the driver code. Either the service routine or the driver invokes the VAX/VMS macro IOFORK. IOFORK saves the context needed for the driver to execute as a fork process, inserts the driver's UCB fork block in the fork queue for the driver's IPL, and requests a software interrupt for that IPL.

3.2.1 Interrupt Service Routine for Fork Dispatching

One interrupt service routine handles all fork process dispatching. When the processor grants an interrupt at fork IPL, the fork dispatcher saves R0 through R5 on the stack and processes the fork queue that corresponds to the IPL of the interrupt. To do so, it removes an entry from the fork queue, restores the fork process context, and reactivates the suspended fork process. When that fork process completes, the dispatcher regains control, removes the next entry, if any, from the queue, restores its fork process context, and reactivates it. This sequence repeats until the fork queue is empty. When the queue is empty, the fork dispatcher restores R0 through R5 from the stack and dismisses the interrupt with an REI instruction.

Figure 3-5 illustrates the fork queue structure.

A newly activated driver fork process executes under the following constraints:

- It cannot refer to the address space of the process initiating the I/O request.
- It can use only R0 through R5 freely; it must save other registers before use and restore them after use. Use of registers other than R0 through R5 is strongly discouraged.
- It must clean up the stack after use; the stack must be in its original state when the fork process relinquishes control to any VAX/VMS routine.
- It must execute at IPLs between driver fork level and IPL$_POWER; it must not lower IPL below driver fork level except by creating a fork process at a lower IPL.
- When it returns control to the fork dispatcher, IPL must be the same as it was when the driver fork process was activated. The driver returns control to the fork dispatcher by invoking the wait for interrupt macro or the request complete macro.

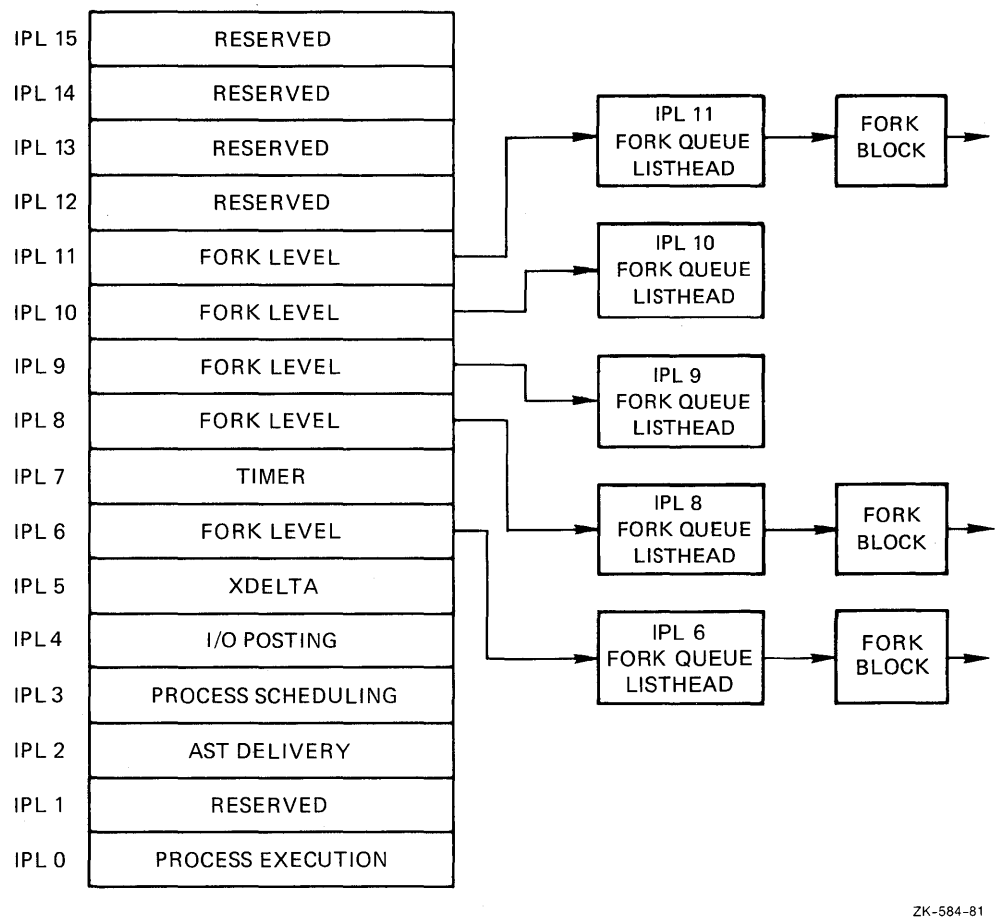


Figure 3-5: Fork Dispatching Data Structure

3.3 RESOURCE WAIT QUEUES

The processing of an I/O request often requires shared system resources such as memory and UNIBUS adapter map registers. The Queue I/O Request system service and driver fork processes call VAX/VMS routines to allocate and deallocate these resources. Since the resources are limited, I/O processing may be delayed until unavailable resources are released by other processes or drivers. Thus, synchronization of access to these resources can have a substantial impact on I/O request processing.

For example, the Queue I/O Request system service calls a VAX/VMS routine to allocate nonpaged system space for an I/O request packet. If the nonpaged pool is empty, the routine calls another VAX/VMS routine to save the process context and change the process state to resource wait mode (also called miscellaneous wait, or MWAIT). Process states and the resources for which processes can wait are described in the VAX/VMS Summary Description and Glossary. As a result of waiting, the process is a candidate to be swapped out of memory. When nonpaged pool becomes available, the scheduler reschedules the process.

During driver fork process execution at raised IPLs, driver context is very small. At any point, the driver can obtain all details about an I/O request by referring to the I/O data base. The driver needs only the address of the device unit control block which is the key to the rest of the data base. Therefore, VAX/VMS routines that control driver resources, such as UBA map registers, use driver fork blocks

and resource wait queues to save minimal driver context. Each entry in a queue consists of the following items:

- The address of the UCB, which is also the contents of R5 in the driver fork process; the UCB also contains the driver fork block
- R3, and normally R4, from the fork process
- A PC for the waiting fork process

When the awaited resource becomes available, the routine controlling the resource performs the following steps:

- Restores the UCB address to R5
- Restores the saved registers R3 and R4
- Grants the resource
- Transfers control to the saved driver return PC address

Because the VAX/VMS routine that controls a particular resource places the driver in a wait state when the driver requests an unavailable resource, drivers are unaware of being suspended and subsequently resumed. Drivers must not leave anything on the stack when calling a routine that may suspend the driver.

3.3.1 Competing for a Controller Data Channel

A controller data channel is a VAX/VMS synchronization mechanism that guarantees for multiunit controllers that one unit uses the controller at a time. A device driver fork process can read and write a device's registers whenever the device unit owns the controller data channel.

Devices that share a multiunit controller, such as disk units, own the controller data channel only when a VAX/VMS routine assigns the channel to the unit's driver fork process. In contrast, a single device unit on a controller always owns the controller data channel. Therefore, if VAX/VMS transfers control to such a driver's start I/O routine, the driver can immediately address the device registers without first obtaining the controller data channel.

An LP11 line printer device, such as the one discussed in Chapter 2, has a dedicated (single-unit) controller attached to the UNIBUS. When VAX/VMS finds the device idle and creates a line printer driver fork process to write data to the line printer data buffer, the controller data channel is guaranteed not to be busy. Because the controller data channel is not busy, the line printer start I/O routine can execute the following simple sequence of events:

- Retrieve the virtual address of the data to be written and the number of bytes to transfer from the device's unit control block
- Retrieve the virtual address of the device's control/status register from the interrupt dispatch block
- Calculate the address of the line printer's data buffer register by adding a constant offset to the control/status register address
- Write data one byte at a time to the line printer's data buffer until all bytes of data have been written

In contrast, a device unit on a multiunit controller must compete for the controller data channel with other devices attached to that controller.

An RK611 controller, for example, controls as many as eight RK06/RK07 devices. The disk driver fork process must gain control of the controller data channel before starting an I/O operation on the unit associated with the fork process. The disk driver's start I/O routine uses the following sequence to start a seek operation on an RK07 device:

- The start I/O routine requests the controller data channel by invoking a VAX/VMS channel arbitration routine.
- The VAX/VMS routine tests the CRB mask field to determine whether the controller data channel is available.
- If the channel is available, the VAX/VMS routine allocates the channel to the driver fork process and returns the address of the device control/status register to the fork process.

  If the channel is busy, the VAX/VMS routine saves the driver fork context in the UCB fork block and inserts the fork block address in the controller channel wait queue.
- When the driver fork process resumes execution, the process owns the controller channel. The fork process can then modify device registers to activate the device.
- The driver's start I/O routine then requests VAX/VMS to suspend driver processing in anticipation of an interrupt or timeout and to release the channel.
- The VAX/VMS channel releasing routine assigns channel ownership to the next driver fork process in the channel wait queue, loads the control/status register address into a general register, and reactivates the suspended driver fork process.
- The reactivated fork process continues execution as though the channel had been available in the first place.

The VAX/VMS channel arbitration routines keep track of controller availability using a flag field in the channel request block. The driver fork process must always request and release the controller data channel by invoking these routines. Once the driver owns a controller data channel, the driver is free to read and modify device registers.

CHAPTER 4

THE UNIBUS ADAPTER

The UNIBUS adapter connects the UNIBUS, an asynchronous bidirectional bus, to the backplane interconnect. The adapter performs the following functions:

- Arbitrates priority interrupts from UNIBUS devices
- Delivers interrupts from UNIBUS devices to the processor
- Allows drivers to gain access to UNIBUS device registers using system virtual addresses
- Translates 18-bit UNIBUS addresses to physical addresses
- Provides a data transfer path to randomly ordered physical addresses, that is, to discontiguous pages
- Provides buffered data transfer paths to consecutively increasing physical addresses
- Permits byte-aligned buffers for UNIBUS devices requiring word-aligned buffer addresses

Together the UNIBUS adapter and the backplane interconnect permit devices and device drivers to exchange data without much awareness of the intervening hardware. Because VAX/VMS routines handle the details of the adapter/backplane interconnect interface, most device drivers do not need to know the interface protocol.

The critical responsibility of UNIBUS device drivers that actively compete for shared UNIBUS adapter resources is that they all execute at the same fork IPL. This IPL convention synchronizes access to the UNIBUS adapter data structures.

In general, device drivers use the UNIBUS adapter for the following purposes:

- Reading and writing device registers
- Mapping UNIBUS addresses to physical addresses and vice versa for direct memory access (DMA) transfers
- Buffering data transfers

Drivers for UNIBUS devices that do not perform DMA transfers are unaware of the presence of the UNIBUS adapter. The UNIBUS adapter provides access to device registers using an address mapping scheme that is invisible to the driver. However, drivers that handle DMA transfers to and from UNIBUS devices must call VAX/VMS routines that establish the appropriate mapping.

4.1 READING AND WRITING DEVICE REGISTERS

Each I/O controller or device directly attached to the UNIBUS has a set of control/status and data registers. These registers are assigned addresses in a portion of the physical address space called the UNIBUS address space. Device drivers obtain device status and activate devices by reading and writing to these registers.

Generally, a device driver can treat the addresses of device registers as identical to all other virtual addresses. The driver can read and write data to the device register as though the device register were a location in memory. The driver must obey the restrictions on instructions described in Section 6.2. The UNIBUS adapter performs the actual mapping of virtual address to UNIBUS addresses that correspond to device registers.

Before a driver for a multiunit controller can gain access to device registers, it must first obtain a controller channel, as described in Section 3.3.1.

4.2 MAPPING UNIBUS AND PHYSICAL ADDRESSES FOR DMA TRANSFERS

The UNIBUS address space consists of 256K bytes of memory, of which 8K bytes are reserved for device control registers. UNIBUS DMA devices read and write data from and to memory locations using 18-bit UNIBUS addresses. The UNIBUS adapter translates the 18-bit UNIBUS addresses into physical addresses. This translation allows the operating system, I/O drivers, and UNIBUS devices to access the same physical address space.

The UNIBUS adapter provides 496 map registers to translate UNIBUS addresses to physical addresses. Each map register represents one page of the UNIBUS address space. A field in the map register identifies the page frame number corresponding to the UNIBUS address that the map register represents.

For example, VAX/VMS routines fill as many map registers with valid page frame addresses as needed for a DMA transfer. A DMA UNIBUS device puts an address on the UNIBUS. The UNIBUS adapter receives the address and translates it using the following information:

- The 9-bit UNIBUS page address field (bits 9 through 17 of the UNIBUS address) identifies the UBA map register.
- The page frame number field in the map register specifies the high order bits of the physical address.
- UNIBUS address bits 2 through 8 map directly to bits 0 through 6 of the physical address.

The resulting physical address locates the longword that is the target of the transfer. The UNIBUS adapter identifies the byte addressed within the longword by interpreting the low-order two bits of the UNIBUS address.

Figure 4-1 illustrates the UNIBUS to physical address mapping.

Each UNIBUS adapter map register also contains a bit called the map register valid bit. The UNIBUS adapter tests this bit every time the map register is used. If the bit is not set, the UNIBUS adapter aborts the UNIBUS transfer. The valid bit is zero whenever the register is not mapped to a physical address.

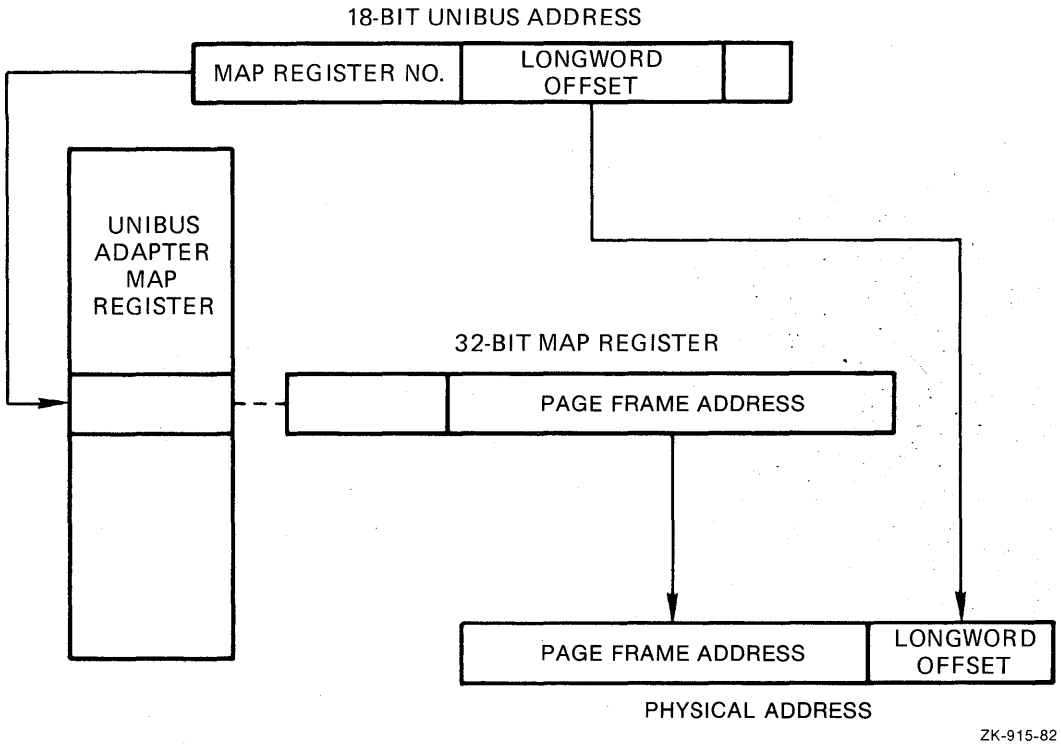


Figure 4-1: UNIBUS to Physical Address Mapping

4.2.1 UNIBUS Adapter Data Transfer Paths

The UNIBUS adapter sends data through one of several data paths for UNIBUS devices performing DMA transfers. One data path, the direct data path (DDP), allows UNIBUS transfers to randomly ordered physical addresses. The direct data path maps each UNIBUS transfer to a backplane interconnect transfer. Thus, a single word or byte of data is transferred per backplane interconnect operation.

The remaining data paths, the buffered data paths (BDPs), allow devices on the UNIBUS to transfer much faster than through the direct data path. The buffered data paths store UNIBUS data so that multiple UNIBUS transfers result in a single backplane interconnect transfer.

The UNIBUS adapter hardware of certain processors restricts normal buffered data paths to referencing only consecutively increasing addresses. Through a special mode of operation, these UNIBUS adapters can also reference data in a randomly ordered, longword-aligned manner. Other processors do not impose this restriction. In order for a device driver to run on both types of processors, it must observe two rules:

- Normal buffered data paths must always transfer data to consecutively increasing addresses.
- To reference random longword aligned data, the longword enable bit (LWAE) must be set.

When a UNIBUS device begins a DMA transfer by placing an address on the UNIBUS, the UNIBUS adapter map register not only performs address mapping but also provides the number of the data path to be used for the transfer. Each UNIBUS adapter map register contains a field that describes the data path. Data path 0 is the direct data path; the other data paths are the buffered data paths.

The sequence below describes a UNIBUS device DMA transfer.

- The UNIBUS device puts an address on the UNIBUS.
- The UNIBUS adapter locates the UNIBUS adapter map register that corresponds to the UNIBUS address.
- The UNIBUS adapter verifies that the map register has the map register valid bit set.
- The UNIBUS adapter maps the UNIBUS address to a page frame number.
- The UNIBUS adapter extracts the number of the data path to be used for the transfer from the map register.
- The data path translates the UNIBUS function to a backplane interconnect function by reading the UNIBUS control lines.
- Based on the UNIBUS function indicated by the UNIBUS control lines, (DATI, DATIP, DATO, or DATOB), the UNIBUS adapter starts appropriate UNIBUS and backplane interconnect operations to transfer data to or from the UNIBUS device.

4.2.1.1 Direct Data Path - Since the direct data path performs a backplane interconnect transfer for every UNIBUS transfer, the data path can be used by more than one UNIBUS device at a time. The UNIBUS adapter arbitrates among devices that wish to use the direct data path simultaneously. The device driver is unaffected by this UNIBUS adapter arbitration.

The direct data path is slower than buffered data paths because each UNIBUS transfer cycle corresponds to a backplane interconnect cycle. One word or byte is transferred per backplane interconnect cycle. On some hardware configurations, the direct data path is unable to transfer a word of data to an odd physical address. Therefore, an FDT routine for a DMA device that uses the direct data path should check that the specified buffer is on a word boundary.

UNIBUS devices that transfer data through the direct data path do so in order to perform the following functions:

- Execute an interlock sequence to the backplane interconnect (DATIP-DATO/DATOB)
- Transfer to randomly ordered addresses instead of consecutively increasing addresses
- Mix read and write functions

The direct data path is the simplest data path to program. Since the direct data path can be shared simultaneously by any number of I/O transfers, the device driver need not allocate that data path. Once the map registers are loaded, the device driver initiates the transfer by setting appropriate device control register bits. The programming sequence is as follows:

- Allocate a set of map registers.
- Load the map registers with physical address mapping data and the data path number (0 for the direct data path).

- Set the valid bit in every map register. The map register that follows the last map register must have the valid bit cleared.
- Load the starting address of the transfer in a device register.
- Load the transfer byte or word count in a device register.
- Set bits in the device control register to initiate the transfer.

The operating system performs the first three steps above. The driver fork process simply calls VAX/VMS routines to allocate and load the map registers.

4.2.1.2 Buffered Data Paths - In contrast to the direct data path, the buffered data paths transfer data much more efficiently between the UNIBUS and the backplane interconnect by decoupling the UNIBUS transfer from the backplane interconnect transfer. Buffered data paths read or write multiple words of data in a transfer, and buffer the unrequested portions of the data in UNIBUS adapter buffers. Thus, several UNIBUS read functions can be accommodated with a single backplane interconnect transfer.

Advantages that buffered data paths offer to UNIBUS devices include the following:

- Fast DMA block transfers to or from consecutively increasing addresses
- Word-oriented block transfers that begin and end on an odd byte of memory; note, however, that these transfers can be quite slow since the UNIBUS adapter may need to perform multiple transfers to complete a 1-word transfer
- 32-bit data transfers from random longword-aligned physical addresses

A buffered data path cannot be assigned to more than one active transfer at a time. When a driver fork process is preparing to transfer data to or from a UNIBUS device on a buffered data path, the driver requests allocation of a free buffered data path and a set of UNIBUS adapter map registers. A VAX/VMS I/O routine writes the number of the data path into each of the assigned map registers.

A UNIBUS device transfer over a buffered data path has the following restrictions:

- All addresses in a block transfer must be consecutively increasing addresses.
- All transfers within a block must be of the same function type (DATI or DATO/DATOB).

A buffered data path stores data from the UNIBUS in a buffer until multiple words of data have been transferred (except in longword-aligned transfer mode; see below). Then, the UNIBUS adapter transfers the contents of the buffer to the appropriate physical address in a single backplane interconnect operation. The procedure for a UNIBUS write operation that transfers data to memory is broken into individual steps as follows:

- The UNIBUS device transfers one word of data to the buffered data path.
- The buffered data path stores the word of data and completes the UNIBUS cycle.
- The buffered data path sets its buffer-not-empty flag to indicate that the buffer contains valid data.
- The UNIBUS device repeats the first three steps until the buffer is full.
- When the UNIBUS device addresses the last byte or word in the buffer, the UNIBUS adapter recognizes a complete data-gathering cycle.
- The buffered data path requests a backplane interconnect write function to write the data from the buffered data path to memory.
- When the backplane interconnect transfer is complete, the buffered data path clears its flag to indicate that the buffer no longer contains valid data.

The procedure for a UNIBUS read function varies according to the type of UNIBUS adapter. Some adapters can perform a prefetch function, while others cannot. Device drivers that adhere to the conventions outlined in this manual will execute properly on either type of UNIBUS adapter with no difference except that of system throughput.

The following paragraphs discuss the UNIBUS read operation with and without the prefetch function.

The prefetch automatically fills the buffer after the contents of a buffered data path are transferred to the UNIBUS. The prefetch speeds up UNIBUS reads from memory. The steps of a UNIBUS read function are listed below.

- The UNIBUS device initiates a read operation from a buffered data path.
- The buffered data path checks to see if its buffers contain valid data.
- If the buffers do not contain valid data, the buffered data path initiates a read function to fill the buffers with data. The transfer completes before the UNIBUS adapter begins a UNIBUS transfer.
- The buffered data path transfers the requested bytes to the UNIBUS. Bytes of data that were not transferred to the UNIBUS remain in the buffer.
- The buffered data path sets its buffer-not-empty flag to indicate that the buffers contain valid data.
- When the UNIBUS device empties the buffers of the buffered data path with a UNIBUS read function that accesses the last word of data, the buffered data path clears the not empty flag to indicate that the buffers no longer contain valid data.
- The buffered data path then initiates a read function to prefetch data from memory.

- When the transfer is complete, the buffered data path sets the buffer-not-empty flag to indicate that the buffers now contain valid data.

The prefetch may attempt to read data beyond the address mapped by the final map register. To avoid a read to memory that does not exist, the VAX/VMS map register allocate and load routines always allocate one extra map register and clear the valid bit before initiating the transfer. When the UNIBUS adapter notices that the map register for the prefetch is invalid, the UNIBUS adapter simply aborts the prefetch without reporting an error.

The steps of a UNIBUS read function without prefetch are listed below.

- The UNIBUS device initiates a read operation from a buffered data path.
- The buffered data path checks to see if its buffers contain valid data.
- If the buffers do not contain valid data, the buffered data path initiates a read function to fill the buffers with data. The transfer completes before the UNIBUS adapter begins a UNIBUS transfer.
- The buffered data path transfers the requested bytes to the UNIBUS. Bytes of data that were not transferred to the UNIBUS remain in the buffer.

4.2.1.3 Byte Offset Data Transfers - Some UNIBUS devices are restricted to transferring integral words of data in word-aligned UNIBUS addresses. The buffered data paths allow these devices to perform transfers to memory that begins and ends on an odd-byte address. A byte-offset bit in the map registers indicates byte-aligned data to the hardware. If the bit is set, the hardware increments physical addresses. A VAX/VMS subroutine that loads map registers determines whether the data is word- or byte-aligned and sets the byte offset bit accordingly.

4.2.1.4 Purging a Buffered Data Path - Since prefetches may read more data from memory than the UNIBUS device wishes to read, driver fork processes must ask the UNIBUS adapter to purge the buffered data path when a transfer is complete. In addition, a transfer from a device to the backplane interconnect can complete with some data left in the buffer. The driver must purge the data path to complete the transfer.

The purge guarantees that the data is not transferred to the next user of the buffered data path. The driver fork process performs the purge by calling a standard VAX/VMS subroutine that:

- Tells the hardware to purge the buffered data path register owned by the fork process. For a UNIBUS read function, the adapter simply clears the buffer-not-empty flag. For a UNIBUS write function, the adapter transfers any data left in the data path buffer to VAX-11 memory, then clears the flag.
- Notifies the driver fork process of any error that occurs during the purge.

The data path must be purged before the driver releases map registers or the buffered data path register.

4.2.1.5 Longword-Aligned 32-Bit Random Access Mode - Another method of transferring data over a buffered data path is in longword-aligned 32-bit random access mode. This mode permits a device that reads data from or writes data to memory in longword-aligned and longword multiples to use the buffered data path for random memory access.

To ensure that random access mode works correctly regardless of processor type, a buffered data path should not repeatedly address the same longword. For example, on certain processors a UNIBUS device that polls a single longword, waiting for data, will constantly be returned the same data.

A longword-aligned transfer over a buffered data path is faster than a direct data path transfer and somewhat slower than a normal buffered data path transfer.

To transfer data in the longword-aligned 32-bit random access mode, the driver fork process sets the longword-access-enable bit (VEC$V_LWAE) in the channel request block (CRB) prior to loading the map registers. The UNIBUS device can then perform a read (DATI) or write (DATO) function.

For a UNIBUS read, the function occurs as follows:

- The driver fork process initiates a read function on the UNIBUS device.
- The UNIBUS adapter clears the buffer-not-empty flag in the assigned buffered data path.
- The UNIBUS adapter issues a read from memory operation.
- The UNIBUS adapter stores the longword of data in the buffered data path and sets the buffer-not-empty flag.
- The UNIBUS adapter initiates two UNIBUS read operations to transfer two words of data.

For a UNIBUS write, the function occurs as follows:

- The driver fork process initiates a write function on the UNIBUS device.
- The UNIBUS adapter clears the buffer-not-empty flag in the assigned buffered data path.
- The UNIBUS adapter issues two write operations to transfer two words of data from the UNIBUS device.
- The UNIBUS adapter stores the longword of data in the buffered data path and sets the buffer-not-empty flag.
- The UNIBUS adapter initiates a backplane interconnect write operation.
- When the backplane interconnect write operation is complete, the UNIBUS adapter clears the buffer-not-empty flag.

4.3 THE VAX-11/780 UNIBUS ADAPTER

The UNIBUS adapter on a VAX-11/780 processor has the following hardware features:

- One direct data path that does not handle byte offsets.
- Fifteen buffered data paths that handle byte offsets. Each data path has an eight-byte buffer and supports the prefetch function and longword random access mode. The UNIBUS adapter uses extended SBI read or write operations to fill a buffered data path.
- The Synchronous Backplane Interconnect (SBI). The SBI uses a 30-bit physical address.
- 496 map registers.
- Nondirect vector interrupt dispatching.
- Longword aligned random access mode. When a data path is set to this mode, data prefetch is disabled and only four bytes of data are buffered.

Figure 4-2 shows the fields within the map register and data path register for the VAX-11/780 UNIBUS adapter.

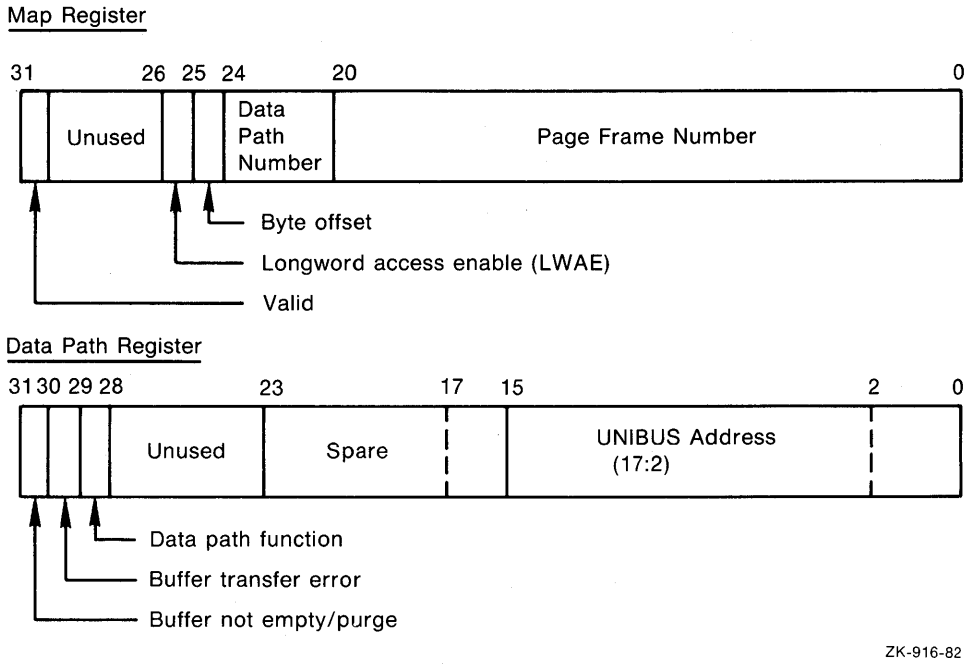


Figure 4-2: VAX-11/780 UNIBUS Adapter Registers

4.4 THE VAX-11/750 UNIBUS ADAPTER

The UNIBUS adapter on a VAX-11/750 processor has the following hardware features:

- One direct data path that handles byte offsets.
- Three buffered data paths that handle byte offsets. Each data path has a four-byte buffer. The buffered data paths do not perform the prefetch function.
- The backplane interconnect. This interconnect uses 24-bit physical addresses.

- 512 map registers. The VAX/VMS system uses only 496 of these registers.
- Direct vector interrupt dispatching.
- Implied longword aligned random access mode. Buffered data paths on the VAX-11/750 only buffer four bytes of data. Since the data paths do not perform a prefetch, they can always reference longwords at random. However, because of a hardware restriction, VAX-11/750 buffered data paths do not allow repeated references to a longword. If a longword is referenced more than once, bad data may be returned. To ensure compatibility between processors, device drivers can set the LWAE bit to indicate longword mode.

Figure 4-3 shows the fields within the map register and the data path register for the VAX-11/750 UNIBUS adapter.

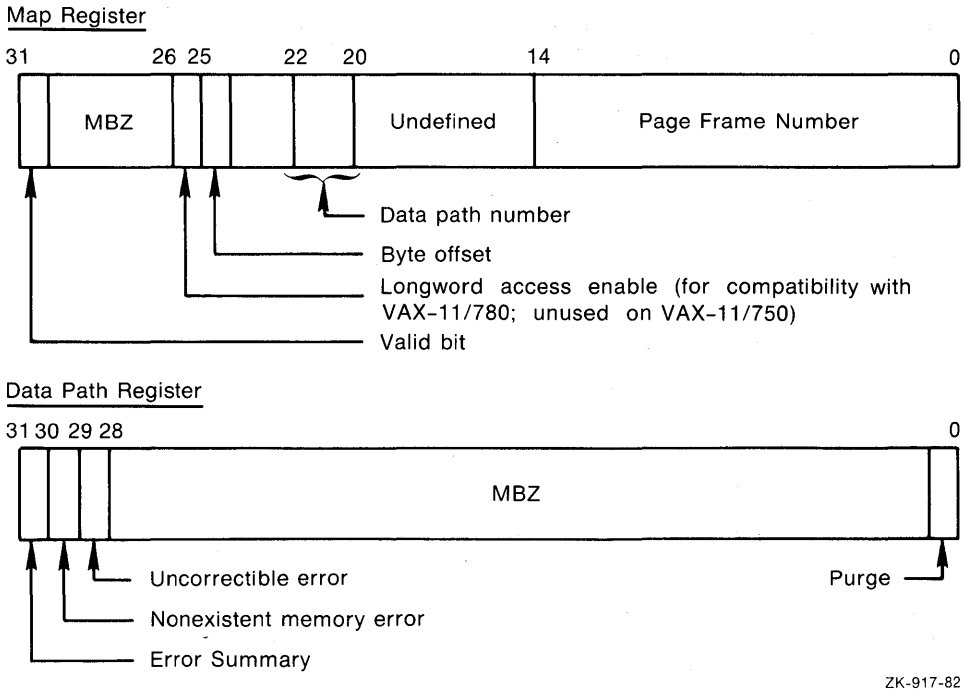


Figure 4-3: VAX-11/750 UNIBUS Adapter Registers

4.5 THE VAX-11/730 UNIBUS ADAPTER

The UNIBUS adapter on a VAX-11/730 processor has the following hardware features:

- One direct data path that handles byte offsets.
- No buffered data paths.
- The backplane interconnect. This interconnect uses 24-bit physical addresses.
- 512 map registers. The VAX/VMS system uses 496 of these registers.
- Direct vector interrupt dispatching.

Figure 4-4 shows the fields within the map register for the VAX-11/730 UNIBUS adapter. This adapter does not use a data path register; it exists for compatibility with the other VAX-11 processors and contains only zeroes. The adapter ignores any data writen to this longword.

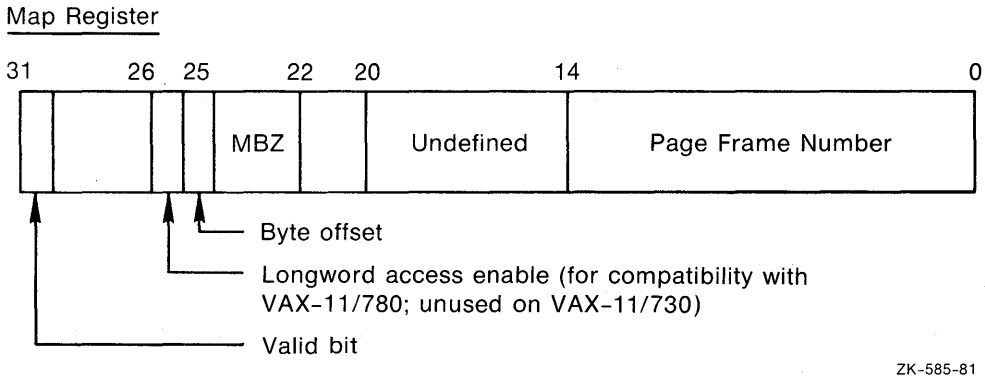


Figure 4-4: VAX-11/730 UNIBUS Adapter Map Register

CHAPTER 5

OVERVIEW OF I/O PROCESSING

Under the VAX/VMS operating system, I/O processing occurs in three major phases:

- I/O request preprocessing
- Device activation and subsequent handling of the device interrupt
- I/O postprocessing

When a user process issues an I/O request, the Queue I/O Request system service gains control. The system service coordinates the preprocessing of the I/O request. The last driver FDT routine called by the Queue I/O Request system service calls a VAX/VMS routine that creates a driver fork process to execute the driver's start I/O routine; this is the routine that activates the device. When the transfer completes, the device requests an interrupt that results in execution of the driver's interrupt service routine. This routine handles the interrupt and requests creation of a driver fork process to perform device-dependent I/O postprocessing. The driver fork process then transfers control to the system to perform device-independent I/O postprocessing. Figure 5-1 illustrates the sequence of events.

5.1 PREPROCESSING AN I/O REQUEST

The Queue I/O Request system service performs device-independent preprocessing of an I/O request and calls driver FDT routines to perform device-dependent preprocessing. To preprocess an I/O request, the Queue I/O Request system service takes the following steps:

- Verifies that the requesting process has assigned a process I/O channel to the target device
- Locates the device driver in the I/O data base
- Validates the I/O function code
- Checks process I/O request quotas
- Validates the I/O status block
- Allocates and sets up the I/O request packet
- Calls driver FDT routines to perform device-dependent preprocessing

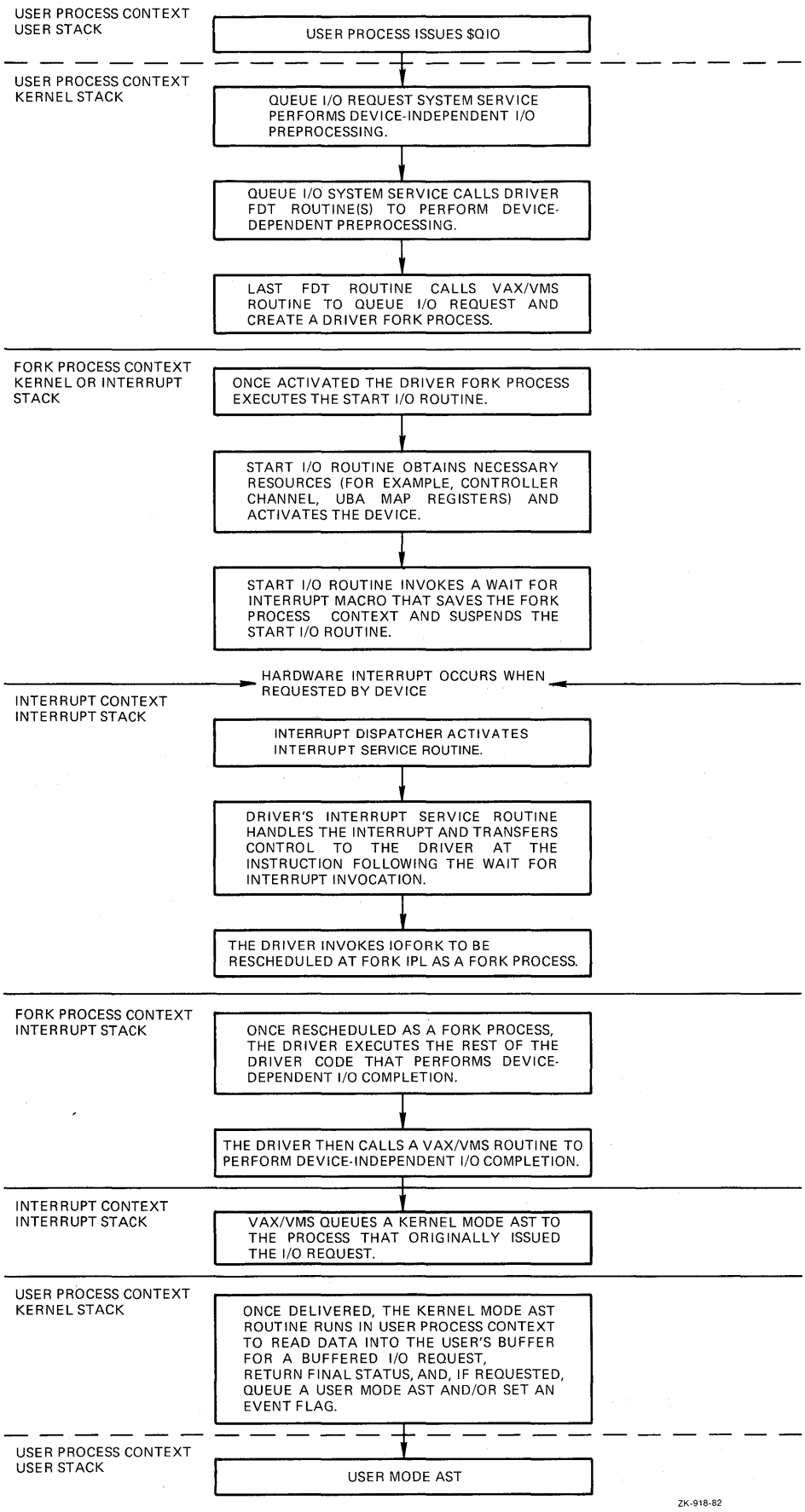


Figure 5-1: Sequence of Driver Execution

5.1.1 Process I/O Channel Assignment

The first step in preprocessing an I/O request is to verify that the I/O request specifies a valid process I/O channel. The process I/O channel is an entry in a system-maintained process table that describes a path of reference from a process to a peripheral device unit. Before a program requests I/O to a device, the program identifies the target device unit by issuing an Assign I/O Channel system service call. The Assign I/O Channel system service performs the following functions:

- Locates an unused entry in the table of process I/O channels
- Creates a pointer to the device unit in the table entry for the channel
- Returns a channel index number to the program

When the program issues an I/O request, the Queue I/O Request system service verifies that the channel number specified is associated with a device and locates the portion of the I/O data base that describes the device. Figure 5-2 illustrates the path from a process channel number to the device's unit control block.

5.1.2 Locating a Device Driver in the I/O Data Base

Using information in the unit control block, a driver can find other I/O data structures associated with the device, including the following:

- Channel request block[1]
- Interrupt dispatch block
- Device data block

5.1.2.1 Unit Control Block (UCB) - The process channel number indirectly points to the unit control block for the target device. The unit control block contains the first in a chain of pointers into the I/O data base. The pointer chain leads to the addresses of driver tables and routines in the driver that handles the target device.

A unit control block describing a device unit exists for each device in the system. The unit control block indicates the current state of the device unit by specifying such information as the following:

- Whether the device is active
- What I/O request is being processed
- Where transfer buffers are located

---

1. Channel request blocks (CRBs) and channel control blocks are two completely separate data structures. It is sometimes helpful to think of the channel request block as the "controller" request block because it describes the hardware controller. The channel control block, on the other hand, describes a logical path from a process to an associated unit control block.

Since drivers run as fork processes and cannot use process address space to store additional context, drivers use the unit control block for temporary data storage during I/O processing. Chapter 7 describes how you can allocate additional UCB space for storing data or device-dependent driver context.

The unit control block also holds the context of a driver fork process when VAX/VMS I/O routines suspend the fork process to wait for an asynchronous event such as a device interrupt.

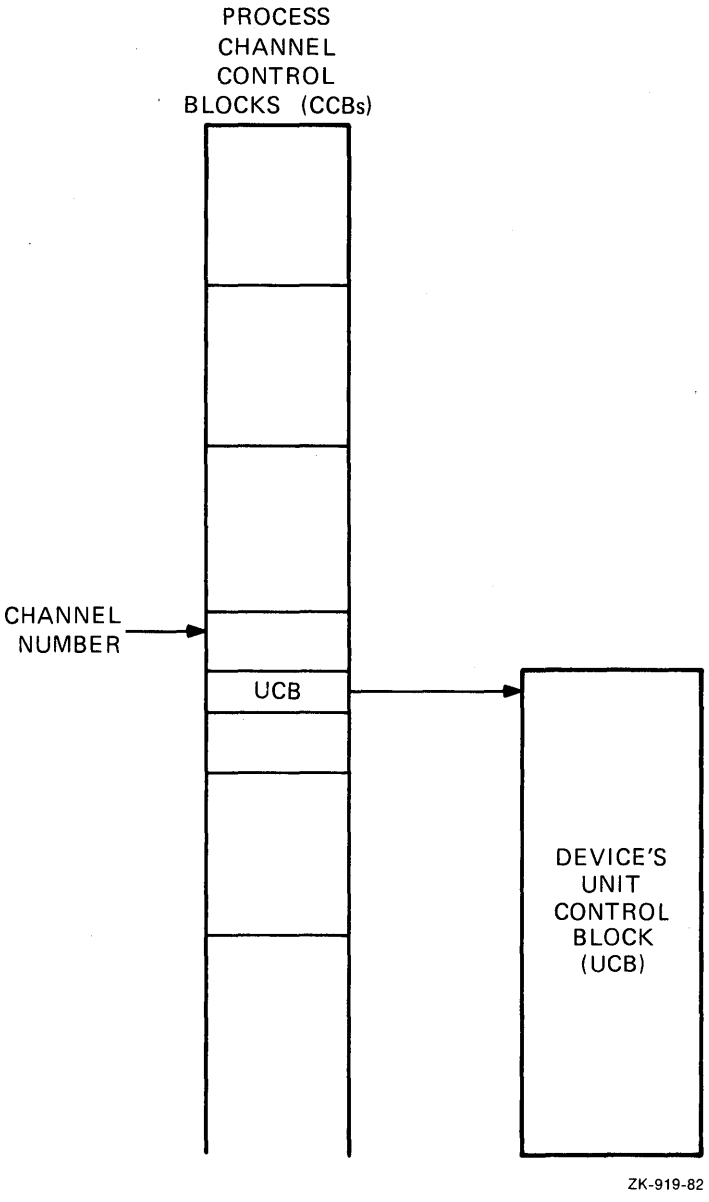


Figure 5-2: Locating the Target Device

5.1.2.2 Channel Request Block (CRB) - All unit control blocks describing device units attached to a particular controller contain a pointer to a single channel request block. The channel request block contains the following information:

- Code that transfers control to a driver interrupt service routine
- Addresses of driver's unit and controller initialization routines
- A pointer to the interrupt dispatch block, which further describes the controller

Controllers can be either multiunit or dedicated. A dedicated controller has only one device unit. The VAX/VMS operating system does not use the channel request block to synchronize I/O operations for a dedicated controller. The channel request block still is present and used by drivers and operating system routines.

For multiunit controllers, a VAX/VMS routine uses a field in the channel request block to arbitrate pending driver requests for the controller. When the system grants ownership of a multiunit controller data channel to a driver fork process, the fork process can initiate an I/O operation on a device attached to that controller.

The unit control blocks for devices attached to a multiunit controller all contain pointers to the same channel request block; this allows the operating system to manage the controller data channel. Figure 5-3 illustrates the data structures required to describe three devices on a multiunit controller.

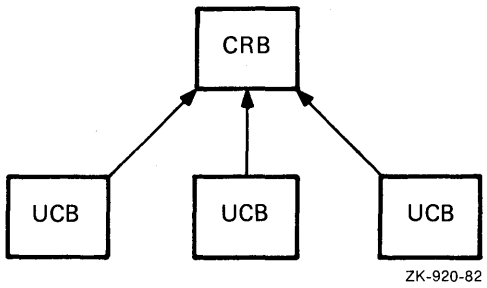


Figure 5-3: Data Structures for Three Devices on One Controller

5.1.2.3 Interrupt Dispatch Block (IDB) - The channel request block also points to an interrupt dispatch block. The interrupt data base contains three critical data structure addresses:

- The address of the UCB of the device unit, if any, that currently owns the controller data channel
- The address of the control/status register (CSR); it is the key to access to device registers
- The address of the adapter control block (ADP) that describes the UNIBUS adapter to which the controller is attached

5.1.2.4 Device Data Block (DDB) - All unit control blocks describing device units attached to a single controller contain a pointer to a single device data block (DDB). The device data block contains the following fields that identify the device and its driver:

- The generic device/controller name
- The name of the device's driver as obtained from the driver prologue table; see Chapters 7 and 14 for the use of the driver name

Figure 5-4 illustrates the relationship between the I/O data structures that describe a group of equivalent devices on two separate controllers.

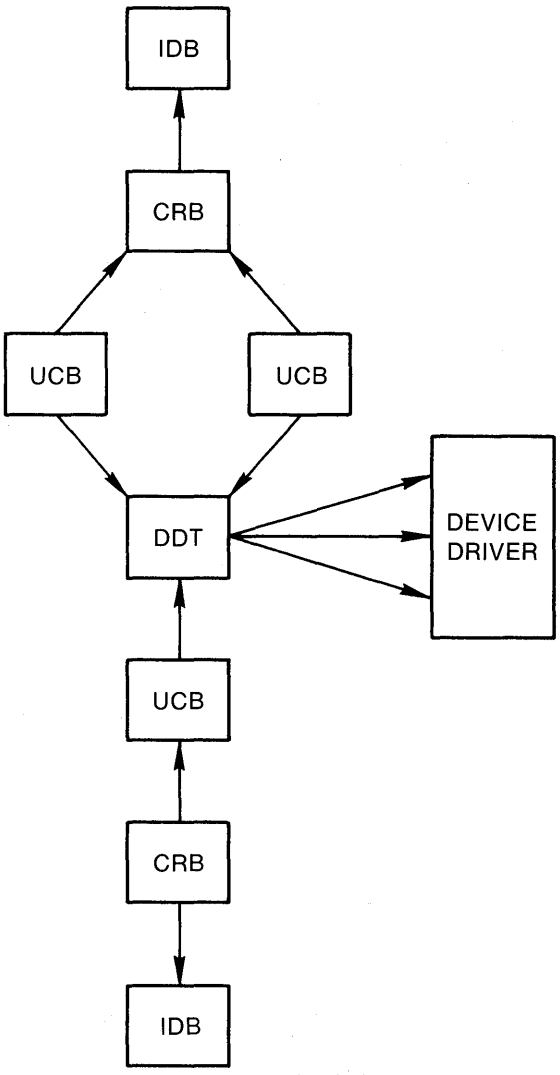


Figure 5-4: I/O Data Base for Two Controllers

In Figure 5-4, one controller has a single device unit, and the other controller has two device units. Devices on both controllers share the same driver code.

5.1.3 Validating the I/O Function

Using the I/O data structures described above, the Queue I/O Request system service locates the address of the driver's function decision

table by following a chain of pointers beginning in the UCB of the target device for the I/O request, as follows:

UCB ⟶ DDT ⟶ FDT

The system service then uses data in the function decision table to analyze the I/O function. The service confirms that the function specified in the I/O request is a valid function for the device.

5.1.4 Checking Process I/O Request Quotas

The Queue I/O Request system service determines whether the I/O request being readied will cause the process to exceed its quota for outstanding direct or buffered I/O requests. If the process remains under quota, the system service allows it to continue I/O preprocessing.

In the case where quota is exceeded, the Queue I/O request system service examines its resource wait flag. If the flag is clear, the system service aborts the I/O request.

If the flag is set, the process is placed in a wait state until it drops below quota, at which time the $QIO system service modifies the process quotas as appropriate for the requested operation.

5.1.5 Validating the I/O Status Block

If the I/O request specifies a quadword I/O status block to receive final I/O status information, the Queue I/O Request system service determines whether the process issuing the request has write access to the status block locations specified. If the process has write access, the system service fills the quadword with zeros. If the process does not have write access, the system service terminates the request with an error status.

5.1.6 Allocating and Setting Up an I/O Request Packet

If validation of the I/O request succeeds to this point, the Queue I/O Request system service allocates a block of nonpaged system memory to contain an I/O request packet.

Before the system service allocates an I/O request packet, it raises the hardware IPL of the processor to IPL$_ASTDEL to block any other asynchronous activity in the process. The new IPL prevents possible termination of the process; process termination would result in the operating system's losing track of the system memory allocated for the I/O request packet.

The Queue I/O Request system service attempts to allocate an I/O request packet from a linked list of preallocated I/O request packets. If no preallocated packets exist, the service calls a VAX/VMS routine that allocates an I/O request packet from nonpaged pool. This allocating routine synchronizes with the rest of the system so that it can allocate the memory needed.

The Queue I/O Request system service continues I/O preprocessing by writing the following description of the I/O request into the packet:

- Size in bytes of the I/O request packet
- A type field identifying the block as an I/O request packet
- Access mode of the process at the time of the I/O request
- Process identification of the requesting process
- If specified in the I/O request, the address of an AST routine and its parameter
- If the device is file-structured, the address of a control block that describes the physical location of part of the file (window control block)
- Address of the target device's unit control block
- I/O function code; read/write virtual block functions are reduced to their logical equivalents before storing a code value
- Number of event flag to set when I/O processing is complete for the I/O request
- Base software priority of the requesting process
- If specified in the I/O request, the address of an I/O status block
- Process I/O channel index number
- A flag indicating whether the I/O function is buffered or direct I/O
- A flag indicating whether the I/O request is an input request
- A flag indicating whether the process has privilege to perform logical or physical I/O functions
- A flag indicating whether the I/O function is a physical I/O function
- If specified in the I/O request, the address and size of a diagnostic buffer and a flag indicating that the buffer is present
- If an AST routine is specified in the I/O request, a flag indicating that the process quota for the use of ASTs has been modified

The Queue I/O Request system service writes the above fields in the I/O request packet because these fields contain device-independent data. Driver routines or VAX/VMS common FDT routines must fill in the device-dependent portions of the I/O request packet.

Appendix A illustrates the format of an I/O request packet.

5.1.7 Function Decision Table Processing

The driver function decision table controls the device-dependent preprocessing of an I/O request. Figure 5-5 illustrates the format of a function decision table.

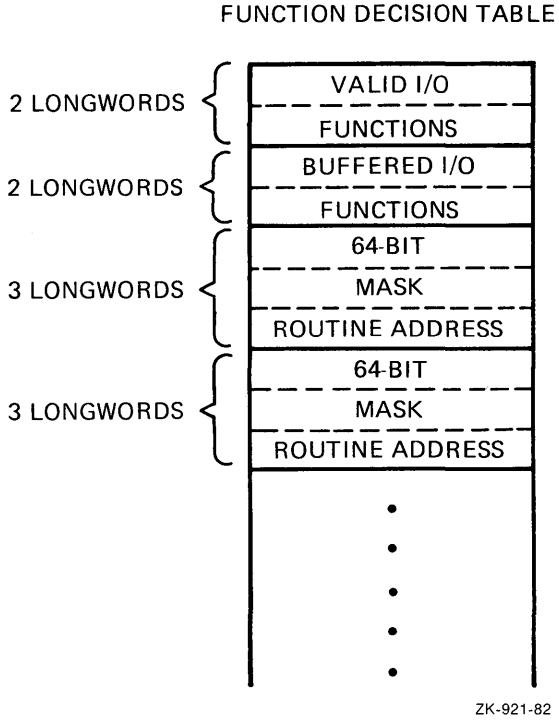


Figure 5-5: Driver Function Decision Table

The I/O function code specified in an I/O request is a 16-bit value consisting of two fields:

- A 6-bit I/O function code (bits 0 through 5)
- A 10-bit I/O function modifier (bits 6 through 15)

The 6-bit function code field permits you to define 64 unique I/O function codes for every device type. Chapter 7 describes how you can define these function codes.

Because each driver can define up to 64 unique I/O function codes, the first two entries of a function decision table are two longwords each; that is, 64 bits each. The first entry is a bit mask of all valid I/O function codes for the device. Each bit represents a unique function code. The second entry is a bit mask of those valid codes that are also buffered I/O functions. The Queue I/O Request system service uses these two bit masks to determine whether the I/O function code is valid and whether the operation is to be buffered or direct I/O.

The remaining entries of a function decision table are three longwords each. The first two longwords form a bit mask of I/O function codes. The third longword is the address of an I/O preprocessing routine to be called for the I/O function codes whose corresponding bits are set in the first two longwords.

The Queue I/O Request system service uses the value of the low-order six bits of the I/O function code to determine which bit to check in each FDT bit mask. That is, if a function code has a value of 22, the system service checks the 23rd bit (bit 22) of each bit mask.

Some of the preprocessing routines are present in the operating system because they provide device-independent services. Chapter 8 describes these routines. Other routines are in the driver because they perform device-dependent services.

The Queue I/O Request system service uses the 3-longword entries in the function decision table to call I/O preprocessing routines in the driver or system, as follows:

- If the bit in the FDT entry corresponding to the value of the function code is set, the system service calls the associated preprocessing routine; that is, the routine whose address is in the longword following the bit mask.
- If the bit corresponding to the I/O function code value is not set, the Queue I/O Request system service advances to the next FDT entry bit mask and repeats the step above.
- When the preprocessing routine completes its activity, the routine either returns control to the system service or transfers control to a VAX/VMS routine that queues the I/O request packet or completes the request.
- If the Queue I/O Request system service regains control, the routine advances to the next FDT entry and repeats the first step above.
- If all preprocessing for the I/O function is complete, the preprocessing routine does not return to the Queue I/O Request system service. Instead, the routine transfers control to either a VAX/VMS routine that queues the I/O request for the driver's start I/O routine or a VAX/VMS routine to complete or abort the request.

Figure 5-6 illustrates the use of FDT routines in I/O preprocessing.

As illustrated in Figure 5-6, FDT routines are responsible for ending the Queue I/O Request system service's scan of the function decision table. For every valid I/O function code for a device, one FDT entry must cause I/O preprocessing for the function to end.

FDT routines execute in the full process context of the process that requested the I/O operation. Thus, FDT routines can gain access to process virtual address space. Once all FDT preprocessing is complete, however, the rest of the processing for the I/O request continues in the limited context of a driver fork process or an interrupt service routine.

5.2 HANDLING DEVICE ACTIVITY

When I/O preprocessing is complete, but the I/O operation is not yet complete, an FDT routine transfers control to a VAX/VMS I/O packet queuing routine that arbitrates device activity. The arbitration routine ensures that it creates only one driver fork process at a time for each device unit on the system. One fork process handles one I/O request packet.

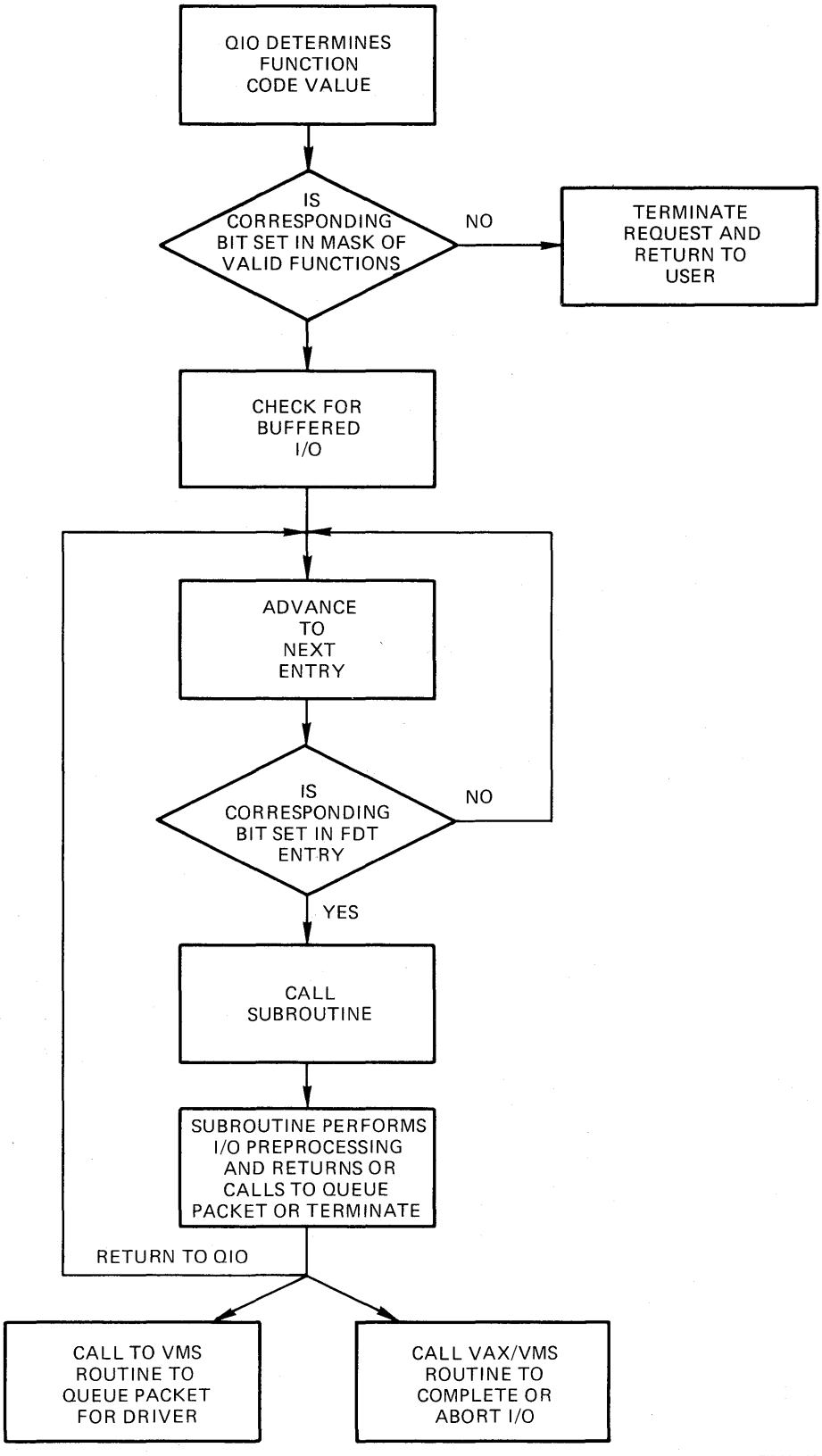


Figure 5-6: FDT Routines and I/O Preprocessing

5.2.1 Creating a Driver Fork Process to Start I/O

The I/O packet queuing routine determines whether a driver fork process exists for the target device, as follows:

- If the device is idle, no driver fork process exists for the device; in this case, the queuing routine immediately creates a driver fork process to execute the start I/O routine and transfers control to it.
- If the device is busy, a driver fork process already exists for the device; in this case, the queuing routine inserts the I/O request packet into a queue of I/O request packets waiting for the device unit. The routine queues the packet according to the base priority of the caller. Within each priority, packets are in first-in/first-out order.

In the latter case, by the time the driver's start I/O routine gains control to dequeue the I/O packet, the originating user's process context is no longer available. The driver must execute in the reduced context of a driver fork process. Because the context of the process initiating the I/O request is not guaranteed to a driver's start I/O routine, the VAX/VMS I/O packet queuing routine always initiates the driver's start I/O routine with a context that is appropriate for a fork process. The driver fork process consists of three registers (or fewer) and a PC. The I/O packet queuing routine establishes this context in the following steps:

- It raises IPL to driver fork IPL.
- It loads the address of the I/O request packet into R3.
- It loads the address of the device's unit control block into R5.
- It transfers control to the driver's start I/O routine entry point using a JMP instruction.

The newly activated driver fork process executes under the following constraints:

- It cannot refer to the address space of the process initiating the I/O request.
- It can use only R0 through R5 freely. It must save other registers before use and restore them after use.
- It must clean up the stack after use. The stack must be in its original state when the fork process relinquishes control to any VAX/VMS routine.
- It must execute at IPLs between driver fork level and IPL$_POWER. It must not lower IPL below device fork except by creating a fork process at a lower IPL.

Each driver fork process executes until one of the following events occurs:

- Device-dependent processing of the I/O request is complete.
- A shared resource needed by the driver is unavailable, as described in Section 3.3.
- Device activity requires the fork process to wait for a device interrupt.

5.2.2 Activating a Device and Waiting for an Interrupt

A device driver's start I/O routine examines the I/O request packet to determine the type of I/O operation to perform and the I/O request specification. Depending on the device type supported by the driver, the start I/O routine performs some or all of the following steps:

- Analyzes the I/O function and branches to driver code that prepares the unit control block and the device for that I/O operation
- Copies I/O request packet fields into the unit control block
- Tests fields in the unit control block to determine whether the device and/or volume mounted on the device are valid
- If the device is attached to a multiunit controller, obtains the controller data channel
- If the I/O operation is a DMA transfer, obtains a UNIBUS adapter data path and loads UNIBUS adapter map registers
- Loads all necessary device registers except for the device's control/status register
- Raises IPL to IPL$_POWER and confirms that a power failure that would invalidate the device operation has not occurred
- Loads the device's control/status register to activate the device
- Invokes a VAX/VMS routine to suspend the driver fork process until a device interrupt or timeout occurs

While the driver is suspended, the context saved for it consists of the unit control block. The context contains the following information:

- A description of the I/O request and the state of the device
- The contents of R3 and R4
- The implicit contents of R5 as the address of the unit control block
- A driver return address
- The address of a device timeout handler
- The time at which the device will time out

By convention, R4 often contains the address of the control/status register (CSR); it permits the driver to examine device registers. When the driver fork process regains control after interrupt processing, R5 contains the UCB address; it is the key to the rest of the I/O data base that is relevant to the current I/O operation.

5.2.3 Handling a Device Interrupt

Once the driver's start I/O routine initiates the transfer, the driver invokes a VAX/VMS routine to wait for an interrupt. When the device requests an interrupt, the interrupt dispatcher transfers control to the driver interrupt service routine. The driver's interrupt service

routine runs at a high interrupt priority level so that the routine can service interrupts quickly. A driver interrupt service routine usually performs the following processing:

- For multiunit device controllers, determines which device unit generated the interrupt
- Examines the unit control block for the device to confirm that the driver fork process expects the interrupt
- Saves device registers
- Reactivates the suspended driver fork process

If necessary, the reactivated driver fork process executes at the high IPL of the interrupt service routine for a few instructions. Very soon, however, the driver lowers its execution priority so that it does not block subsequent interrupts for other devices in the system.

5.2.4 Switching from Interrupt to Fork Process Context

To lower its priority, the driver calls a VAX/VMS fork process queuing routine (IOFORK) that performs the following steps:

- Disables the timeout that was specified in the wait for interrupt routine
- Saves R3 and R4 (these are the registers needed to execute as a fork process)
- Saves the address of the instruction following the IOFORK request in the UCB fork block
- Places the address of the UCB fork block from R5 in a fork queue for the driver's fork level
- Returns to the driver's interrupt service routine

The interrupt service routine then cleans up the stack, restores registers, and dismisses the interrupt. Figure 5-7 illustrates the flow of a driver to create a fork process after a device interrupt.

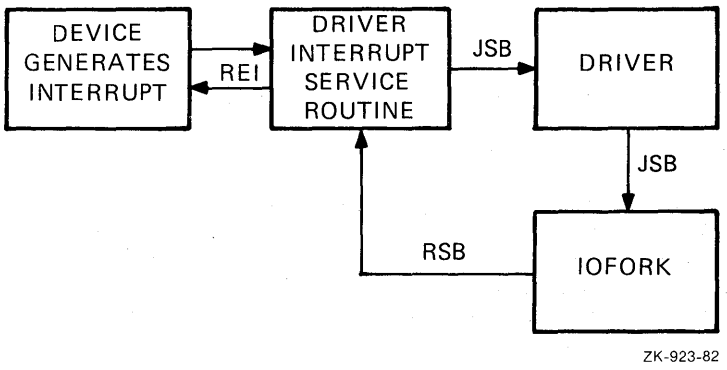


Figure 5-7: Creating a Fork Process After an Interrupt

5.2.5 Activating a Fork Process from a Fork Queue

When no hardware interrupts are pending, the software interrupt priority arbitration logic of the processor transfers control to the

software interrupt fork dispatcher. When the processor grants an interrupt at a fork IPL, the fork dispatcher processes the fork queue that corresponds to the IPL of the interrupt. To do so, the dispatcher performs the following steps:

- Removes a driver fork block from the fork queue
- Restores fork context
- Transfers control back to the fork process

Thus, the driver code calls VAX/VMS code that coordinates suspension and restoration of a driver fork process. This convention allows VAX/VMS to service hardware device interrupts in a timely manner and reactivate driver fork processes as soon as no device requires attention.

When a given fork process completes, the fork dispatcher removes the next entry, if any, from the fork queue, restores its fork process context, and reactivates it. This sequence repeats until the fork queue is empty. When the queue is empty, the fork dispatcher restores R0 through R5 from the stack and dismisses the interrupt with an REI instruction.

Figure 5-8 illustrates the reactivation of a driver fork process.

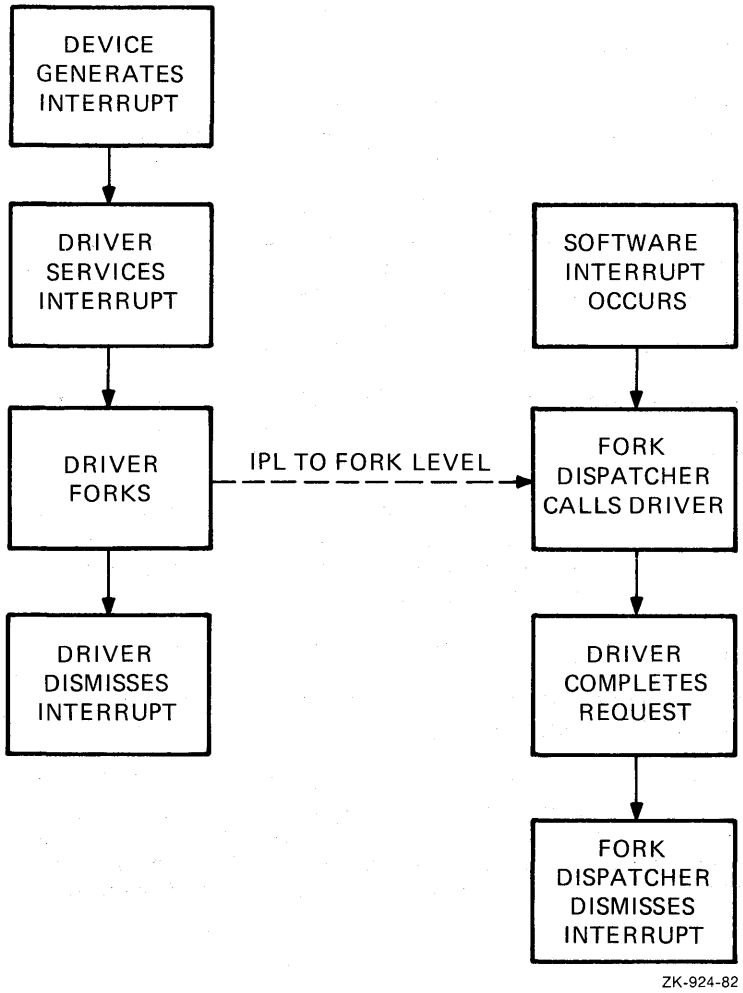


Figure 5-8: Reactivation of a Driver Fork Process

5.3 COMPLETION OF AN I/O REQUEST

Once reactivated, a driver fork process completes the I/O request as follows:

- Releases shared driver resources such as UNIBUS adapter and map registers and ownership of the controller
- Returns status to the VAX/VMS I/O completion routine

The I/O completion routine performs the following steps to start postprocessing of the I/O request and to start processing the next I/O request in the device's queue:

- Writes return status from the driver into the I/O request packet
- Inserts the finished I/O request packet in the I/O postprocessing fork queue and requests an interrupt at IPL$_IOPOST
- Creates a new fork process for the next I/O request packet in the device's I/O request packet wait queue
- Activates the new driver fork process

5.3.1 I/O Postprocessing

When processor priority drops below the I/O postprocessing IPL, the processor dispatches to the I/O postprocessing interrupt service routine. This VAX/VMS routine completes device-independent processing of the I/O request.

Using the I/O request packet as a source of information, the I/O postprocessing dispatcher executes the sequence below for each I/O request packet in the postprocessing queue:

- Removes the I/O request packet from the queue
- If the I/O function was a direct I/O function, adjusts the recorded use of the issuing process's direct I/O quota and unlocks the pages involved in the I/O transfer
- If the I/O function was a buffered I/O function, adjusts the recorded use of the issuing process's buffered I/O quota and, if the I/O was a write function, deallocates the system buffers used in the transfer
- Posts the event flag associated with the I/O request
- Queues a kernel mode AST routine to the process that issued the Queue I/O Request system service call

The queuing of a kernel mode AST routine allows I/O postprocessing to execute in the context of the user process but in a privileged access mode. Process context is needed to return the results of the I/O operation to the process's address space. The kernel mode AST routine writes the following data into the process's address space:

- Data read in a buffered I/O operation
- If specified in the I/O request, the contents of the diagnostic buffer
- If specified in the I/O request, the two longwords of I/O status

If the I/O request specifies a user AST routine, the kernel mode AST routine queues the user mode AST for the process. When VAX/VMS delivers the user mode AST, the system AST delivery routine deallocates the I/O request packet. The first part of an I/O request packet is the AST control block for user requested ASTs.

PART II

OVERVIEW

Device drivers consist of static tables, routines that perform I/O preprocessing, and routines that handle the device and controller. The chapters that follow describe how to write the following sections of a driver:

- Static tables
- Routines that use the device driver's function decision table (FDT)
- Routines that start an I/O operation on the device and complete the I/O operation
- Routines that handle interrupts
- Routines that request allocation of UNIBUS adapter map registers and data paths
- Routines that initialize devices and controllers
- Routines that cancel an I/O operation
- Routines that log errors

The "how to" chapters are preceded by a chapter that contains a driver template. The template illustrates the general organization and writing of a driver.

NOTE

The "how to" chapters describe a common approach to the design of various driver routines; they are examples. They do not present the only approach that can be taken to writing a driver.

CHAPTER 6

TEMPLATE FOR AN I/O DRIVER

The pages that follow describe conventions to be used by device drivers and provide a template for a device driver. Drivers do not necessarily need all of the routines indicated by the template, nor do driver routines and tables need to follow the exact order of the template. However, the VAX/VMS operating system does place a few restrictions on the order and content of driver routines and tables.

Figure 6-1 illustrates the organization of a device driver. The first item in a device driver is the driver prologue table. This table must be the first generated code in a driver. The order of the remaining tables and routines varies from driver to driver. However, the last statement in every driver, except for the .END assembly directive, must be a label marking the end of the driver. The address of this label is stored in the driver prologue table. The driver loading procedure uses this address to calculate the size of the driver. Chapter 14 describes the driver loading procedure.

Some drivers contain no device-dependent function decision table routines. Other drivers need only minimal initialization procedures. However, every driver normally contains static driver tables and a start I/O routine or an interrupt service routine.

6.1 CODING CONVENTIONS

The driver loading procedure loads a device driver into a block of nonpaged system memory whose location is chosen by the operating system memory allocation routines. Therefore, the driver must consist of position-independent code only.

In addition, the system may call a device driver repeatedly to process I/O requests and interrupts. The driver often does not complete one I/O operation before the system transfers control to the driver to begin another on a different unit. For this reason, the code must be reentrant.

The rules of position-independent and reentrant code are listed below.

- Instructions can branch only to relative addresses within the driver and to global addresses listed in the VAX/VMS symbol table (SYS$SYSTEM:SYS.STB).

- Static tables can list only relative addresses within the driver and global addresses.

- The driver cannot store temporary data in local driver tables for dynamic driver context. All dynamic temporary storage must be contained within the unit control block corresponding to an I/O request or the current I/O request block.

- The driver must refer to the I/O data base by loading the address of a data structure into a general register and using displacement addressing to the fields of the data structure.

Refer to the VAX-11 MACRO User's Guide for additional information about position-independent and reentrant code.

DRIVER ORGANIZATION

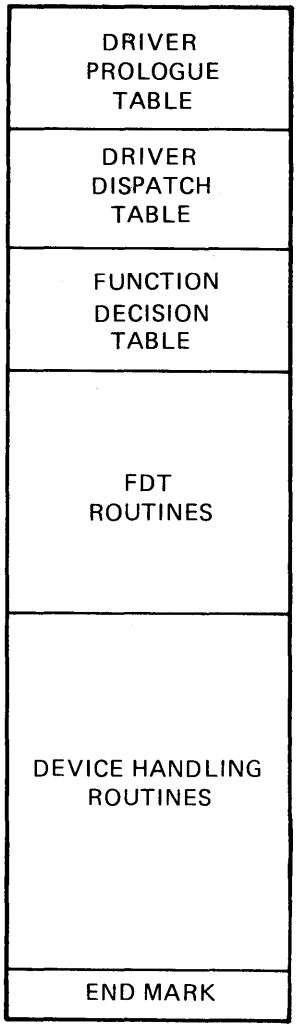


Figure 6-1: Driver Operation

Device drivers must also restrict their use of general registers and the stack:

- FDT routines can use R0 through R2 and R9 through R11 as available registers. The routines can use other registers by saving the registers before use and restoring them before exiting from the FDT routine.

- All other driver routines can use R0 through R5 as available registers. The routines can use other registers, if necessary, by saving and restoring them but using other registers in this way is discouraged.

- All driver routines can use the stack for temporary storage only if the routines restore the stack to its previous state before calling any VAX/VMS routines or executing RSB instructions.

6.2 RESTRICTIONS ON DEVICE REGISTER I/O SPACE USE

The programmer of a device driver for a UNIBUS device must observe the following restrictions on the use of a device registers:

- Drivers should always store the address of a device control register in a general register and then gain access to the device register indirectly through the general register. The example below defines symbolic word offsets for each device register and gains access to them using displacement mode addressing from R4.

```
;
; Device register offsets
;

LP_CSR = 0                              ; CSR offset
LP_DBR = 2                              ; Buffer address offset

   .
   .
   .
   MOVL   UCB$L_CRB(R5),R4              ; Get address of CRB
   MOVL   CRB$L_INTD+VEC$L_IDB(R4),R4   ; Get the address of
                                        ; the device's CSR
     .
     .
     .
   TSTW   LP_CSR(R4)                    ; Is printer online?
```

- Floating, double, field, queue, or quadword operands are not allowed in I/O address space, nor can an instruction obtain the position, size, length, or base of an operand from I/O space. For example, a driver cannot use a field instruction to test a bit in a device register.

- Drivers cannot use string instructions.

- Drivers can use only those instructions with a maximum of one modify or write destination. The destination must be the last operand.

- Registers of devices connected to the backplane interconnect (for example, UNIBUS adapter device registers and MASSBUS device registers) are longwords. Registers of devices connected to the UNIBUS are words. Instructions that refer to UNIBUS adapter registers must use longword context. All driver instructions that affect UNIBUS device registers must use word context, for example, BISW, MOVW, and ADDW3, unless the register is byte-addressable.

- An instruction that refers to I/O space must not generate an exception or be interrupted. If the instruction is allowed to restart, it will re-read the device register, which causes undesirable device side-effects or data loss.

- To access I/O space, use the instructions listed below. These instructions are not interruptible unless they use autoincrement deferred addressing mode or any of the displacement deferred modes when specifying an operand.

```
ADAWI                       MCOM(B,W,L)
ADD(B,W,L)2                 MFPR
ADD(B,W,L)3                 MNEG(B,W,L)
ADWC                        MOV(B,W,L)
BIC(B,W,L)2                 MOVA(B,W,L)
BIC(B,W,L)3                 MOVAQ
BICPSW                      MOVPSL
BIS(B,W,L)2                 MOVZ(BW,BL,WL)
BIS(B,W,L)3                 MTPR
BISPSL                      PROBE(R,W)
BISPSW                      PUSHA(B,W,L)
BIT(B,W,L)                  PUSHAQ
CASE(B,W,L)                 PUSHL
CHM(K,E,S,U)                SBWC
CLR(B,W,L)                  SUB(B,W,L)2
CMP(B,W,L)                  SUB(B,W,L)3
CVT(BW,BL,WB,               TST(B,W,L)
   WL,LB,LW)                XOR(B,W,L)2
DEC(B,W,L)                  XOR(B,W,L)3
INC(B,W,L)
```

The following pages list the VAX/VMS template driver. A machine-readable copy is also available. Its file specification is:

SYS$EXAMPLES:TDRIVER.MAR

```
        .TITLE  TDRIVER - VAX/VMS TEMPLATE DRIVER
        .IDENT  'V03-002'

;
;
;
;  Copyright (c) 1978,1979,1980, 1982
;  by DIGITAL Equipment Corporation, Maynard, Massachusetts
;
;  This software is furnished under a license and may be used and  copied
;  only  in  accordance  with  the  terms  of  such  license and with the
;  inclusion of the above copyright notice.  This software or  any  other
;  copies  thereof may not be provided or otherwise made available to any
;  other person.  No title to and ownership of  the  software  is  hereby
;  transferred.
;
;  The information in this software is subject to change  without  notice
;  and  should  not  be  construed  as  a commitment by DIGITAL Equipment
;  Corporation.
;
;  DIGITAL assumes no responsibility for the use or  reliability  of  its
;  software on equipment which is not supplied by DIGITAL.
;
;
;

;++
;
; FACILITY:
;
;       VAX/VMS Template driver
;
; ABSTRACT:
;
;       This module contains the outline of a driver:
;
;               Models of driver tables
;               Controller and unit initialization routines
;               An FDT routine
;               The start I/O routine
;               The interrupt service routine
;               The cancel I/O routine
;               The device register dump routine
;
; AUTHOR:
;
;       S. Programmer   11-NOV-1979
;
; REVISION HISTORY:
;
;       V02     JHP001  J. Programmer   2-Aug-1979    11:27
;               Remove BLBC instruction from CANCEL routine.
;
;       V02-001 JHP001  J. Programmer   11-Feb-1981     13:10
;               Add description of reason argument to CANCEL
;               routine.  Correct references to channel index
;               number.
;
;--
```

```
        .SBTTL  External and local symbol definitions

;
; External symbols
;
        $CANDEF                         ; Cancel reason codes
        $CRBDEF                         ; Channel request block
        $DCDEF                          ; Device classes and types
        $DDBDEF                         ; Device data block
        $DEVDEF                         ; Device characteristics
        $IDBDEF                         ; Interrupt dispatch block
        $IODEF                          ; I/O function codes
        $IPLDEF                         ; Hardware IPL definitions
        $IRPDEF                         ; I/O request packet
        $SSDEF                          ; System status codes
        $UCBDEF                         ; Unit control block
        $VECDEF                         ; Interrupt vector block

;
; Local symbols
;

;
; Argument list (AP) offsets for device-dependent QIO parameters
;

P1      = 0                             ; First QIO parameter
P2      = 4                             ; Second QIO parameter
P3      = 8                             ; Third QIO parameter
P4      = 12                            ; Fourth QIO parameter
P5      = 16                            ; Fifth QIO parameter
P6      = 20                            ; Sixth QIO parameter

;
; Other constants
;

TD_DEF_BUFSIZ   = 1024                  ; Default buffer size
TD_TIMEOUT_SEC  = 10                    ; 10 second device timeout
TD_NUM_REGS     = 4                     ; Device has 4 registers

;
; Definitions that follow the standard UCB fields
;

        $DEFINI UCB                     ; Start of UCB definitions

        .=UCB$K_LENGTH                  ; Position at end of UCB

$DEF    UCB$W_TD_WORD                   ; A sample word
                        .BLKW   1
$DEF    UCB$W_TD_STATUS                 ; Device's CSR register
                        .BLKW   1
$DEF    UCB$W_TD_WRDCNT                 ; Device's word count register
                        .BLKW   1
$DEF    UCB$W_TD_BUFADR                 ; Device's buffer address
                        .BLKW   1       ; register
$DEF    UCB$W_TD_DATBUF                 ; Device's data buffer register
                        .BLKW   1
$DEF    UCB$K_TD_UCBLEN                 ; Length of extended UCB
```

```
;
; Bit positions for device-dependent status field in UCB
;
        $VIELD  UCB,0,<-                ; Device status
                <BIT_ZERO,,M>,-         ; First bit
                <BIT_ONE,,M>,-          ; Second bit
                >

        $DEFEND UCB                     ; End of UCB definitions

;
; Device register offsets from CSR address
;
        $DEFINI TD                      ; Start of status definitions

$DEF    TD_STATUS                       ; Control/status
                        .BLKW   1

;
; Bit positions for device control/status register
;

        _VIELD  TD_STS,0,<-             ; Control/status register
                <GO,,M>,-               ; Start device
                <BIT1,,M>,-             ; Bit one
                <BIT2,,M>,-             ; Bit two
                <BIT3,,M>,-             ; Bit three
                <XBA,2,M>,-             ; Extended address bits
                <INTEN,,M>,-            ; Enable interrupts
                <READY,,M>,-            ; Device ready for command
                <BIT8,,M>,-             ; Bit eight
                <BIT9,,M>,-             ; Bit nine
                <BIT10,,M>,-            ; Bit ten
                <BIT11,,M>,-            ; Bit eleven
                <,1>,-                  ; Disregarded bit
                <ATTN,,M>,-             ; Attention bit
                <NEX,,M>,-              ; Nonexistent memory flag
                <ERROR,,M>,-            ; Error or external interrupt
        >

$DEF    TD_WRDCNT                       ; Word count
                        .BLKW   1
$DEF    TD_BUFADR                       ; Buffer address
                        .BLKW   1
$DEF    TD_DATBUF                       ; Data buffer
                        .BLKW   1

        $DEFEND TD                      ; End of device register
                                        ; definitions.
```

```
        .SBTTL  Standard tables

;
; Driver prologue table
;

        DPTAB   -                                       ; DPT-creation macro
                END=TD_END,-                            ; End of driver label
                ADAPTER=UBA,-                           ; Adapter type
                UCBSIZE=<UCB$K_TD_UCBLEN>,-             ; Length of UCB
                NAME=TDDRIVER                           ; Driver name
        DPT_STORE INIT                                  ; Start of load
                                                        ; initialization table
        DPT_STORE UCB,UCB$B_FIPL,B,8                    ; Device fork IPL
        DPT_STORE UCB,UCB$B_DIPL,B,22                   ; Device interrupt IPL
        DPT_STORE UCB,UCB$L_DEVCHAR,L,<-                ; Device characteristics
                DEV$M_IDV!-                             ;   input device
                DEV$M_ODV>                              ;   output device
        DPT_STORE UCB,UCB$B_DEVCLASS,B,DC$_SCOM         ; Sample device class
        DPT_STORE UCB,UCB$W_DEVBUFSIZ,W,-               ; Default buffer size
                TD_DEF_BUFSIZ

        DPT_STORE REINIT                                ; Start of reload
                                                        ; initialization table
        DPT_STORE DDB,DDB$L_DDT,D,TD$DDT                ; Address of DDT
        DPT_STORE CRB,CRB$L_INTD+4,D,-                  ; Address of interrupt
                TD_INTERRUPT                            ; service routine
        DPT_STORE CRB,-                                 ; Address of controller
                CRB$L_INTD+VEC$L_INITIAL,-              ; initialization routine
                D,TD_CONTROL_INIT
        DPT_STORE CRB,-                                 ; Address of device
                CRB$L_INTD+VEC$L_UNITINIT,-             ; unit initialization
                D,TD_UNIT_INIT                          ; routine

        DPT_STORE END                                   ; End of initialization
                                                        ; tables

;
; Driver dispatch table
;

        DDTAB   -                                       ; DDT-creation macro
                DEVNAM=TD,-                             ; Name of device
                START=TD_START,-                        ; Start I/O routine
                FUNCTB=TD_FUNCTABLE,-                   ; FDT address
                CANCEL=TD_CANCEL,-                      ; Cancel I/O routine
                REGDMP=TD_REG_DUMP                      ; Register dump routine

;
; Function decision table
;
```

```
TD_FUNCTABLE:                                   ; FDT for driver
        FUNCTAB ,-                              ; Valid I/O functions
                <READVBLK,-                     ; Read virtual
                READLBLK,-                      ; Read logical
                READPBLK,-                      ; Read physical
                WRITEVBLK,-                     ; Write virtual
                WRITELBLK,-                     ; Write logical
                WRITEPBLK,-                     ; Write physical
                SETMODE,-                       ; Set device mode
                SETCHAR>                        ; Set device chars.
        FUNCTAB ,                               ; No buffered functions
        FUNCTAB +EXE$READ,-                     ; FDT read routine for
                <READVBLK,-                     ; read virtual,
                READLBLK,-                      ; read logical,
                READPBLK>                       ; and read physical.
        FUNCTAB +EXE$WRITE,-                    ; FDT write routine for
                <WRITEVBLK,-                    ; write virtual,
                WRITELBLK,-                     ; write logical,
                WRITEPBLK>                      ; and write physical.
        FUNCTAB +EXE$SETMODE,-                  ; FDT set mode routine
                <SETCHAR,-                      ; for set chars. and
                SETMODE>                        ; set mode.
```

```
        .SBTTL  TD_CONTROL_INIT, Controller initialization routine

;++
; TD_CONTROL_INIT, Readies controller for I/O operations
;
; Functional description:
;
;       The operating system calls this routine in 3 places:
;
;               at system startup
;               during driver loading and reloading
;               during recovery from a power failure
;
; Inputs:
;
;       R4      - address of the CSR (controller status register)
;       R5      - address of the IDB (interrupt dispatch block)
;       R6      - address of the DDB (device data block)
;       R8      - address of the CRB (channel request block)
;
; Outputs:
;
;       The routine must preserve all registers except R0-R3.
;
;--

TD_CONTROL_INIT:                        ; Initialize controller
        RSB                             ; Return
```

```
        .SBTTL  TD_UNIT_INIT, Unit initialization routine

;++
; TD_UNIT_INIT, Readies unit for I/O operations
;
; Functional description:
;
;       The operating system calls this routine after calling the
;       controller initialization routine:
;
;               at system startup
;               during driver loading
;               during recovery from a power failure
;
; Inputs:
;
;       R4      - address of the CSR (controller status register)
;       R5      - address of the UCB (unit control block)
;
; Outputs:
;
;       The routine must preserve all registers except R0-R3.
;
;--

TD_UNIT_INIT:                           ; Initialize unit
        BISW    #UCB$M_ONLINE, -
                UCB$W_STS(R5)           ; Set unit online
        RSB                             ; Return
```

```
        .SBTTL  TD_FDT_ROUTINE, Sample FDT routine

;++
; TD_FDT_ROUTINE, Sample FDT routine
;
; Functional description:
;
;       SUPPLIED BY USER
;
; Inputs:
;
;       R0-R2   - scratch registers
;       R3      - address of the IRP (I/O request packet)
;       R4      - address of the PCB (process control block)
;       R5      - address of the UCB (unit control block)
;       R6      - address of the CCB (channel control block)
;       R7      - bit number of the I/O function code
;       R8      - address of the FDT table entry for this routine
;       R9-R11  - scratch registers
;       AP      - address of the 1st function dependent QIO parameter
;
; Outputs:
;
;       The routine must preserve all registers except R0-R2, and
;       R9-R11.
;
;--

TD_FDT_ROUTINE:                         ; Sample FDT routine
        RSB                             ; Return
```

```
        .SBTTL  TD_START, Start I/O routine

;++
; TD_START - Start a transmit, receive, or set mode operation
;
; Functional description:
;
;       SUPPLIED BY USER
;
; Inputs:
;
;       R3      - address of the IRP (I/O request packet)
;       R5      - address of the UCB (unit control block)
;
; Outputs:
;
;       R0      - 1st longword of I/O status: contains status code and
;                 number of bytes transferred
;       R1      - 2nd longword of I/O status: device-dependent
;
;       The routine must preserve all registers except R0-R2 and R4.
;
;--

TD_START:                                       ; Process an I/O packet

        WFIKPCH TD_TIMEOUT,#TD_TIMEOUT_SEC
;
; After a transfer completes successfully, return the number of bytes
; transferred and a success status code.
;

        IOFORK
        INSV    UCB$W_BCNT(R5),#16,-            ; Load number of bytes trans-
                #16,R0                          ; ferred into high word of R0.
        MOVW    #SS$_NORMAL,R0                  ; Load a success code into R0.

;
; Call I/O postprocessing.
;

COMPLETE_IO:                                    ; Driver processing is finished.
        REQCOM                                  ; Complete I/O.

;
; Device timeout handling. Return an error status code.
;

TD_TIMEOUT:                                     ; Timeout handling
        SETIPL  UCB$B_FIPL(R5)                  ; Lower to driver fork IPL
        MOVZWL  #SS$_TIMEOUT,R0                 ; Return error status.
        BRB     COMPLETE_IO                     ; Call I/O postprocessing.
```

```
        .SBTTL  TD_INTERRUPT, Interrupt service routine

;++
; TD_INTERRUPT, Analyzes interrupts, processes solicited interrupts
;
; Functional description:
;
;       The sample code assumes either
;
;               that the driver is for a single-unit controller, and
;               that the unit initialization code has stored the
;               address of the UCB in the IDB; or
;
;               that the driver's start I/O routine acquired the
;               controller's channel with a REQPCHANL macro call, and
;               then invoked the WFIKPCH macro to keep the channel
;               while waiting for an interrupt.
;
; Inputs:
;
;        0(SP)  - pointer to the address of the IDB (interrupt dispatch
;                 block)
;        4(SP)  - saved R0
;        8(SP)  - saved R1
;       12(SP)  - saved R2
;       16(SP)  - saved R3
;       20(SP)  - saved R4
;       24(SP)  - saved R5
;       28(SP)  - saved PSL (program status longword)
;       32(SP)  - saved PC
;
;       The IDB contains the CSR address and the UCB address.
;
; Outputs:
;
;       The routine must preserve all registers except R0-R5.
;
;--

TD_INTERRUPT:                                   ; Service device interrupt
        MOVL    @(SP)+,R4                       ; Get address of IDB and remove
                                                ; pointer from stack.
        MOVL    IDB$L_OWNER(R4),R5              ; Get address of device owner's
                                                ; UCB.
        MOVL    IDB$L_CSR(R4),R4                ; Get address of device's CSR.
        BBCC    #UCB$V_INT,-                    ; If device does not expect
                UCB$W_STS(R5),-                 ; interrupt, dismiss it.
                UNSOL_INTERRUPT

;
; This is a solicited interrupt. Save
; the contents of the device registers in the UCB.
;

        MOVW    TD_STATUS(R4),-                 ; Otherwise, save all device
                UCB$W_TD_STATUS(R5)             ; registers. First the CSR.
        MOVW    TD_WRDCNT(R4),-                 ; Save the word count register.
                UCB$W_TD_WRDCNT(R5)
        MOVW    TD_BUFADR(R4),-                 ; Save the buffer address
                UCB$W_TD_BUFADR(R5)             ; register.
        MOVW    TD_DATBUF(R4),-                 ; Save the data buffer register.
                UCB$W_TD_DATBUF(R5)
```

```
;
; Restore control to the main driver.
;

RESTORE_DRIVER:                             ; Jump to main driver code.
        MOVL    UCB$L_FR3(R5),R3            ; Restore driver's R3 (use a
                                            ; MOVQ to restore R3-R4).
        JSB     @UCB$L_FPC(R5)              ; Call driver at interrupt
                                            ; wait address.

;
; Dismiss the interrupt.
;

UNSOL_INTERRUPT:                            ; Dismiss unsolicited interrupt.
        POPR    #^M<R0,R1,R2,R3,R4,R5>      ; Restore R0-R5
        REI                                 ; Return from interrupt.
```

```
        .SBTTL  TD_CANCEL, Cancel I/O routine

;++
; TD_CANCEL, Cancels an I/O operation in progress
;
; Functional description:
;
;       This routine calls IOC$CANCELIO to set the cancel bit in the
;       UCB status word if:
;
;               the device is busy,
;               the IRP's process ID matches the cancel process ID,
;               the IRP channel matches the cancel channel.
;
;       If IOC$CANCELIO sets the cancel bit, then this driver routine
;       does device-dependent cancel I/O fixups.
;
; Inputs:
;
;       R2      - channel index number
;       R3      - address of the current IRP (I/O request packet)
;       R4      - address of the PCB (process control block) for the
;                 process canceling I/O
;       R5      - address of the UCB (unit control block)
;       R8      - cancel reason code, one of:
                        CAN$C_CANCEL    if called through $CANCEL or
                                        $DALLOC system service
                        CAN$C_DASSGN    if called through $DASSGN
                                        system service
;
; Outputs:
;
;       The routine must preserve all registers except R0-R3.
;
;       The routine may set the UCB$M_CANCEL bit in UCB$W_STS.
;
;--

TD_CANCEL:                              ; Cancel an I/O operation
        JSB     G^IOC$CANCELIO          ; Set cancel bit if appropriate.
        BBC     #UCB$V_CANCEL,-         ; If the cancel bit is not set,
                UCB$W_STS(R5),10$       ; just return.

;
; Device-dependent cancel operations go next.
;

;
; Finally, the return.
;

10$:
        RSB                             ; Return
```

```
        .SBTTL  TD_REG_DUMP, Device register dump routine

;++
; TD_REG_DUMP, Dumps the contents of device registers to a buffer
;
; Functional description:
;
;       Writes the number of device registers, and their current
;       contents into a diagnostic or error buffer.
;
; Inputs:
;
;       R0      - address of the output buffer
;       R4      - address of the CSR (controller status register)
;       R5      - address of the UCB (unit control block)
;
; Outputs:
;
;       The routine must preserve all registers except R1-R3.
;
;       The output buffer contains the current contents of the device
;       registers. R0 contains the address of the next empty longword in
;       the output buffer.
;
;--

TD_REG_DUMP:                                    ; Dump device registers
        MOVZBL  #TD_NUM_REGS,(R0)+              ; Store device register count.
        MOVZWL  UCB$W_TD_STATUS(R5),-           ; Store device status register.
                (R0)+
        MOVZWL  UCB$W_TD_WRDCNT(R5),-           ; Store word count register.
                (R0)+
        MOVZWL  UCB$W_TD_BUFADR(R5),-           ; Store buffer address register.
                (R0)+
        MOVZWL  UCB$W_TD_DATBUF(R5),-           ; Store data buffer register.
                (R0)+
        RSB                                     ; Return
```

```
        .SBTTL  TD_END, End of driver

;++
; Label that marks the end of the driver
;--

TD_END:                                 ; Last location in driver
        .END
```

CHAPTER 7

WRITING DEVICE DRIVER TABLES

Every device driver declares three static tables that describe the device and driver:

- **Driver prologue table** that describes the device type, driver name, and fields in the I/O data base to be initialized during driver loading and reloading
- **Driver dispatch table** that lists some of the driver entry points to which VAX/VMS transfers control; the channel request block and function decision table list other entry points
- **Function decision table** that lists valid functions of the driver and entry points to routines that perform I/O preprocessing for each function

The VAX/VMS operating system provides macros that drivers can invoke to create the tables listed above. Descriptions of individual tables in the sections that follow also describe the macros invoked to create the tables. All of the macros described in this chapter are keyword macros; that is, parameter values can be expressed in the following format:

```
KEYWORD=parameter-value
```

The VAX-11 MACRO Language Reference Manual describes the syntax rules for keyword macros in detail. The sections that follow provide examples of macro usage.

7.1 DRIVER PROLOGUE TABLE (DPT)

The driver prologue table is the first generated code in every device driver. This table, along with parameters to the SYSGEN command that request driver loading, describes the driver to the driver loading procedure. In turn, the driver loading procedure computes the size of the driver, loads it into nonpaged system memory, and creates control blocks for the new device(s) in the I/O data base. Chapter 14 describes how the driver loading procedure decides which control blocks to build for a given device.

Device drivers can pass control block initialization information to the driver loading procedure through values stored in the driver prologue table. In addition, the driver loading procedure initializes some fields within the device control blocks using information from its own tables. Drivers must treat many of the fields initialized by the driver loading procedure as read-only fields. These fields are marked with an asterisk (*) in Appendix A.

To create a driver prologue table, the driver invokes the DPTAB macro, described in Section 7.1.1.

When the DPTAB macro expands, it creates a control block that the driver loading procedure uses to load the driver. The loading procedure loads the driver prologue table and the driver together in virtual memory. The loading procedure also links the new driver prologue table into a list of all driver prologue tables known to the system.

Most device drivers need to initialize certain fields of the I/O data base with driver-specific values. The DPT_STORE macro provides the driver with a means of communicating its initialization needs to the driver loading procedure. When invoked, the DPT_STORE macro places information in the driver prologue table that the driver loading procedure uses to load specified values into specified fields. The DPT_STORE macro accepts two lists of fields:

- Fields to be initialized when the control blocks are built using the SYSGEN command CONNECT and when the driver is reloaded
- Fields to be initialized only when the driver is reloaded using the SYSGEN command RELOAD

The DPTAB macro stores the relative addresses of these two lists, called initialization and reinitialization data, in the driver prologue table. The list of one or more invocations of the DPT_STORE macro must appear after the DPTAB macro. Section 7.1.2 describes the format of the DPT_STORE macro.

Drivers must use the DPT_STORE macro to supply initialization data for the following fields:

UCB$B_FIPL	Driver fork IPL
UCB$B_DIPL	Hardware device IPL
UCB$L_DEVCHAR	Device characteristics (see Appendix A)

The driver also must provide reinitialization data for the device data block field DDB$L_DDT and for any of the following routine addresses in the channel request block:

DDB$L_DDT	Address of the driver dispatch table
CRB$L_INTD+4	Entry point to the driver interrupt service routine, if one exists
CRB$L_INTD+VEC$L_INITIAL	Address of a controller initialization routine, if one exists
CRB$L_INTD+VEC$L_UNITINIT	Address of a device unit initialization routine, if one exists. This entry point is used by UNIBUS devices.

7.1.1 DPTAB Macro

The DPTAB macro creates a driver prologue table.

Format

```
DPTAB  end,adapter,[flags],ucbsize,[unload],[maxunits],[defunits],
       [deliver],[vector],name
```

end
The address of the end of the driver module.

adapter
The adapter type.

```
UBA  =  UNIBUS adapter
MBA  =  MASSBUS adapter
DR   =  DR device
NULL =  No actual device for driver
```

flags
The driver loader flags.

DPT$M_SVP — Indicates, when set, that the device requires a permanently allocated system page. This flag causes the driver loading procedure to allocate a permanent system page table entry for the device. The virtual address of the system page table entry is written into the system page field of the UCB (UCB$L_SVPN) during creation of the UCB. Disk drivers use this page table entry during ECC error correction.

DPT$M_NOUNLOAD — Indicates, when set, that the driver cannot be reloaded. A system bootstrap must occur before drivers with this bit set can be reloaded.

ucbsize
The size of each device unit control block in bytes. This argument is required. This field allows drivers to extend the unit control block to store device-dependent data describing an I/O operation. Appendix A provides examples. Driver routines and VAX/VMS ECC routines interpret fields in the extended part of the unit control block. The amount that the unit control block is extended is variable for each driver type.

unload
The address of a routine to call before the driver is reloaded. The driver loading procedure calls this routine before reinitializing all controllers and device units associated with the driver.

maxunits
The maximum number of units on a controller that this driver supports. This field affects the size of the interrupt dispatch block created the SYSGEN CONNECT command. If this field is omitted, the default is 8 units. You can override the maxunits field by appending the /MAXUNITS qualifier to the CONNECT command.

defunits
The number of units created by default for each controller that the AUTOCONFIGURE command to SYSGEN processes on behalf of this driver. The unit numbers created are zero through defunits minus one. If the deliver argument to the DPTAB macro is omitted, AUTOCONFIGURE creates the number of units specified by defunits. If the deliver argument is present, it names an action routine that AUTOCONFIGURE calls to determine whether or not to create each unit automatically.

deliver
The address of a unit delivery action routine that AUTOCONFIGURE calls to determine which units to configure automatically for the device supported by this driver.

vector
The address of a driver-specific transfer vector. Use of this argument is reserved to DIGITAL.

name
The name of the device driver module. The driver loading procedure will permit only one copy of the driver associated with the name given in this field to be loaded. By convention, a driver name is formed by appending the string DRIVER to the 2-alphabetic character generic device name, for example, DBDRIVER.

7.1.2 DPT_STORE Macro

The DPT_STORE macro either declares an assembly language label or describes a field to be initialized. When the macro declares a label, the macro has format 1. When the DPT_STORE macro describes a field to be initialized, the macro has format 2.

Format 1

```
DPT_STORE label-name
```

label-name
The name of the label to be declared. It can be one of the following:

INIT	Indicates the start of fields to initialize when the driver is loaded.
REINIT	Indicates the start of additional fields to initialize when the driver is loaded or reloaded.
END	Indicates the end of the two lists.

Format 2

```
DPT_STORE       struc-type,struc-offset,operation,expression,
                [position],[size]
```

struc-type
The type of I/O data base control block that contains the field to be initialized. The type can be one of the following:

DDB	device data block
UCB	unit control block
CRB	channel request block
IDB	interrupt dispatch block

struc-offset
The unsigned offset into the control block. The driver loading procedure can initialize only the first 256 bytes of each data structure. Unit and controller initialization routines can initialize additional data fields.

operation

The type of operation to be performed. The type can be one of the following:

B	write a byte value
W	write a word value
L	write a longword value
D	write an address relative to the driver
V	write a bit field

The V operation takes the following longword of data and the position and size arguments as operands of an INSV instruction.

An at sign (@) preceding the operation parameter indicates that the expression parameter that follows is the address of the initialization data.

expression

An expression to be stored in the control block or, if an at sign (@) is specified preceding the operation parameter, the address of an expression. For example, the following macro indicates that DEVICE_CHARS is the address of the data to write into the DEVCHAR field of the UCB.

```
DPT_STORE UCB,UCB$L_DEVCHAR,@L,DEVICE_CHARS
```

position

The starting bit position within the specified field. This parameter is specified only for V operations.

size

The number of bits in the field. This parameter is specified only for V operations.

7.1.3 Example of DPTAB and DPT_STORE Macro Use

The following example invokes the DPTAB macro and DPT_STORE macros to describe a device driver and its data base.

```
DPTAB -                                 ; Define DPT
      END=XX_END,-                      ; End of driver
      ADAPTER=UBA,-                     ; Adapter type
      UCBSIZE=UCB$K_XX_LENGTH, -        ; Size of UCB
      NAME=XXDRIVER                     ; Name of driver module
DPT_STORE INIT                          ; Start of control block
                                        ; initialization values
DPT_STORE UCB,UCB$B_FIPL,B,8            ; Driver fork IPL
DPT_STORE UCB,UCB$L_DEVCHAR,L,-         ; Device characteristics:
          <DEV$M_REC-                   ; record-oriented
          !DEV$M_AVL-                   ; available
          !DEV$M_ODV>                   ; output device
DPT_STORE UCB,UCB$B_DEVCLASS,B,-        ; Device class
          DC$_XX
DPT_STORE UCB,UCB$B_DEVTYPE,B,-         ; Device type
          XX$_XL78
DPT_STORE UCB,UCB$W_DEVBUFSIZ,W,-       ; Default buffer size
          132
DPT_STORE UCB,UCB$B_DIPL,B,22           ; Device IPL
```

```
DPT_STORE REINIT                    ; Start of control block
                                    ; reinitialization values
DPT_STORE CRB,CRB$L_INTD+4,D,-      ; Interrupt service
          XX_INTERRUPT              ; routine address
DPT_STORE CRB,CRB$L_INTD+VEC$L_UNITINIT,-
          D,XX_XL78_INIT            ; Unit initialization
                                    ; routine address
DPT_STORE DDB,DDB$L_DDT,D,XX$DDT    ; Address of driver
                                    ; dispatch table
DPT_STORE END                       ; End of field
                                    ; initialization
```

7.2 DRIVER DISPATCH TABLE (DDT)

The driver dispatch table lists some of the entry points for driver routines to be called by VAX/VMS for I/O processing. Every driver must create a driver dispatch table. The routines listed can reside in the driver module or in a VAX/VMS module. Appendix A describes the VAX/VMS device-independent routines that can be specified. Device-dependent routines are normally located in the driver module. The driver dispatch table contains relative addresses for routines located in the driver module and absolute addresses for routines located in the operating system. At load time, the driver loading procedure changes the relative addresses of driver routines to absolute addresses.

The driver creates the driver dispatch table by invoking the macro DDTAB. The driver loading procedure writes the address of the driver dispatch table, as specified in a DPT_STORE macro, into the device data block.

7.2.1 DDTAB Macro

The DDTAB macro creates a driver dispatch table. The table has a label of devnam$DDT. Just preceding the table, DDTAB generates the driver code program section with the following statement:

```
.PSECT $$$115_DRIVER
```

Format

```
DDTAB devnam,start,[unsolic],functb,[cancel],[regdmp],[diagbf],
      [erlgbf],[unitinit],[altstart],[mntver]
```

devnam
The generic name of the device driven by this device driver.

start
The address of the driver's start I/O routine.

unsolic
The address of the routine that services unsolicited interrupts from the device. This field is used only by MASSBUS devices.

functb
The address of the function decision table for this driver.

cancel
The address of the cancel I/O operation routine.

regdmp
The address of the routine that dumps the device registers to an error log buffer or to a diagnostic buffer.

diagbf
The length in bytes of the diagnostic buffer used for this device.

erlgbf
The length in bytes of the error log buffer used for this device.

unitinit
The address of the device initialization routine, if one exists. MASSBUS drivers should use this field rather than CRB$L_INTD + VEC$L_UNITINIT. UNIBUS drivers may use either one.

altstart
The address of the alternate start I/O routine. To initiate this routine, use the VAX/VMS routine EXE$ALTQUEPKT instead of EXE$QIODRVPKT.

mntver
The address of a VAX/VMS routine that is called at the beginning and end of a mount verification operation. If no routine is specified, the routine IOC$MNTVER is called. Use of this field to call any routine other than IOC$MNTVER is reserved to DIGITAL.

The DDTAB macro writes the address of the VAX/VMS routine IOC$RETURN into routine address fields of the driver dispatch table that are not supplied in the macro invocation (with the exception of the mntver argument). IOC$RETURN executes an RSB instruction; for further information, refer to Appendix C.

In the example below, notice that a plus sign (+) precedes the address of the entry point to the cancel I/O routine. The plus sign indicates that the routine is part of VAX/VMS. No plus sign precedes the address of the start I/O routine because it is part of the driver module. Omitting a required plus sign is a common error in device drivers.

7.2.2 Example of a DDTAB Macro

A sample invocation of the DDTAB macro follows.

```
DDTAB   DEVNAM=XX,-                ; Driver dispatch table
        START=STARTIO,-            ; Start I/O operation
        FUNCTB=FUNCTABLE,-         ; Function decision table
        CANCEL=+IOC$CANCELIO       ; Cancel I/O
```

7.3 FUNCTION DECISION TABLE (FDT)

The function decision table lists codes for I/O functions that are valid for the device; indicates whether the functions are buffered I/O functions; and specifies routines to perform preprocessing for particular functions. Every device driver must create a function decision table containing three or more entries:

- The list of valid I/O function codes
- The list of buffered I/O function codes

- One or more entries, each of which specifies all or a subset of I/O function codes and the address of a routine that performs I/O preprocessing for those function codes

If no buffered I/O functions are defined for the device, the second entry contains an empty list.

Taken together, the third through last entries in the function decision table specify one or more FDT routines for each valid I/O function code for the device. It is the responsibility of the FDT routines to terminate the I/O preprocessing for each type of function by transferring control out of the Queue I/O Request system service and into a routine that queues the I/O request to a driver, inserts the I/O request in the postprocessing queue, or aborts the I/O request.

Refer to Chapter 8 for information on the writing of FDT routines.

Table 7-1 lists the physical, logical, and virtual I/O function codes that a function decision table most commonly uses. A complete list of function codes is contained in the macro $IODEF in SYS$LIBRARY:STARLET.MLB.

7.3.1 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 a 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. For example, the assembly code that follows defines three device-specific physical I/O function codes.

```
IO$_STARTCLOCK=IO$_ERASETAPE      ; Start hardware clock
IO$_STOPCLOCK=IO$_OFFSET          ; Stop hardware clock
IO$_STARTDATA=IO$_SPACEFILE       ; Start data acquisition
```

The device driver creates a function decision table by invoking the FUNCTAB macro. Each invocation of the FUNCTAB macro creates a 2- or 3-longword entry in the function decision table. The first two invocations create 2-longword entries because they specify only function codes; they do not specify an accompanying action routine.

All subsequent invocations of the FUNCTAB macro must specify both function codes and the address of an action routine that is to perform preprocessing for those function codes. These invocations create 3-longword entries.

The Queue I/O Request system service processes entries in the order in which they appear in the function decision table. When a function code is present in more than one 3-longword entry, the system service sequentially calls every action routine specified for the function code until an action routine stops the scan by aborting, completing, or queuing an I/O request.

Table 7-1: VAX/VMS I/O Function Codes

Type of Function		Codes Defined
Physical codes	IO$_DIAGNOSE	Diagnose
	IO$_DRVCLR	Drive clear
	IO$_ERASETAPE	Erase tape
	IO$_NOP	No operation
	IO$_OFFSET	Offset read heads
	IO$_PACKACK	Pack acknowledge
	IO$_READHEAD	Read header and data
	IO$_READPBLK	Read physical block
	IO$_READPRESET	Read in preset
	IO$_READTRACKD	Read track data
	IO$_RECAL	Recalibrate drive
	IO$_RELEASE	Release port
	IO$_RETCENTER	Return to center line
	IO$_SEARCH	Search for sector
	IO$_SEEK	Seek cylinder
	IO$_SENSECHAR	Sense device characteristics
	IO$_SETCHAR	Set device characteristics
	IO$_SPACEFILE	Space files
	IO$_SPACERECORD	Space records
	IO$_STARTSPNDL	Start spindle
	IO$_UNLOAD	Unload drive
	IO$_WRITECHECK	Write check data
	IO$_WRITECHECKH	Write check header and data
	IO$_WRITEHEAD	Write header and data
	IO$_WRITEMARK	Write tape mark
	IO$_WRITEPBLK	Write physical block
	IO$_WRITETRACKD	Write track data
Logical codes	IO$_READLBLK	Read logical block
	IO$_REWIND	Rewind tape
	IO$_REWINDOFF	Rewind and set offline
	IO$_SENSEMODE	Sense device mode
	IO$_SETMODE	Set mode
	IO$_SKIPFILE	Skip files
	IO$_SKIPRECORD	Skip records
	IO$_WRITELBLK	Write logical block
	IO$_WRITEOF	Write end of file
Virtual codes	IO$_ACCESS	Access file
	IO$_ACPCONTROL	Miscellaneous ACP control
	IO$_CREATE	Create file
	IO$_DEACCESS	Deaccess file
	IO$_DELETE	Delete file
	IO$_MODIFY	Modify file
	IO$_MOUNT	Mount volume
	IO$_READPROMPT	Read terminal with prompt message
	IO$_READVBLK	Read virtual block
	IO$_WRITEVBLK	Write virtual block

7.3.2 Determining Those Functions that are Buffered I/O

The second entry in a function decision table indicates those functions that are handled as buffered I/O operations. In selecting the functions that are to be buffered, you should take the following information into consideration:

- Direct I/O is intended only for devices whose I/O operations always complete quickly. For example, although terminal I/O is 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, a terminal interrupt service routine) cannot gain access to the data if direct I/O is used. Therefore, transfers that require data manipulation must be buffered I/O.
- VAX/VMS handles the quotas differently for direct I/O and buffered I/O, as described in the VAX/VMS System Management and Operations Guide.
- Generally, DMA devices use direct I/O, while programmed I/O devices use buffered I/O.

Section 7.3.4 provides an example of the functions handled as buffered I/O operations.

7.3.3 FUNCTAB Macro

The FUNCTAB macro creates the function decision table for a driver.

Format

```
FUNCTAB     [action],codes
```

action
The address of an action routine to call during I/O preprocessing of the specified action code or codes. An action routine is specified only for the third through last entries of the table. The list of valid I/O functions and the list of buffered I/O functions have no associated action routine.

codes
The list of I/O function codes. The macro expansion prefixes each code specified with the string IO$_; for example, READVBLK expands to IO$_READVBLK.

7.3.4 Example of FUNCTAB Macro Use

In the example below, the routine (named XX_READ) called for a read function is a driver routine. It appears later in the driver module.

The routines EXE$SETMODE and EXE$SENSEMODE, preceded by plus signs (+) in the macro argument, are VAX/VMS routines that preprocess I/O requests for the device's set characteristics and sense mode functions.

```
XX_FUNCTABLE:                           ; Function decision table

        FUNCTAB ,-                      ; Valid functions
                <READLBLK,-             ;  Read logical block
                 READPBLK,-             ;  Read physical block
                 READVBLK,-             ;  Read virtual block
                 SENSEMODE,-            ;  Sense reader mode
                 SENSECHAR,-            ;  Sense reader characteristics
                 SETMODE,-              ;  Set reader mode
                 SETCHAR,-              ;  Set reader characteristics
                >
        FUNCTAB ,-                      ; Buffered I/O functions
                <READLBLK,-             ;   Read logical block
                 READPBLK,-             ;   Read physical block
                 READVBLK,-             ;   Read virtual block
                 SENSEMODE,-            ;   Sense reader mode
                 SENSECHAR,-            ;   Sense reader characteristics
                 SETMODE,-              ;   Set reader mode
                 SETCHAR,-              ;   Set reader characteristics
                >
        FUNCTAB XX_READ,-               ; Read functions
                <READLBLK,-             ;   Read logical block
                 READPBLK,-             ;   Read physical block
                 READVBLK,-             ;   Read virtual block
                >
        FUNCTAB +EXE$SETMODE,-          ; Set mode/characteristics
                <SETCHAR,-              ;   Set reader characteristics
                 SETMODE,-              ;   Set reader mode
                >
        FUNCTAB +EXE$SENSEMODE,-        ; Sense mode/characteristics
                <SENSECHAR,-            ;   Sense reader characteristics
                 SENSEMODE,-            ;   Sense reader mode
                >
```

CHAPTER 8

WRITING FDT ROUTINES

The Queue I/O Request system service uses the driver's function decision table to determine which FDT routines to call. These FDT routines validate user-specified arguments in the I/O request. VAX/VMS contains many device-independent FDT routines. Device drivers contain device-dependent FDT routines.

A driver should call the VAX/VMS device-independent FDT routines whenever possible. This practice encourages the use of well-debugged routines and minimizes driver size.

8.1 CONTEXT FOR FDT ROUTINE EXECUTION

The Queue I/O Request system service calls all FDT routines in the context of the process that requested the I/O operation. Characteristics of process context at the time of a call to an FDT routine are as follows:

- Virtual addresses are mapped according to the process page tables. This mapping allows FDT routines access to user-specified virtual addresses.
- The process is executing in kernel mode because the Queue I/O Request system service call executes a Change Mode to Kernel instruction.
- The process privileges remain unchanged.
- Interrupt priority level is set to IPL$_ASTDEL. Therefore, the process can be rescheduled but cannot receive ASTs. Paging can occur.
- FDT routines cannot call system services or VAX-11 RMS services.

8.2 REGISTERS PRESET FOR FDT ROUTINE EXECUTION

The Queue I/O Request system service also sets up a series of registers for the FDT routines before calling them. Table 8-1 lists the registers.

Table 8-1: Registers Loaded by Queue I/O Request Service

Register	Content
R0	Address of the FDT routine being called
R3	Address of the I/O request packet for the current I/O request
R4	Address of the process control block (PCB) of the current process
R5	Address of the unit control block of the device assigned to the user-specified process I/O channel
R6	Address of the channel control block that describes the user-specified process I/O channel
R7	Bit number of the user-specified I/O function code
R8	Address of the current entry in the function decision table
AP	Address of the first function-dependent parameter specified in the user's request

8.3 CONVENTIONS FOLLOWED BY FDT ROUTINES

Because FDT routines are called by the Queue I/O Request system service and return to it or, in turn, call another VAX/VMS routine, they must follow certain conventions to preserve register content and the expected process context.

8.3.1 Register Conventions

FDT routines are responsible for preserving the contents of R3 through R8 across subroutine calls. FDT routines can use R0 through R2 and R9 through R11 without saving their previous contents. If an FDT routine needs to use R3 through R8, the routine can use the push and pop register instructions to save registers on the stack and later restore them. The following is an example.

```
PUSHR   #^M<R3,R4,R5>           ; Save R3-R5 on the stack
        .
        .
        .
POPR    #^M<R3,R4,R5>           ; Restore R3-R5 from the stack
```

8.3.2 Process Context Conventions

The Queue I/O Request system service executes in the context of the process that issues the I/O request, but in kernel mode and at IPL$_ASTDEL. The Queue I/O Request system service expects FDT routines to preserve this context. Therefore, an FDT routine observes the following conventions:

- It does not lower IPL below IPL$_ASTDEL.
- If a routine raises IPL, it must lower IPL to IPL$_ASTDEL before exiting.
- It does not alter the stack without restoring its original state before exiting.
- It must observe the register conventions described in the previous section.
- It exits either by an RSB instruction to return control to the system service, or it issues a JMP instruction to one of the VAX/VMS routines described in Section 8.4.

8.4 TRANSFERRING INTO AND OUT OF AN FDT ROUTINE

To transfer control to an FDT routine, the Queue I/O Request system service loads the address of the FDT routine into a register and executes a jump to subroutine instruction, as follows:

```
JSB    (R0)
```

Each FDT routine chooses an exit path based on the following factors:

- Whether another FDT routine needs to be called 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

Figure 8-1 illustrates the FDT processing loop in the Queue I/O Request system service.

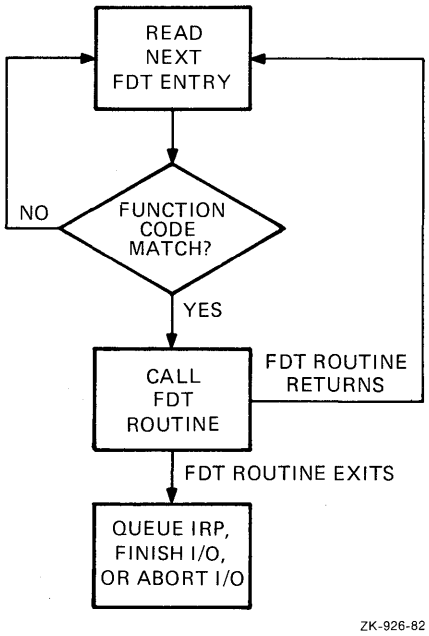


Figure 8-1: Queue I/O Request Scan of a Function Decision Table

As illustrated in Figure 8-1, the FDT routines are responsible for transferring control out of the FDT processing loop and into a VAX/VMS routine that queues an I/O request packet or completes an I/O request. The Queue I/O Request system service does not know when to stop scanning the function decision table. Therefore, you should ensure that all valid function codes in a driver's function decision table eventually call an FDT routine that does not return to the Queue I/O Request system service.

An FDT routine can exit using any of the methods summarized in Table 8-2. The first method returns to the Queue I/O Request system service. All other methods jump to VAX/VMS routines that take the appropriate action. See Section 8.8 for detailed descriptions of these routines.

Table 8-2: FDT Exit Methods

Exit Method	Result
RSB	Returns to the Queue I/O Request system service. The FDT routine returns to the system service because the routine knows that the function decision table contains a subsequent entry with the same function code bit set. As a result, the system service calls another FDT routine.
JMP G^EXE$QIODRVPKT or JSB G^EXE$ALTQUEPKT	Transfers control to a VAX/VMS routine that queues an I/O packet to a driver. The FDT routine uses this exit method if all preprocessing is complete, if no fatal errors are found in the specification of an I/O request, and if device activity is required to complete the I/O request. Once an FDT routine transfers control to either of these routines, no driver code that further processes the I/O request can refer to the process virtual address space. EXE$QIODRVPKT is the standard method used to queue an I/O request for device activity. This routine initiates driver action only if the device unit is currently idle; that is, there is no I/O request being processed. If the device unit is busy, EXE$QIODRVPKT queues the request to the unit so that the driver will process it when the unit becomes available. In contrast, EXE$ALTQUEPKT initiates driver action at a special driver entry point without regard for the device unit's activity status. This routine is called by drivers that can handle two or more I/O requests simultaneously.

(continued on next page)

Table 8-2 (Cont.): FDT Exit Methods

Exit Method	Result
JMP G^EXE$FINISHIO	Transfers control to a VAX/VMS routine that writes a quadword of final I/O status from R0 and R1 into the I/O status field of the I/O request packet (IRP$L_MEDIA and IRP$L_MEDIA+4). The routine then inserts the I/O request packet in the I/O postprocessing queue. An FDT routine that discovers a device-dependent error should always return status using EXE$FINISHIO or EXE$FINISHIOC. The routine returns to the Queue I/O Request system service the two longwords of status contained in the I/O status block (if any) specified in the Queue I/O Request.
JMP G^EXE$FINISHIOC	Transfers control to a routine that performs the same functions as EXE$FINISHIO except that this routine always clears the second longword of the final I/O status.
JMP G^EXE$ABORTIO	Transfers control to a VAX/VMS routine that aborts an I/O request. An FDT routine that discovers a device-independent error in an I/O request should always use this method of exit. The routine stores a longword of status in R0 and returns this to the system service. Inability to gain access to a data buffer is an example of a device-independent error.

8.5 FDT ROUTINES FOR DIRECT I/O

The VAX/VMS operating system provides two standard FDT routines that are applicable for direct I/O operations: EXE$READ and EXE$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$READ and EXE$WRITE are described in 8.7.

8.6 FDT ROUTINES FOR BUFFERED I/O

Device drivers for buffered I/O operations must contain their own device-specific FDT routines. An FDT routine for buffered I/O must perform the following steps:

- Confirm either read or write access to the user's buffer
- Allocate a buffer in system space

8.6.1 Checking the User's Buffer

First the FDT routine calls EXE$READCHK or EXE$WRITECHK to confirm write or read access, respectively, to the user's buffer. Both of these routines write the transfer byte count into IRP$W_BCNT. EXE$READCHK also sets IRP$V_FUNC in IRP$W_STS to indicate that the function is a read.

8.6.2 Allocating the System Buffer

Next, the FDT routine allocates a system buffer. First, it adds 12 bytes for a buffer header to the byte count passed in the P2 parameter of the user's I/O request. This is the total system buffer size. The FDT routine then calls EXE$BUFFRQUOTA to ensure that the user has sufficient remaining resources. If EXE$BUFFRQUOTA returns with a success code, the FDT routine calls EXE$ALLOCBUF which allocates the buffer and writes the buffer's size and type into its third longword.

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

- Loads the address of the system buffer into IRP$L_SVAPTE.
- Loads the total size of the system buffer into IRP$W_BOFF.
- Subtracts the system buffer size from JIB$L_BYTCNT. A longword in the PCB (PCB$L_JIB) points to the location of the job information block (JIB).
- Stores the starting address of the system buffer data area in the first longword of the buffer header.
- Stores the user's buffer address in the second longword of the header.
- Copies data from the user buffer to the system buffer if the I/O request is a write operation.

At this point, buffers are ready for the transfer. Figure 8-2 illustrates the format of the system buffer.

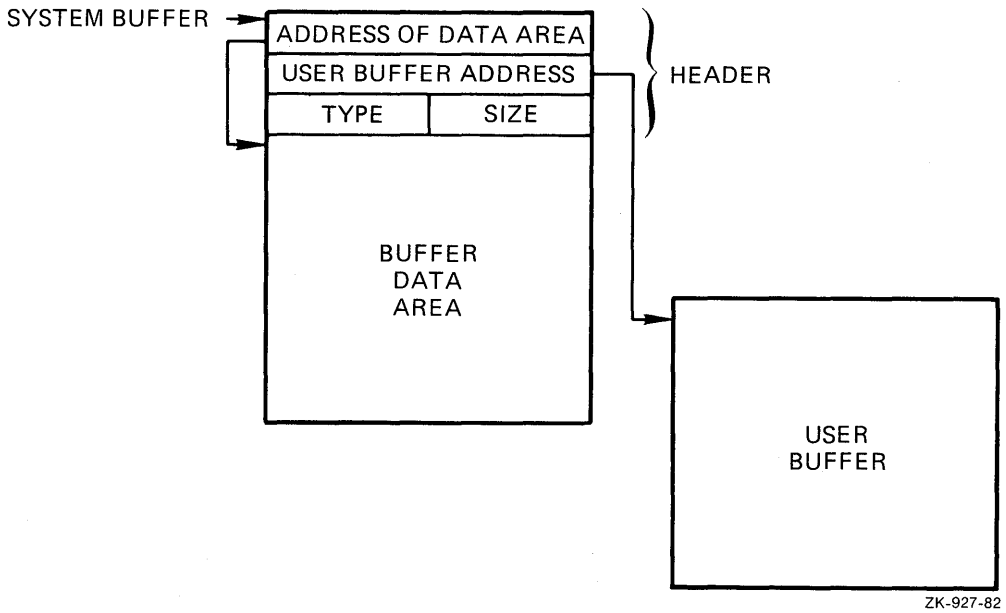


Figure 8-2: Format of System Buffer for Buffered I/O Read Operations

Appendix C provides additional information about EXE$READCHK, EXE$WRITECHK, EXE$BUFFRQUOTA, and EXE$ALLOCBUF.

8.6.3 Completion of Buffered I/O in I/O Postprocessing

When the transfer finishes, the driver returns control to VAX/VMS for completion of the I/O request. The driver writes the final count of bytes transferred into the high-order word of R0 and the final request status in the low order words of R0 and R1. The driver must leave the buffer header intact; I/O postprocessing relies on the header's accuracy. When VAX/VMS I/O postprocessing gains control, it performs the following steps:

- Adds the value in IRP$W_BOFF to JIB$L_BYTCNT to update the user's byte count quota
- If IRP$L_SVAPTE is nonzero, assumes a system buffer was allocated and checks to see whether IRP$V_FUNC is set in IRP$W_STS
- If IRP$V_FUNC is clear, deallocates the system buffer used for the write operation; if IRP$V_FUNC is set, the 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 kernel mode AST performs the following steps to complete a buffered read operation:

- Obtains the address of the system buffer from IRP$L_SVAPTE
- Obtains the number of bytes to write to the user's buffer from IRP$W_BCNT (for a read operation)
- Obtains the address of the user's buffer from the second longword of the system buffer header
- Checks for write accessibility on all pages of the user's buffer (for a read operation)
- Copies the data from the system buffer to the process's buffer (for a read operation)
- Deallocates the system buffer. Note that the system uses the size listed in the buffer's header to deallocate the buffer.

8.7 FDT ROUTINES PROVIDED BY VAX/VMS

The VAX/VMS FDT routines perform I/O request validation that is common to many devices. Whenever possible, drivers should take advantage of these routines. Normally, if a VAX/VMS FDT routine is called, no additional FDT processing is required. All of the VAX/VMS FDT routines described here exit by transferring control to one of the following VAX/VMS routines:

- EXE$QIODRVPKT
- EXE$FINISHIO
- EXE$FINISHIOC
- EXE$ABORTIO

Once a VAX/VMS FDT routine is called, no subsequent FDT processing occurs.

For information about additional FDT routines, see Appendix C.

8.7.1 EXE$ONEPARM

EXE$ONEPARM processes an I/O function code that has one parameter associated with it.

Exit Method

Queues the I/O request packet to the driver.

Description

Processes an I/O function code that requires only one parameter that needs no checking; for example, the parameter does not have to be checked for read or write accessibility. EXE$ONEPARM stores the parameter, found at 0(AP), in IRP$L_MEDIA of the I/O request packet. Then, it queues the I/O request packet to the driver.

8.7.2 EXE$READ

EXE$READ processes a logical or physical read function code for a direct I/O operation. EXE$READ cannot be used for buffered I/O operations.

Exit Method

Aborts the I/O request if an error occurs, or dismisses and resubmits the I/O request if the user I/O buffers cannot be locked in memory; otherwise, queues the I/O request packet to a driver.

Description

Sets the I/O function bit in the status field (IRP$V_FUNC in IRP$W_STS) of the I/O request packet. This bit indicates that the function is a read.

EXE$READ writes the fourth parameter, located at 12(AP) into the carriage control field (IRP$B_CARCON).

The routine replaces the logical function code IO$_READLBLK with the physical function code IO$_READPBLK in the function code field (IRP$W_FUNC) of the I/O request packet.

If the second parameter (the transfer byte count) is zero, EXE$READ queues the I/O request packet to a device driver. The second parameter is found at 4(AP). If the byte count is not zero, EXE$READ uses the starting address of the transfer, found at 0(AP), and the transfer byte count as arguments to the routine EXE$READLOCK.

The routine EXE$READLOCK calls EXE$READLOCKR, which immediately calls EXE$READCHKR. This last subroutine determines whether the caller's buffer permits write access.

If EXE$READCHKR finds that the buffer is accessible, it updates the I/O request packet by writing the size in bytes of the transfer to IRP$W_BCNT and setting the read status bit in IRP$W_STS (IRP$V_FUNC). The maximum number of bytes that EXE$READ can transfer is 65535 (128 pages minus one byte).

If the buffer does not allow write access, EXE$READCHKR returns access violation status to its caller, EXE$READLOCKR, which summons its caller (EXE$READLOCK) as a coroutine.

When EXE$READLOCK is called as a coroutine, it does not take any error action. Instead, it passes control to EXE$READLOCKR, which aborts the queue I/O request with access violation status. EXE$READLOCK is called as a coroutine for the convenience of drivers that call EXE$READLOCKR directly. See Appendix C for more details.

After EXE$READCHKR confirms the buffer's write accessibility, EXE$READLOCKR calls the routine MMG$IOLOCK to lock into memory those pages that contain the buffer. MMG$IOLOCK, can return success, page fault, or error status to EXE$READLOCKR.

If MMG$IOLOCK succeeds, EXE$READLOCKR stores the address of the process page table entry (PTE) in the field IRP$L_SVAPTE and returns success status to EXE$READLOCK.

However, if MMG$IOLOCK reports a page fault, EXE$READLOCKR adjusts direct I/O count and AST count to the values they held before the I/O request, deallocates the I/O request packet and restarts the request procedure at the Queue I/O Request system service. This procedure is carried out so that the user process can receive asynchronous system traps while it waits for the page fault to complete. Once the page is faulted into memory, the system service will resubmit the queue I/O request.

MMG$IOLOCK can report either of two errors: access violation (SS$_ACCVIO) and insufficient working set limit (SS$_INSFWSL). When EXE$READLOCKR receives an error, it aborts the request with error status.

After EXE$READLOCK returns to EXE$READ, the routine passes control to the exit routine EXE$QIODRVPKT so that the request is queued to the driver.

8.7.3 EXE$SENSEMODE

EXE$SENSEMODE processes the sense device mode and characteristics functions by reading fields of the unit control block. No device activity occurs.

Exit Method

Transfers control to EXE$FINISHIO.

Description

Loads the device-dependent characteristics field (UCB$L_DEVDEPEND) of the unit control block into R1. EXE$SENSEMODE then loads a normal completion status (SS$_NORMAL) into R0. Finally, it transfers control to EXE$FINISHIO to insert the I/O request packet in the I/O postprocessing queue.

8.7.4 EXE$SETCHAR

EXE$SETCHAR processes the set device mode and characteristics functions. If setting device characteristics requires no device activity or requires no synchronization with fork processing, the driver's FDT entry can specify EXE$SETCHAR; otherwise, it must specify EXE$SETMODE.

Exit Method

Aborts the I/O request on error; otherwise, transfers control to EXE$FINISHIO.

Description

Determines whether the process has read access to the quadword that describes the new characteristics for the device. The first parameter, found at 0(AP), specifies the address of the quadword. If the process does not have read access to the quadword, EXE$SETCHAR aborts the request.

If the process has read access, EXE$SETCHAR stores the new characteristics in fields of the device's unit control block. If the function is IO$_SETCHAR, the device type and class fields (UCB$B_DEVCLASS and UCB$B_DEVTYPE, respectively) of the unit control block receive the first word of data addressed by the parameter.

For both the IO$_SETCHAR and IO$_SETMODE functions, the routine writes the second word of data into the UCB default buffer size field (UCB$W_DEVBUFSIZ) and the third and fourth words of data into the device-dependent characteristics field (UCB$L_DEVDEPEND).

Finally, EXE$SETCHAR stores the normal completion status (SS$_NORMAL) in R0 and transfers control to EXE$FINISHIO to insert the I/O request packet in the I/O postprocessing queue.

8.7.5 EXE$SETMODE

EXE$SETMODE processes the set device mode and characteristics functions by activating the device.

Exit Method

Aborts the I/O request if an error occurs; otherwise, queues the I/O request packet to the device driver.

Description

Determines whether the process has read access to the quadword that describes the new characteristics for the device. The first parameter, found at 0(AP), specifies the address of the quadword. If the process does not have read access to the quadword, EXE$SETMODE aborts the request.

If the process has read access, EXE$SETMODE stores the new characteristics in the media field (IRP$L_MEDIA and IRP$L_MEDIA+4) of the I/O request packet. The routine then transfers control to the exit routine EXE$QIODRVPKT, which queues the request to the appropriate device driver.

8.7.6 EXE$WRITE

EXE$WRITE processes a logical or physical write function code for a direct I/O operation. EXE$WRITE cannot be used for buffered I/O operations.

Exit Method

Aborts the I/O request if an error occurs, or dismisses the I/O request if the user I/O buffers cannot be locked in memory; otherwise, queues the I/O request packet to a driver.

Description

Writes the fourth parameter, found at 12(AP) into the I/O request packet's carriage control field (IRP$B_CARCON).

EXE$WRITE replaces the logical function code IO$_WRITELBLK with the physical function code IO$_WRITEPBLK in the function code field of the I/O request packet (IRP$W_FUNC).

If the second parameter (the transfer byte count) is zero, EXE$WRITE queues the I/O request packet to the driver. The second parameter is found at 4(AP). If the byte count is not zero, EXE$WRITE uses the starting address of the transfer, found at 0(AP), and the transfer byte count as arguments to the routine EXE$WRITELOCK.

The routine EXE$WRITELOCK calls EXE$WRITELOCKR, which immediately calls EXE$WRITECHKR. This last subroutine determines whether the caller's buffer permits read access.

If EXE$WRITECHKR finds that the buffer is accessible, it updates the I/O request packet by writing the size in bytes of the transfer to IRP$W_BCNT. EXE$WRITE can transfer a maximum of 65535 bytes (128 pages minus one byte).

If the buffer does not allow read access, EXE$WRITECHKR returns access violation status to its caller, EXE$WRITELOCKR, which summons its caller (EXE$WRITELOCK) as a coroutine.

When EXE$WRITELOCK is called as a coroutine, it does not take any error action. Instead, it passes control to EXE$WRITELOCKR, which aborts the queue I/O request with access violation status. EXE$WRITELOCK is called as a coroutine for the convenience of drivers that call EXE$WRITELOCKR directly. See Appendix C for more details.

After EXE$WRITECHKR confirms the buffer's read accessibility, EXE$WRITELOCKR calls the routine MMG$IOLOCK to lock into memory those pages that contain the buffer. MMG$IOLOCK can return success, page fault, or error status to EXE$WRITELOCKR.

If MMG$IOLOCK succeeds, EXE$WRITELOCKR stores the address of the process page table entry (PTE) in IRP$L_SVAPTE and returns success status to EXE$WRITELOCK.

However, if MMG$IOLOCK reports a page fault, EXE$WRITELOCKR adjusts direct I/O count and AST count to the values they held before the I/O request packet and restarts the request procedure at the Queue I/O system service. The routine carries out this procedure 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 system service will resubmit the queue I/O request.

MMG$IOLOCK can report either of two errors: access violation (SS$_ACCVIO) and insufficient working set limit (SS$_INSFWSL). When EXE$WRITELOCKR receives an error, it aborts the request with error status.

After EXE$WRITELOCK returns to EXE$WRITE, the routine passes control to the exit routine EXE$QIODRVPKT so that the request is queued to the driver.

8.7.7 EXE$ZEROPARM

EXE$ZEROPARM processes an I/O function code that has no associated parameters.

Exit Method

Queues the I/O request packet to the driver.

Description

Processes an I/O function code that describes an I/O operation completely without any additional function-specific parameters. The only FDT processing necessary for a zero-parameter function code is to zero-fill the field of the I/O request packet that normally contains a user-specified parameter (IRP$L_MEDIA). Then EXE$ZEROPARM queues the I/O request packet to a device driver.

8.8 EXIT ROUTINES IN THE VAX/VMS SYSTEM

Ultimately, FDT processing must terminate by transferring control to one of the following VAX/VMS routines: EXE$ABORTIO, EXE$FINISHIO, EXE$FINISHIOC, EXE$ALTQUEPKT, or EXE$QIODRVPKT. Each of these routines returns the system service status code to the user.

8.8.1 EXE$ABORTIO

When an FDT routine determines that an I/O request cannot be completed because of an error in the specification of the request or in FDT processing, the FDT routine transfers control to the VAX/VMS routine EXE$ABORTIO to abort the request. EXE$ABORTIO gains control without any change in the process context. Interrupt priority level is at IPL$_ASTDEL; the process virtual space is mapped; and the process is executing in kernel mode.

Required Register Contents

R0	Queue I/O Request system service final status code
R3	Address of the current I/O request packet
R4	Address of the process control block of the current process
R5	Address of the unit control block of the device unit assigned to the process I/O channel

R3 through R5 always contain the I/O request packet, PCB, and UCB addresses at the entry to an FDT routine. The FDT routine should be careful not to destroy these values.

Description

EXE$ABORTIO clears the address of the I/O status block in the I/O request packet (IRP$L_IOSB) so that no status will be returned during I/O postprocessing. EXE$ABORTIO also clears the bit in the I/O request packet (ACB$V_QUOTA in the field IRP$B_RMOD). When set, this bit indicates that the requesting process specified an AST routine. If necessary, the routine readjusts the process's use of its AST quota.

Then EXE$ABORTIO inserts the I/O request packet in the I/O postprocessing queue. If no other entries are in the queue, EXE$ABORTIO requests a software interrupt at IPL$_IOPOST. This interrupt causes postprocessing to occur before any other instructions in the EXE$ABORTIO routine are executed.

When all I/O postprocessing has been completed, EXE$ABORTIO regains control and finishes the I/O operation as follows:

- Lowers IPL to zero, which is the normal IPL for a process
- Changes mode back to the original processor access mode
- Returns from the system service to the code of the image that originally requested the I/O operation. EXE$ABORTIO returns R0, which contains the final status code saved when the exit routine was called, to its caller.

  As a result of this exit method, any ASTs specified when the I/O request was issued will not be delivered, and any event flags requested will not be set.

8.8.2 EXE$FINISHIO and EXE$FINISHIOC

Many I/O requests need no device activity to be completed. The FDT routine(s) can complete the entire I/O request and immediately return status concerning the operation to the process. However, the VAX/VMS operating system provides two VAX/VMS I/O completion routines: EXE$FINISHIO and EXE$FINISHIOC. EXE$FINISHIO returns a quadword of I/O status. EXE$FINISHIOC returns a quadword of I/O status with the second longword containing zero.

These routines gain control without any change in process context. Interrupt priority level is at IPL$_ASTDEL; the process page tables are mapped; and the process is executing in kernel mode.

Required Register Content

R0	Value to be placed in the first longword of final I/O status when the Queue I/O Request system service returns final status
R1	Value to be placed in the second longword of final I/O status (EXE$FINISHIO only)
R3	Address of the current I/O request packet
R4	Address of the process control block of the current process
R5	Address of the unit control block of the device unit assigned to the process I/O channel

R3 through R5 always contain the I/O request packet, PCB, and UCB addresses at the entry to an FDT routine. The FDT routine should be careful not to destroy these values.

Description

EXE$FINISHIO and EXE$FINISHIOC modify fields in the I/O data base and then complete the I/O request in the following steps:

- Increase the number of I/O operations completed on the current device in the operation count field of the unit control block (UCB$L_OPCNT)
- Store the contents of R0 and R1 in the media fields of the I/O request packet (IRP$L_MEDIA and IRP$L_MEDIA+4)
- Insert the I/O request packet in the I/O postprocessing queue and, if the queue is empty, request a software interrupt at IPL$_IOPOST

EXE$FINISHIO and EXE$FINISHIOC lose control to I/O postprocessing because postprocessing executes at the higher IPL of IPL$_IOPOST. When EXE$FINISHIO and EXE$FINISHIOC regain control, they complete processing in the following steps:

- Lower IPL to zero, which is the normal IPL for a process
- Change mode back to the original processor access mode
- Return from the system service to the image that originally requested the I/O operation. The image receives status SS$_NORMAL in R0, indicating that the queue I/O request has completed without device-independent error.

8.8.3 EXE$QIODRVPKT

Some I/O functions require device activity, or at least access to device registers, for the I/O operation to be completed. Common examples are read and write functions. While FDT routines can perform extensive preprocessing, such as determining whether user buffers are accessible and reformatting data into buffers in the system address space, they should not access device registers because the device might be active. Furthermore, FDT routines should exercise restraint when modifying the unit control block. Routines usually access the UCB at driver fork IPL to synchronize modifications, and FDT routines do not operate at this interrupt priority level. Drivers containing FDT routines that access device registers or carelessly modify the unit control block risk unpredictable operation or a system failure.

For this type of I/O function, the associated FDT routines perform all preprocessing and then transfer control to the VAX/VMS routine EXE$QIODRVPKT. It queues the I/O packet to a device driver and attempts to transfer control to the device driver's start I/O routine. If the device unit is busy, EXE$QIODRVPKT inserts the I/O request packet in a priority-ordered queue of packets waiting for the unit.

Required Register Contents

R3	Address of the I/O request packet
R4	Address of the process control block of the current process
R5	Address of the unit control block for the device unit assigned to the process I/O channel

Description

EXE$QIODRVPKT calls EXE$INSIOQ, which first raises the interrupt priority level of the process to the fork level of the driver (UCB$B_FIPL). Driver fork level is, by convention, the interrupt priority level at which device drivers and VAX/VMS read and alter critical portions of the device's unit control block. By executing at fork level, EXE$INSIOQ ensures that, while it is running, a driver fork process for the device unit cannot also be running.

EXE$INSIOQ tests the UCB status word to see if the unit is busy.

If the device unit is not busy, EXE$INSIOQ calls the VAX/VMS routine IOC$INITIATE to create a fork process context in which the driver can process the I/O request. IOC$INITIATE creates this context and activates the driver in the following steps:

- Sets the busy bit of the device's unit control block (UCB$V_BSY in UCB$W_STS)
- Stores the address of the current I/O request packet in the UCB field UCB$L_IRP
- Copies the transfer parameters contained in the I/O request packet into the unit control block:
  - Copies the starting address from IRP$L_SVAPTE to UCB$L_SVAPTE
  - Copies the byte offset within the page from IRP$W_BOFF to UCB$W_BOFF
  - Copies the low order word of the byte count from IRP$W_BCNT to UCB$W_BCNT
- Clears the cancel I/O and timeout bits in the UCB status word (UCB$V_CANCEL and UCB$V_TIMOUT in UCB$W_STS)
- If the I/O request specifies a diagnostic buffer, as indicated by the bit IRP$V_DIAGBUF in IRP$W_STS, stores the system time in the buffer (IRP$L_DIAGBUF) (the Queue I/O Request system service having already allocated the buffer)
- Finds the entry point of the device driver's start I/O routine using the following chain of pointers:

  UCB ⟶ DDT ⟶ start I/O entry point

- Transfers control to the driver start I/O routine using a JMP instruction

If, on the other hand, EXE$INSIOQ finds that the device is busy, it inserts the I/O packet in the device unit's I/O request packet wait queue for processing later by calling EXE$INESRTIRP. The I/O request packet wait queue is ordered by two factors:

- The time that the entry is queued; that is, within any given priority the queue is first-in/first-out
- The priority of the I/O request packet, which is derived from the requesting process's base priority and stored in the field IRP$B_PRI

After completing one of the two operations described above, EXE$INSIOQ reduces the interrupt priority level to the level at the beginning of its execution; that is, to IPL$_ASTDEL. EXE$INSIOQ returns control to EXE$QIODRVPKT. Finally, EXE$QIODRVPKT returns from the Queue I/O Request system service in the following steps:

- Loads a success status code (SS$_NORMAL) into R0
- Reduces the interrupt priority level to 0
- Changes mode to the access mode of the process at the time of the I/O request by issuing an REI instruction
- Returns from the system service call

The system sets and clears the busy bit in the UCB status word for the device unit. This bit prevents the driver from being called to service a device unit that is already engaged in another I/O request.

When a device driver's start I/O routine gains control, the process that queued the I/O request may no longer be the mapped process. Therefore, the driver must assume that all information regarding the I/O request is in the unit control block or the I/O request packet and that all buffer addresses in the unit control block 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 since paging cannot occur at driver fork level and higher interrupt priority levels. The process virtual address space is not guaranteed to be mapped again until VAX/VMS delivers a kernel mode AST to the requesting process as part of I/O postprocessing.

8.8.4 EXE$ALTQUEPKT

Special purpose drivers may want to use their own internal I/O queues as well as the device unit I/O queue (UCB$L_IOQFL) provided by VAX/VMS. These internal queues allow the driver to handle I/O requests even if the device is busy with another I/O operation.

EXE$ALTQUEPKT permits the driver to ignore unit I/O queue synchronization. When called by an FDT routine, EXE$ALTQUEPKT gains access to the driver at the alternate start I/O entry point specified in the driver dispatch table (offset DDT$L_ALTSTART). This entry point bypasses the unit I/O queue and the device busy flag; thus, the driver is activated regardless of whether the device unit is busy.

A driver that uses EXE$ALTQUEPKT becomes responsible not only for its internal queues but also for any synchronization between those queues and the unit I/O queue maintained by the operating system.

Drivers complete I/O request packets obtained from EXE$ALTQUEPKT by calling the routine COM$POST. This routine places the I/O request packet in a postprocessing queue and returns control to the driver. The driver may then fetch another packet from an internal queue. For further information about COM$POST, see Appendix C.

If a driver processes more than one I/O request packet at the same time, separate fork blocks must be used.

Be aware that programming a device driver to process simultaneous I/O requests requires detailed knowledge of VAX/VMS internal design.

Required Register Contents

R3 Address of the I/O request packet
R5 Address of the unit control block

You must assume that the contents of R0 through R5 are destroyed upon return to the FDT routine.

Description

EXE$ALTQUEPKT performs the following steps:

- Saves the current interrupt priority level on the stack
- Raises interrupt priority level to driver fork level (UCB$B_FIPL)
- Finds the entry point of the alternate start I/O routine using the following chain of pointers:

  UCB ⟶ DDT ⟶ alternate start I/O address

- Calls the driver at alternate start I/O address

When the alternate start I/O routine finishes, it returns control to EXE$ALTQUEPKT by executing an RSB instruction. Unlike the other FDT exit routines, EXE$ALTQUEPKT is called with a JSB instruction rather than a JMP instruction. EXE$ALTQUEPKT restores interrupt priority level to that which existed when it was called, then returns control to the FDT routine that called it. The FDT routine performs any postprocessing and transfers control to the routine EXE$QIORETURN.

When EXE$QIORETURN gains control, it performs the following steps:

- Sets the success status code SS$_NORMAL in R0
- Lowers the interrupt priority level to zero
- Returns (with the RET instruction) to the system service dispatcher

CHAPTER 9

WRITING THE START I/O ROUTINE

A driver start I/O routine activates a device and then waits for a device interrupt or timeout. This chapter describes the start I/O routine. Chapter 12 describes the reactivation of the driver routine that performs device-dependent I/O postprocessing. With a few exceptions, the start I/O routine discussed in the following sections describes a DMA transfer using a single-unit controller.

9.1 TRANSFERRING CONTROL TO START I/O

The start I/O routine of a device driver gains control from either of two VAX/VMS routines: EXE$QIODRVPKT or IOC$REQCOM.

When FDT processing is complete for an I/O packet, the FDT routine transfers control to EXE$QIODRVPKT. If the designated device is idle, IOC$INITIATE is called to create a driver fork process. (This procedure is detailed in Section 8.8.3.) The driver fork process then gains control in the start I/O routine of the appropriate driver. If the device is busy, EXE$QIODRVPKT calls EXE$INSIOQ, which queues the packet to the device unit's I/O request packet wait queue.

After a device completes an I/O operation, the driver fork process exits by transferring control to IOC$REQCOM. IOC$REQCOM inserts the finished I/O packet in the postprocessing queue. It then dequeues the next I/O request packet from the device unit's I/O request packet wait queue and calls IOC$INITIATE to create a new driver fork process that gains control at the entry point of the driver's start I/O routine.

9.2 CONTEXT OF A DRIVER FORK PROCESS

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

System mapping	Only system page tables are mapped. Therefore, driver code cannot refer to virtual addresses in process address space.
Kernel mode	Execution occurs in the most privileged access mode and can, therefore, change IPL.
High IPL	The VAX/VMS routine that creates a driver fork process raises IPL to driver fork level before activating the driver. The driver can raise and lower IPL between driver fork level and IPL$_POWER.

Kernel or interrupt stack	Execution occurs on the kernel or interrupt stack. The driver must not alter the state of the stack without restoring it to its previous state before relinquishing control. The stack used depends upon whether the I/O startup is the result of a new I/O request or because a previous I/O request has completed. The choice of stacks must not affect start I/O routine operation.

In addition to the context described, the VAX/VMS packet queuing routines set up R3 and R5 for a driver start I/O routine, as follows:

- R3 contains the address of the I/O request packet.
- R5 contains the address of the unit control block for the device.

All registers must be preserved except for R0, R1, R2 and R4.

Before the packet queuing routines call the start I/O routine, they copy the following I/O request packet fields into their corresponding slots in the device's unit control block:

- IRP$W_BCNT ⟶ UCB$W_BCNT
- IRP$W_BOFF ⟶ UCB$W_BOFF
- IRP$L_SVAPTE ⟶ UCB$L_SVAPTE

9.3 ACTIVATING THE DEVICE

The processing performed by a start I/O routine is device-specific. A start I/O routine normally contains elements to perform the following functions:

- Analyze the I/O function
- Transfer the details of a transfer from the I/O request packet into the unit control block
- Obtain and initialize the controller and, for DMA transfers, UNIBUS adapter resources
- Modify device registers to activate the device

The start I/O routine elements listed above execute a series of steps to activate the device. The sections that follow describe those steps as performed for a sample DMA device such as a parallel communications link; the details of processing, however, are specific to the particular device. UNIBUS-related details of DMA transfers are described in Chapter 10.

9.3.1 Obtaining Controller Access

If the device is attached to a multiunit controller, the start I/O routine invokes the VAX/VMS macro REQPCHAN to assign the controller data channel to the device unit. Single-unit controllers do not require arbitration for the controller data channel. REQPCHAN calls the VAX/VMS routine IOC$REQPCHANL that acquires ownership of the controller data channel.

The transfer being controlled by the start I/O routine discussed here requires no seek preceding the transfer. Disk I/O is an example of a transfer that requires a seek first. To permit seeks to be overlapped with transfers, invoke REQPCHAN with the argument PRI=HIGH. Specifying PRI=HIGH inserts a request for a channel at the head of the channel wait queue.

If the channel is not available, IOC$REQPCHANL suspends driver processing by saving the driver's context in the UCB fork block and inserting the fork block address in the channel wait queue. IOC$REQPCHANL then returns control to the caller of the driver, that is, to IOC$INSIOQ, as illustrated in Figure 9-1.

The UCB fork block now represents the entire context of the suspended driver:

- Saved R3 containing the address of the I/O request packet
- Implicit saved R5 containing the UCB address
- A return address in the driver

IOC$REQPCHANL does not save R4 since it writes R4 before returning control to the driver.

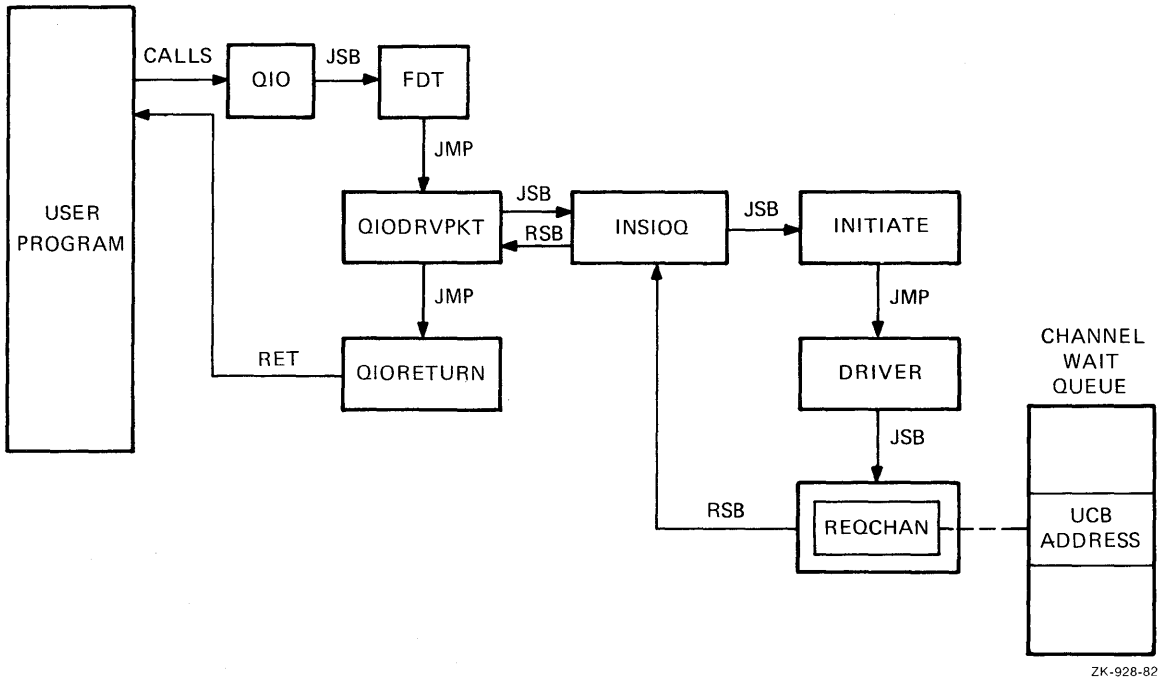


Figure 9-1: Driver Insertion into Channel Wait Queue

If the channel is available, IOC$REQPCHANL locates the interrupt dispatch block for the channel with a pointer in the unit control block:

UCB ⟶ CRB ⟶ IDB

The interrupt dispatch block contains the address of the control/status register for the channel (IDB$L_CSR). IOC$REQPCHANL returns the control/status register address in R4. The driver for a unit attached to a single-unit controller must contain the code needed to load the control/status address into R4.

IOC$REQPCHANL also writes the UCB address of the new channel owner in the owner field of the interrupt dispatch block (IDB$L_OWNER). The driver interrupt service routine later reads this IDB field to determine which device unit owns the controller data channel. A driver for a single-unit controller must fill the IDB$L_OWNER field in its controller or unit initialization routine.

The driver must maintain the stack in a known and consistent state for the resource wait queue mechanism to work. When IOC$REQPCHANL gains control, the top two items on the stack must be two return addresses:

- 0(SP) -- Address of the next instruction to be executed in the driver fork process
- 4(SP) -- Address of the next instruction to be executed in the routine that called the driver start I/O routine

9.3.2 Getting the I/O Function Code and Converting the Code and Modifiers

The start I/O routine extracts the I/O function code and function modifiers from the field IRP$W_FUNC and translates them into device-specific function codes to be loaded into the device's control/status register or other control registers. The I/O routine being described in this chapter sets up a bit mask that is to be modified further in subsequent instructions and loaded into the control/status register when the driver actually starts the device. That is, the start I/O routine converts the function modifiers contained in IRP$W_FUNC into device-specific bit settings in the general register.

At this point, the device driver follows procedures to obtain UNIBUS resources. These procedures are detailed in Chapter 10.

9.3.3 Computing the Transfer Length

Because the device driven by this particular driver expects the transfer as a word count, the start I/O routine computes the length of the transfer in words by dividing the byte count field of the unit control block (UCB$W_BCNT) by 2. The routine loads the computed value into the word count device register. One of the FDT routines that processes the I/O request must ensure that the byte count for the transfer is even. An odd byte count results in the user's not receiving the last byte of data.

9.3.4 Computing the Transfer Start Address

The start I/O routine calculates the address of the transfer using the byte offset field of the unit control block (UCB$W_BOFF) and the number of the starting map register (CRB$L_INTD+VEC$W_MAPREG). The result is an 18-bit value representing an address in UNIBUS address space. Section 10.4 details the calculation of the starting address for a UNIBUS transfer.

The start I/O routine stores the low-order 16 bits of the computed value in the buffer address device register. It stores the two high-order bits of the computed value in the memory extension bits of the bit mask (described in Section 9.3.2) to contain the device control/status register data.

9.3.5 Preparing the Device Activation Bit Mask

The start I/O routine prepares the device activation bit mask by setting the interrupt enable and go bits in the general register used previously. The general register contains a complete command to start the transfer at this point. When the start I/O routine copies the contents of the register into the device's control/status register, the device starts the transfer. However, before activating the device, the start I/O routine should perform the steps described in Sections 9.3.6 and 9.3.7.

9.3.6 Blocking All Interrupts

The start I/O routine invokes the VAX/VMS macro DSBINT to block all interrupts. DSBINT raises IPL to IPL$_POWER and saves the previous IPL setting on the top of the stack.

9.3.7 Checking for Power Failure

The start I/O routine examines the powerfail bits in the UCB status word (UCB$V_POWER in UCB$W_STS) to determine whether a 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 may 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$V_POWER is set, the start I/O routine branches to an error handler in the driver. The driver is responsible for clearing UCB$V_POWER before recovery or error procedures can be initiated. Many drivers clear this field and transfer to the beginning of the start I/O routine, which restarts processing of the I/O request.

9.3.8 Activating the Device

If no power failure has occurred, the start I/O routine copies the contents of the control mask into the device control/status register. When the device notices the new contents of the device register, the actual transfer begins.

9.4 WAITING FOR AN INTERRUPT OR TIMEOUT

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

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

Still executing at IPL$_POWER, the driver's start I/O routine asks VAX/VMS to suspend the driver fork process by invoking one of the following VAX/VMS 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 of these macros invoke routines that return IPL to the previous level when they exit. These routines expect to find the return IPL on the stack. This IPL is saved on the stack by the DSBINT macro as described in Section 9.3.7.

Drivers generally keep the controller data channel while waiting for the interrupt or timeout. Drivers for single unit controllers always keep the channel because there are no other units present that may need it. On multiunit controllers, some operations, such as disk seeks, do not require the controller once the operation has begun. In this case, the driver may want to release the controller data channel while waiting for interrupt or timeout so that other units on the controller can start their operations.

9.4.1 WFIKPCH and WFIRLCH Macro Formats

A start I/O routine invokes either the WFIKPCH or WFIRLCH macro to wait for device interrupt.

Formats

```
WFIKPCH     excpt,[time]

WFIRLCH     excpt,[time]
```

excpt
The address of the timeout routine for this device.

time
The number of seconds to wait before signaling a device timeout. The number must be greater than or equal to 2. A minimum value of 2 is required because the timeout mechanism is accurate only to within one second. If no number is specified, the macro uses the value 65536 by default.

9.4.2 Expansion of WFIKPCH Macro

Because the WFIKPCH and WFIRLCH macros are similar, the description that follows analyzes the expansion of WFIKPCH only.

If the driver specifies the time argument in the macro call, the macro pushes the value of the argument into the stack. If the time argument is not specified, the macro pushes the value 65536 onto the stack.

The VAX/VMS timer routine uses the time value to calculate the length of time to wait before transferring control to a device timeout handler.

WFIKPCH completes its expansion with the following two lines of code:

```
JSB     G^IOC$WFIKPCH
.WORD   EXCPT-.
```

The execution of the JSB instruction pushes the address following the JSB onto the stack as the address to which the called routine would normally return with an RSB instruction.

9.4.3 IOC$WFIKPCH Routine

The VAX/VMS routine IOC$WFIKPCH invoked by the macro WFIKPCH performs the functions necessary for the driver fork process to wait for a device interrupt or timeout. IOC$WFIKPCH first adds 2 to the address on the top of the stack so that the top of the stack contains the address of the next instruction in the driver after the macro invocation. This address is where the driver processing actually resumes as a result of an interrupt service routine JSB instruction.

IOC$WFIKPCH then saves the contents of R3, R4, and the driver return address from the top of the stack in the first part of the unit control block; that is, in the UCB fork block. The interrupt service routine must restore R5 to contain the address of the unit control block after an interrupt. The interrupt service routine normally obtains the address of the unit control block from the field IDB$L_OWNER of the interrupt dispatch block.

The VAX/VMS routine that detects a device timeout calculates the address of the driver timeout routine by subtracting 2 from the saved PC in the UCB fork block and calling indirectly through the result, for example:

```
MOVL    UCB$L_FPC(R5),R2        ; Get saved PC
CVTWL   -(R2),-(SP)             ; Get offset to timeout
                                ; handler
ADDL    (SP)+,R2                ; Add to relative driver
                                ; address to obtain relative
                                ; handler address
JSB     (R2)                    ; Call timeout handler
```

IOC$WFIKPCH sets bits in the unit control block (UCB$V_INT and UCB$V_TIM in UCB$W_STS) to indicate that interrupts and timeouts are expected from the device. IOC$WFIKPCH also writes the device timeout absolute time in the field UCB$L_DUETIM. The absolute time is the number of seconds since the operating system was bootstrapped plus the number of seconds specified in the time argument to the macro.

Finally, IOC$WFIKPCH reenables interrupts by lowering IPL to its previous level in the driver, that is, to driver fork level, and returns control to the caller of the driver.

9.5 RESPONDING TO AN EXPECTED DEVICE INTERRUPT

The only context saved for the driver is now in the unit control block. It contains the following information:

- A description of the I/O request and the state of the device
- The contents of R3 and R4
- The implicit contents of R5, that is, the address of the UCB fork block
- A driver return address
- The implicit address of a device timeout routine

By convention, R4 often contains the address of the control/status register; it permits the driver to examine device registers. When the driver fork process regains control after an interrupt processing, R5 contains the UCB address. It is the key to the I/O data base that is relevant to the current I/O operation.

When a device interrupts, the driver interrupt service routine analyzes the interrupt, as detailed in Chapter 11 and summarized below:

- Identifies the UCB address of the device that generated the interrupt
- Obtains device or controller status from the device registers, if necessary, and stores the status in the unit control block
- Restores the driver fork process registers from the UCB fork block, restores R5 with the UCB address, and reactivates the suspended driver at the PC stored in the UCB fork block

If, instead of requesting an interrupt, the device times out, a VAX/VMS timer routine reactivates the suspended driver fork process at the address of the timeout routine. Section 12.2 discusses device timeout handling in detail.

CHAPTER 10

WRITING UNIBUS DMA TRANSFERS

A driver performing DMA transfers over the UNIBUS must take UNIBUS operation into consideration. The VAX/VMS operating system and the I/O data base handle most UNIBUS map register and data path resource management for the device drivers. You must choose the type of data path (either direct or buffered) appropriate to the device and ensure that UCB fields are written to describe the virtual memory locations to be read or written. Once these actions have been taken, the driver fork process calls VAX/VMS routines to take care of the detailed operation of the UNIBUS adapter.

The I/O data base contains an adapter control block (ADP) that describes the UNIBUS adapter. This block contains allocation information for the UNIBUS adapter data paths and map registers.

The adapter control block also contains the virtual address of the UNIBUS adapter configuration register. All other adapter registers are located at fixed offsets from the configuration register. The VAX/VMS UNIBUS adapter-handling routines modify the UNIBUS adapter data path and map registers according to requests from driver fork processes.

In general, driver fork processes do not access the UNIBUS adapter control blocks. Instead, drivers call VAX/VMS routines that perform adapter-related services, such as the following:

- Allocate a buffered data path
- Allocate map registers
- Load map registers
- Deallocate map registers
- Purge a buffered data path
- Deallocate a buffered data path

The system creates a driver fork process by calling the start I/O routine in a device driver. The fork process takes some or all the following steps to initiate an I/O transfer on a UNIBUS device:

- Requests buffered data path
- Requests map registers
- Loads map registers
- Calculates starting UNIBUS address

- Activates device
- Waits for interrupt

When a hardware interrupt indicates that the I/O transfer is complete, the driver fork process checks the success or failure of the transfer. The driver then concludes with the following steps:

- Purges the buffered data path
- Releases the data path
- Releases the map registers

All of the steps above involve the UNIBUS adapter. VAX/VMS, however, hides most of the UNIBUS interfacing from the driver.

10.1 REQUESTING A BUFFERED DATA PATH

The system allows a driver to request temporary or permanent allocation of a buffered data path. After the driver fork process gains access to the controller (see Section 9.3.1), it requests a buffered data path by invoking the VAX/VMS macro REQDPR. REQDPR calls a VAX/VMS routine named IOC$REQDATAP that locates the UNIBUS adapter control block. To do this, IOC$REQDATAP uses a series of pointers that begin in the current unit control block, as follows:

UCB ⟶ CRB ⟶ ADP

The ADP data path allocation information indicates the buffered data paths that are available. IOC$REQDATAP allocates a data path to the driver by storing the data path number in the channel request block and indicating in the adapter control block (ADP) that the data path is in use. Then, control returns to the driver fork process. Appendix A describes the adapter control block.

If no data path is available, IOC$REQDATAP saves driver context (R3, R4, and PC) in the UCB fork block and inserts the address of the fork block, which is also the address of the unit control block and the content of R5, in the ADP data path wait queue. The driver fork block remains in the queue until both of the following conditions are met:

- A data path is available
- The driver fork block is the next entry in the data path wait queue

Then, the VAX/VMS routine IOC$RELDATAP allocates the data path to the suspended driver and reactivates the driver fork process.

10.1.1 Requesting a Permanent Buffered Data Path

A device driver can permanently allocate a buffered data path with code in a unit initialization routine. The following steps permanently allocate a buffered data path:

- Test the path lock bit (VEC$V_PATHLOCK) in the data path number field of the channel request block (CRB$L_INTD+VEC$B_DATAPATH) to ensure that a data path is not already allocated for this device.

- Call the subroutine IOC$REQDATAPNW to allocate the data path as shown below:

```
JSB  G^IOC$REQDATAPNW
```

  If IOC$REQDATAPNW successfully allocates the data path, it stores the number of the data path it obtained in the channel request block at VEC$B_DATAPATH and returns with the low-order bit set in R0. If IOC$REQDATAPNW cannot allocate a data path, it returns with the low-order bit clear in R0.

- Set the path lock bit (VEC$V_PATHLOCK) in the channel request block at VEC$B_DATAPATH

The driver loading procedure calls the unit initialization routine for each unit that the driver serves. A unit initialization routine that contains the code described above will permanently allocate one buffered data path for each CRB associated with the driver; that is, one path for each device controller that the driver serves.

Some VAX-11 processors have a small number of buffered data paths. If device drivers running on these processors do not limit permanent allocation of buffered data paths, the system may not have any paths left for its own use. For example, the VAX-11/750 has three buffered data paths. If device drivers loaded on this machine permanently allocate all three data paths, the operating system will have no buffered data paths left for normal operations. In this case, I/O transfers that require a buffered data path will wait forever.

10.1.2 Requesting the Direct Data Path

Because the UNIBUS adapter arbitrates among devices that wish to use the direct data path and because the CRB is initialized to 0 (0 = direct data path), drivers are not required to invoke the REQDPR macro to request the direct data path.

Some VAX-11 processors, such as the VAX-11/780, do not permit byte-offset transfers on the direct data path. On these processors, drivers for word-aligned devices must ensure that the data buffer is word-aligned.

10.1.3 Mixed Direct and Buffered Data Path Transfers

A device driver can use the buffered data path for certain operations, then use the direct data path for other operations. To accomplish this task, the driver should allocate a buffered data path for buffered I/O. When the operation completes, the driver should then purge and release the data path. The release automatically resets the data path number to zero, which signifies a direct data path. However, the driver should not release the direct data path, although it should purge the path. (A purge of the direct data path is a NOP and always yields success.)

10.2 REQUESTING UNIBUS ADAPTER MAP REGISTERS

The operating system allows a driver to allocate map registers as needed or to allocate them permanently.

10.2.1 Allocation of Map Registers

After the driver fork process gains access to the controller (see Section 9.3.1), it requests a set of UNIBUS adapter map registers by invoking the VAX/VMS macro REQMPR. This macro calls the routine IOC$REQMAPREG. IOC$REQMAPREG calculates the number of map registers needed for a transfer. The calculation is based on the transfer byte count field and the byte offset fields of the device's unit control block (UCB$W_BCNT and UCB$W_BOFF).

The procedure for allocating map registers is similar to that used to allocate a buffered data path. First, IOC$REQMAPREG locates the adapter control block from a series of pointers that begin with the current unit control block, as follows:

UCB ⟶ CRB ⟶ ADP

Then, the routine examines the map register allocation information to locate the required number of contiguous map registers. If the registers are not currently available, IOC$REQMAPREG saves the driver context (R3, R4, and PC) in the UCB fork block and inserts the fork block address (same as UCB address and the contents of R5) in the map register wait queue.

When the map registers are available, IOC$REQMAPREG allocates them and adjusts the appropriate map register allocation information in the adapter control block. IOC$REQMAPREG then writes the number of the starting map register and the number of map registers allocated into the channel request block and returns control to the driver fork process.

10.2.2 Permanent Allocation of Map Registers

A device driver can permanently allocate a set of map registers with code in the unit initialization routine. The number of map registers permanently allocated must be sufficient for the longest possible transfer. The following steps permanently allocate a set of map registers:

- Test the map lock bit (VEC$V_MAPLOCK) in the channel request block (CRB$L_INTD+VEC$W_MAPREG).
- Load the number of map registers required into R3.
- Call the VAX/VMS routine IOC$ALOUBAMAPN with a JSB instruction:

  ```
  JSB   G^IOC$ALOUBAMAPN
  ```

  If IOC$ALOUBAMAPN successfully allocates the map registers, it stores the number of map registers allocated and the starting map register's number in the channel request block at CRB$L_INTD+VEC$B_NUMREG and CRB$L_INTD+VEC$W_MAPREG, respectively, and returns with the low-order bit set in R0.

  Otherwise, it returns with the low-order bit of R0 clear.

- Set the map lock bit in the channel request block (VEC$V_MAPLOCK in CRB$L_INTD+VEC$W_MAPREG).

The driver loading procedure calls the unit initialization routine once for each unit associated with the driver. If the unit initialization routines contains the code described above, it

permanently allocates one set of map registers for each CRB associated with the driver; that is, one set of registers for each device controller that the driver serves.

10.3 LOADING THE UNIBUS ADAPTER MAP REGISTERS

Once a driver fork process has assigned a data path and allocated a set of map registers, it can request VAX/VMS to load the map registers with physical page frame numbers by invoking the VAX/VMS macro LOADUBA. LOADUBA calls a VAX/VMS routine IOC$LOADUBAMAP that loads each allocated map register with five data items:

- A bit setting to indicate whether the map register is valid.
- A bit setting to indicate whether the transfer is to start on the odd or even byte within a word; this bit is set if the low-order bit of UCB$W_BOFF is a 1.
- The number of the data path to use for the transfer.
- The page frame number of a page in memory.
- A bit setting to indicate that the transfer operates in longword-aligned random access mode; This bit is set when VEC$V_LWAE is set in VEC$B_DATAPATH.

IOC$LOADUBAMAP loads the page frame number of the first page of the transfer into the first allocated map register, the page frame number of the second page of the transfer into the second map register, and so forth.

IOC$LOADUBAMAP sets the valid bit in every allocated map register except the last. It clears the valid bit in the final map register to stop a prefetch from an invalid page frame number.

To calculate the page frame number used in the I/O transfer, IOC$LOADUBAMAP uses three fields that VAX/VMS has written into the unit control block:

- UCB$W_BOFF -- byte offset in the first page of the transfer
- UCB$W_BCNT -- number of bytes to transfer
- UCB$L_SVAPTE -- virtual address of the page table entry that contains the page frame number of the first page of the transfer

IOC$LOADUBAMAP determines the data path number, the number of the first map register, the address of the first map register, and the number of map registers from the channel request block and the UNIBUS adapter control block, as follows:

UCB ⟶ CRB ⟶ data path number

UCB ⟶ CRB ⟶ number of first map register

UCB ⟶ CRB ⟶ ADP ⟶ virtual address of first map register

UCB ⟶ CRB ⟶ number of map registers

Drivers that handle byte-addressable UNIBUS devices call the routine IOC$LOADUBAMAPA. This routine performs the same function as

IOC$LOADUBAMAP, with one exception. When IOC$LOADUBAMAPA loads map registers, it clears the byte offset bit even if the transfer begins on an odd-byte address.

When IOC$LOADUBAMAP has loaded all the map registers and marked the last map register invalid, it returns control to the driver fork process.

10.4 COMPUTING THE STARTING ADDRESS OF A TRANSFER

The driver fork process must calculate the starting address of a UNIBUS transfer and load this address into the appropriate device register. The driver takes the following steps to make the calculation:

- Writes the byte-offset-in-page field of the UCB (UCB$W_BOFF) into bits 0 through 8 of a register
- Gets the number of the starting map register for the transfer from the channel request block; the number is a 9-bit value
- Writes bits 0 through 6 of the map register number into bits 9 through 15 of the register containing the byte offset field
- Writes bits 0 through 15 of the register into the buffer address register for the device
- Writes bits 7 and 8 of the map register number into the extended memory bits of the appropriate device register (usually the control/status register)

10.5 ACTIVATING THE DEVICE

Because a driver fork process can address device registers as though they were any other virtual address, the loading of the UNIBUS buffer address register and control/status register both are simple procedures. The driver locates the CSR address of the device in the interrupt data block, as follows:

UCB ⟶ CRB ⟶ IDB ⟶ CSR address

The CSR address is the virtual address of a device register. All other device registers are located at constant offsets from the CSR address. If, for example, the control/status register is the first device register and the device word count is the third device register, the device driver can load the word count register with the following sequence of instructions:

- Move the CSR address into R4.
- Move the number of words to transfer with a MOVW instruction that addresses 4(R4).

10.6 COMPLETION OF A DMA TRANSFER

After a driver fork process activates a DMA UNIBUS device, the driver waits for a device interrupt by invoking a VAX/VMS macro that suspends the driver. When the UNIBUS device requests a hardware interrupt, the interrupt dispatcher gains control. The dispatcher saves R0 through

R5 and transfers control to the driver interrupt service routine. If the service routine can match the interrupt with a suspended driver fork process, the interrupt service routine reactivates the driver fork process at the point that execution was suspended. The driver almost immediately invokes the VAX/VMS macro IOFORK.

IOFORK calls the VAX/VMS routine EXE$IOFORK. EXE$IOFORK saves the driver context (R3, R4, and PC) in the UCB fork block and inserts the address of the fork block (R5) in the device's fork queue. EXE$IOFORK then returns control to the driver's interrupt service routine, which dismisses the interrupt.

When the fork dispatcher reactivates the driver fork process, the driver performs any necessary UNIBUS adapter clean-up operations, such as data path purging and deallocation of UNIBUS adapter resources used in the DMA transfer.

10.6.1 Purging the Data Path

Driver fork processes that use buffered data paths must purge the data path after the DMA transfer is complete. The driver invokes the macro PURDPR, which in turn calls the VAX/VMS routine IOC$PURGEDATAP. This routine takes the following steps to purge the data path:

- Saves the contents of R4 on the stack.
- Locates the channel request block as follows:

  R5 ⟶ UCB ⟶ CRB

- Obtains the starting address of UNIBUS adapter register space and stores it in R2.
- Extracts the number of the data path to be purged from the channel request block and loads it into R1.
- Stores the address of the data path in R4.
- Instructs the UNIBUS adapter to purge the data path. The routine then modifies R0 through R2 to contain the following information:

  R0 Success/failure status. If the purge completes without error, the routine sets SS$_NORMAL in this register. If a data path error does occur, R0 is clear and the hardware is reset.

  R1 Contents of the data path register.

  R2 Address of the first UNIBUS adapter map register.

  The address of the channel request block remains in R3. This address, along with the information in R1 and R2, is used as input to the error-logging routine in the event of a data path error.

- Restores the information stored on the stack to R4 and returns to PURDPR.

If a data path error occurs during a data path purge, the driver should retry the entire DMA transfer.

10.6.2 Releasing a Buffered Data Path

A driver fork process releases a buffered data path by invoking the VAX/VMS macro RELDPR. RELDPR calls a VAX/VMS routine IOC$RELDATAP that determines which data path was assigned to the driver fork process and releases the data path to a waiting driver. The driver must be executing at fork IPL.

The data path number is stored in the channel request block. IOC$RELDATAP locates it as follows:

UCB ⟶ CRB ⟶ data path number

If the data path is permanently assigned to a device, IOC$RELDATAP does not release the data path. Otherwise, the data path number in the channel request block (CRB$L_INTD + VEC$B_DATAP) is zeroed. The IOC$RELDATAP routine attempts to dequeue a waiting driver fork process from the data path wait queue stored in the adapter control block as follows:

UCB ⟶ CRB ⟶ ADP ⟶ data path wait queue

If another driver is waiting for a buffered data path, IOC$RELDATAP grants that driver fork process the data path, restores its driver context from its UCB fork block, and transfers control to the saved driver PC. When IOC$RELDATAP can allocate no more data paths, the routine returns to the driver that released the data path. This diversion of driver processing is transparent to the driver fork process.

If the data path wait queue is empty, IOC$RELDATAP marks the data path as available in the adapter control block and returns control to the driver.

10.7 RELEASING UNIBUS ADAPTER MAP REGISTERS

A driver fork process releases a set of UNIBUS adapter map registers by invoking the VAX/VMS macro RELMPR. RELMPR calls the VAX/VMS routine IOC$RELMAPREG that releases map registers in a manner similar to that in which data paths are released. The channel request block records the map register numbers assigned to the device. The number of the first map register and the number of map registers are located as follows. The driver must be executing at fork IPL.

UCB ⟶ CRB ⟶ number of the first map register

UCB ⟶ CRB ⟶ number of map registers allocated

IOC$RELMAPREG releases the map registers by adjusting the map register allocation information in the adapter control block.

Then, IOC$RELMAPREG attempts to dequeue a driver fork process from the map register wait queue. If a suspended driver is found, IOC$RELMAPREG takes the following steps:

- Dequeues the fork block and restores driver context
- Fills the map register request, if possible
- Reactivates the driver fork process at the instruction following the driver's request for map registers
- Returns to the driver fork process

If the map register wait queue is empty or if IOC$RELMAPREG still does not have enough contiguous map registers for any of the waiting fork processes, it returns control to the driver fork process that released the map registers.

CHAPTER 11

WRITING INTERRUPT SERVICE ROUTINES

The driver prologue table of most device drivers contains, in the reinitialization section established using the DPT_STORE macro, the address of one or more interrupt service routines. Each interrupt service routine corresponds to an interrupt vector on the UNIBUS. You specify the UNIBUS vector address using the SYSGEN command CONNECT, as described in Chapter 14.

Most interrupt service routines in device drivers perform the following functions:

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

Figure 11-1 illustrates the general flow of interrupt handling. The remaining sections of this chapter describe the handling of solicited and unsolicited interrupts in further detail.

11.1 DELIVERING A DEVICE INTERRUPT TO A DRIVER

When a UNIBUS device generates a hardware interrupt, the device requests the interrupt at its device IPL. The UNIBUS adapter then requests a processor interrupt at that device IPL. When the processor executes at an interrupt priority level below the device IPL, interrupt dispatching begins.

On a configuration that uses nondirect vector interrupts, the following sequence occurs:

- The processor saves the PC and PSL of the currently executing code on the interrupt stack and transfers control to the VAX/VMS UNIBUS adapter interrupt service routine.
- The UNIBUS adapter interrupt service routine reads the vector register within the UNIBUS adapter that corresponds to the interrupt level of the device. The UNIBUS adapter acknowledges the interrupt and the interrupting device supplies its vector address to the UNIBUS adapter interrupt service routine.
- The UNIBUS adapter service routine then saves R0 through R5 on the stack and, using a JMP instruction, transfers control to an interrupt dispatching field within the channel request block.

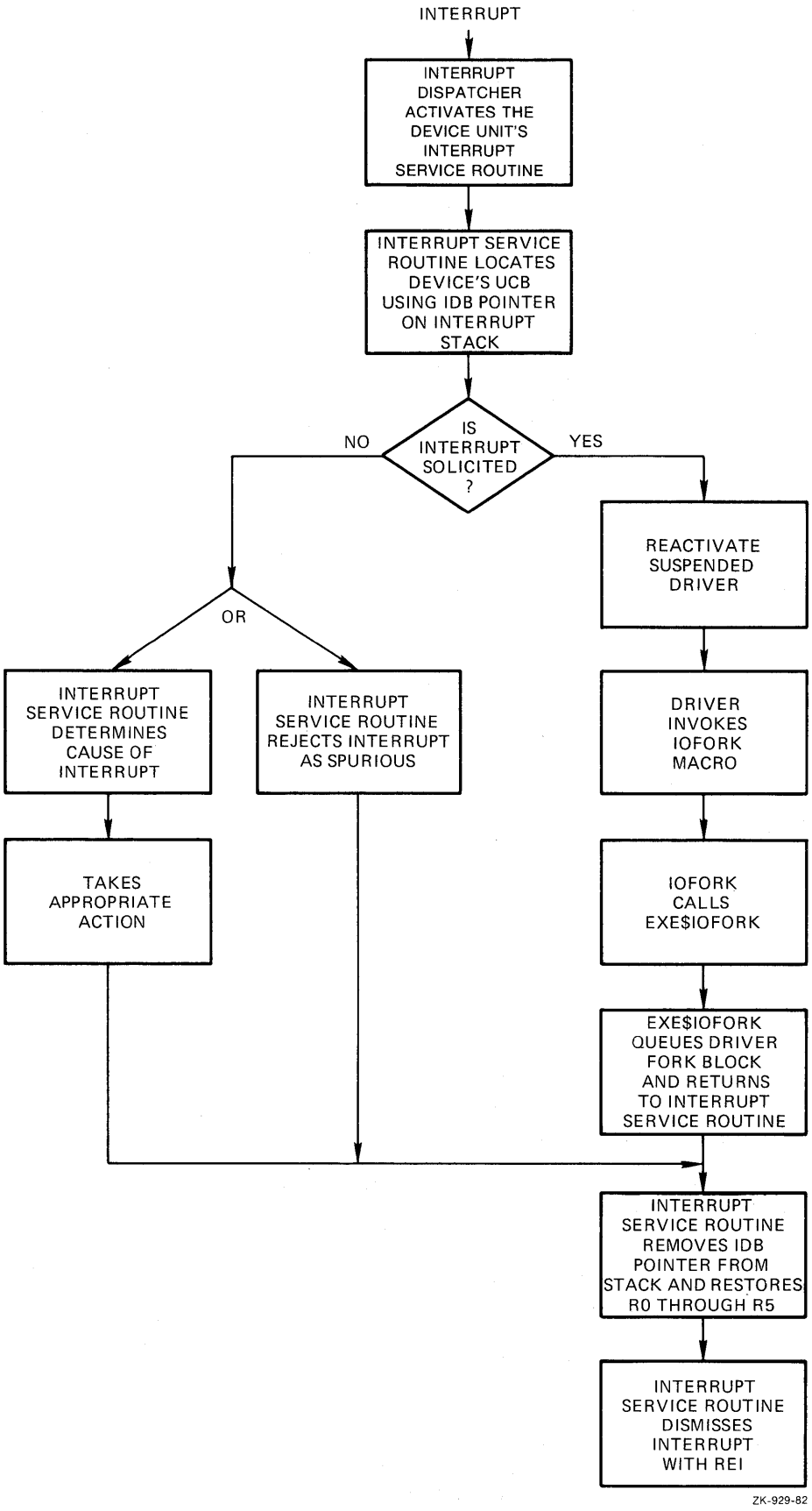


Figure 11-1: Interrupt Handling Flow

- The CRB interrupt dispatching field (CRB$L_INTD+2) contains executable code that the driver loading procedure has associated with the interrupting vector. Interrupt dispatching fields for nondirect vectors contain the following executable instruction:

```
JSB @#address-of-driver-isr
```

On a configuration that uses direct vector interrupts, the following sequence occurs:

- The processor saves the PC and PSL of the currently executing code on the interrupt stack and acknowledges the device interrupt.
- The UNIBUS device supplies its vector address, which the processor uses as an index into a table of addresses in the second (or third) page of the system control block (see Section 3.1.6).
- When the processor locates the address in the SCB that corresponds to the vector address, it transfers control to an interrupt dispatching field in the channel request block.
- The CRB interrupt dispatching field (CRB$L_INTD) contains executable code that the driver loading procedure has associated with the interrupting vector. Interrupt dispatching fields of direct vectors contain the following executable instructions:

```
PUSHR <R0,R1,R2,R3,R4,R5>

JSB   @#address-of-driver-isr
```

The driver loading procedure determines how many interrupt dispatching fields to build within the CRB from the number of vectors specified in the /NUMVEC qualifier to the SYSGEN command CONNECT (see Section 14.2.2). The driver loading procedure obtains the interrupt service routine address for each interrupt dispatching field from the reinitialization portion of the driver prologue table. This section of the DPT contains one or more DPT_STORE macros that identify the interrupt service routine addresses. The number of DPT_STORE macros that identify interrupt service routines must equal the number of vectors given in the /NUMVEC qualifier to avoid errors in device initialization or interrupt handling.

Immediately following the JSB instruction in the channel request block is the address of the interrupt dispatch block associated with the CRB. When the JSB instruction executes, a pointer to the address of the interrupt dispatch block is pushed onto the top of the stack as though it were a return address. The driver interrupt service routine can use this IDB address as a pointer into the I/O data base. Figure 11-2 illustrates the portion of a channel request block that contains the interrupt service routine address.

11.2 INTERRUPT CONTEXT

When the interrupt dispatcher calls a driver interrupt service routine, execution context is as follows:

- R0 through R5 are saved on the stack.
- System address space is mapped. The service routine can gain access to appropriate control blocks in the I/O data base.

- IPL is at hardware device interrupt level.
- The processor is running in kernel mode.
- The processor is running on the interrupt stack.

The stack contains the following information:

Stack Location	Content
0(SP)	Pointer to the address of the interrupt dispatch block
4(SP) through 24(SP)	Saved R0 through R5
28(SP)	PC at the time of the interrupt
32(SP)	PSL at the time of the interrupt

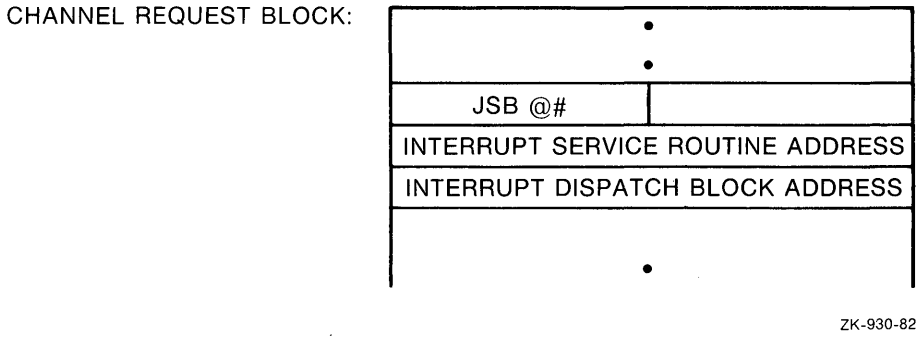


Figure 11-2: Channel Request Block Containing an Interrupt Service Routine Address

11.3 SERVICING A SOLICITED INTERRUPT

When a driver fork process activates a device and expects to service a device interrupt as a result, the driver suspends fork processing and waits for an interrupt to occur. The suspended driver is represented only by the contents of the device's unit control block, which contains a description of the I/O request and the driver fork process. When the driver regains control from the interrupt service routine, only R3, R4, R5, and the PC address are restored to their previous state by the interrupt service routine.

In the sequence below, a driver interrupt service routine returns control to the waiting driver:

- First, the interrupt service routine obtains the address of the device's unit control block from the interrupt data block, as follows:

  0(SP) ⟶ CRB ⟶ IDB ⟶ IDB$L_OWNER ⟶ UCB for the device

- The service routine then tests the software interrupt expected bit in the UCB status word (UCB$V_INT in UCB$W_STS). If the bit is set, the driver is waiting for an interrupt from this device. The interrupt service routine then clears UCB$V_INT in UCB$W_STS to indicate that it has received the expected interrupt.

- The interrupt service routine restores R5 of the driver fork process with the address of the UCB fork block. It restores R3 and R4 of the driver process using two fields from the UCB fork block, UCB$L_FR3 and UCB$L_FR4, respectively.

- Finally, the interrupt service routine transfers control to the driver PC address saved in the UCB fork block at UCB$L_FPC by issuing a JSB instruction.

The restored driver can execute a few instructions in the context of the interrupt, such as copying device status information from the device registers into the device's UCB. Before completing the I/O operation, however, the driver routine creates a fork process to lower its execution IPL to driver fork level instead of continuing execution at hardware device interrupt IPL. The driver routine creates a fork process by invoking the VAX/VMS macro IOFORK, as described in Section 12.1.1.

IOFORK calls the VAX/VMS routine EXE$IOFORK. EXE$IOFORK inserts the UCB fork block describing the driver process in the appropriate fork queue and returns control to the driver interrupt service routine. The interrupt service routine then performs the following steps:

- Removes the IDB pointer from the stack

- Restores R0 through R5

- Dismisses the interrupt with an REI instruction

11.4 SERVICING AN UNSOLICITED INTERRUPT

Devices request interrupts to indicate to a driver interrupt service routine that the device has changed status. If a driver 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 of device status occur when the device has not been activated by a device driver. The device reports these changes by requesting unsolicited interrupts. For example, when a user types on a terminal that is not attached to a process, the terminal requests an interrupt that is fielded by the terminal driver. As a result of the interrupt, the terminal driver causes the login procedure to be invoked for the user at the terminal.

Another example of an unsolicited interrupt is one that the unit requests when an operator changes the volume on a disk drive. The disk driver services the interrupt by altering volume and unit status bits in the disk device's unit control block.

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

The driver interrupt service routine decides whether an interrupt is solicited or not by examining the software interrupt expected bit in the UCB status word (UCB$V_INT in UCB$W_STS). All UNIBUS device

drivers must use this method to determine whether or not an interrupt is solicited; the unsolicited interrupt routine address specified in the driver dispatch table is used only by MASSBUS drivers.

If the interrupt is unsolicited, the driver can reject the interrupt with the following code sequence:

- Remove the IDB pointer from the stack
- Restore R0 through R5
- Dismiss the interrupt with an REI instruction

Rather than rejecting the interrupt, the driver may wish to handle it. For example, the driver can send a message to the operator or the job controller mailbox when an unsolicited interrupt occurs.

Drivers should use extreme caution when creating a fork process to handle unsolicited interrupts from busy devices. The unit control block of a busy device may contain the active fork block of a previously created driver fork process. If an unsolicited interrupt service routine should create a fork process to handle its request, it may destroy the driver fork context currently stored there. Drivers should always handle this type of unsolicited interrupt at hardware device IPL.

11.4.1 Examples of Unsolicited Input Handling

A card reader device requests an unsolicited interrupt if any user turns the reader online. Once the card reader driver interrupt service routine determines that the interrupt is unsolicited, the routine analyzes the interrupt, as in the following example:

- It obtains the address of the control/status register using the interrupt dispatch block pointed to by the address on the top of the interrupt stack, as follows:

  0(SP) ⟶ CRB ⟶ IDB ⟶ IDB$L_CSR ⟶ CSR for the device

  Since the card reader controller is a single-unit controller, the IDB$L_OWNER field always points to the single UCB for the card reader:

  0(SP) ⟶ CRB ⟶ IDB ⟶ IDB$L_OWNER ⟶ UCB for the device

- It confirms that the interrupt is unsolicited by testing the interrupt enable bit in the UCB status word (UCB$V_INT in UCB$W_STS).
- Since the interrupt is unsolicited, the routine clears all control/ status register bits except for the interrupt enable bit.
- It confirms that the reader was just placed online by examining a saved copy of the control/status register.
- It examines the reference count field of the device's unit control block (UCB$W_REFC) to determine whether a process has allocated the device or assigned a channel to it.
- If the reference count is zero, the interrupt service routine tests the job-attached bit in the device-dependent status field (UCB$V_JOB in UCB$W_DEVSTS) to make sure it has not

already sent the job controller a message about the card reader being placed online. By using the job-attached bit to synchronize message sending, the interrupt service routine protects the send-message-to-job-controller function from the adverse effects of frequent online interrupts.

- If the job-attached bit is not set, the routine sets the bit and creates a fork process to send the message to the job controller saying the reader has come online. Only one sequence of instructions can use the UCB as a fork block. Therefore, the interrupt service must perform the following steps before it can create the fork process:

  -- Ensure that no one is using the card reader and that no one desires to use it by determining that the reference count (UCB$W_REFCNT) is zero.

  -- Ensure that it is not already using the UCB to fork to a lower IPL and send a message to the job controller by testing the job-attached bit (UCB$V_JOB in UCB$W_DEVSTS).

  The VAX/VMS routine that creates the fork process (once the above conditions are satisfied) returns control to the interrupt service routine.

- When the interrupt service routine regains control, it restores R0 through R5 and dismisses the interrupt with an REI instruction. (The interrupt service routine removed the IDB pointer from the stack earlier in its execution in order to obtain CSR and UCB addresses.) If a fork process was created, it executes after IPL drops below UCB$B_FIPL. The fork process writes the message about the card reader's coming online to the job controller's mailbox. The fork process cannot send the message at device IPL or any IPL greater than IPL$_MAILBOX.

  If the send message request fails, the fork process clears the job-attached bit so that the job controller will receive a message if any change in the card reader's state occurs. If the fork process successfully sends the message, it leaves the job-attached bit set to prevent the job controller from receiving any further messages about the card reader's state. (The driver cancel I/O routine later clears the bit.)

Another example of unsolicited interrupt processing occurs in a device driver for a multiunit controller. When the operator removes a disk volume, the disk drive requests an interrupt. The driver interrupt service routine must determine what drive unit requested the interrupt, obtain drive status from the drive's control/status register, and then decide whether the interrupt was solicited. If the interrupt is unsolicited, the driver service routine calls its unsolicited interrupt routine. The routine checks the status of the volume, as described in the following steps:

- It sets a bit in the unit control block to indicate that the unit is online (UCB$V_ONLINE in UCB$W_STS).

- If the UCB volume valid bit is set (UCB$V_VALID in UCB$W_STS), the routine tests the volume valid status bit in a device register to determine whether the volume status has changed. If the volume is no longer valid, the routine clears the UCB volume valid bit.

- Finally, the routine returns to the normal driver interrupt service routine.

The driver interrupt service routine then polls the other device units on the controller to determine whether any other units requested interrupts while the first interrupt was being processed. When no unit requires interrupt servicing, the routine removes the IDB pointer from the stack, restores registers R0 through R5, and dismisses the interrupt with an REI instruction.

CHAPTER 12

COMPLETING THE I/O REQUEST

Once a driver has activated the device and invoked the wait for interrupt macro, the driver remains suspended until one of the following events occurs:

- The device requests an interrupt.
- The device times out.

If the device requests an interrupt, the driver 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 handler. The address of the timeout handler is specified as an argument to the wait for interrupt macro invocation.

12.1 I/O POSTPROCESSING

Once the driver interrupt service routine has handled an interrupt, it transfers control to the driver by issuing a JSB instruction. 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 VAX/VMS macro IOFORK. 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.

12.1.1 EXE$IOFORK

IOFORK is a macro that generates a call to the VAX/VMS routine EXE$IOFORK. EXE$IOFORK converts the driver context from that of an interrupt service routine to the context of a driver fork process in the following steps:

- It disables software timeouts by clearing the timeout enable bit in the UCB status word (UCB$V_TIM in UCB$W_STS).
- It saves R3 and R4 of the current driver context in the UCB fork block (UCB$L_FR3 and UCB$L_FR4).

- EXE$IOFORK then saves the current driver PC in the UCB fork block (UCB$L_FPC). The driver PC is the first longword on the stack upon entry to EXE$IOFORK as a result of the JSB instruction.
- It obtains the fork IPL of the device from the UCB (UCB$B_FIPL).
- It inserts the address of the UCB fork block (R5) into the fork queue corresponding to the driver fork IPL.
- Finally, if the fork block is the first entry in the fork queue, EXE$IOFORK requests a software interrupt at driver fork IPL.

The steps listed above move the critical driver fork process context into the UCB fork block; that is, they save R3 through R5 and the driver PC address. The driver fork process resumes processing when the VAX/VMS fork dispatcher dequeues the UCB fork block from the fork queue and reactivates the driver at driver fork IPL.

12.1.2 Completing an I/O Request

When VAX/VMS reactivates a driver fork process by dequeuing the fork block, the driver resumes processing of the I/O operation. If the device has completed the I/O operation without errors, the driver fork process for a DMA device proceeds as follows:

- Purges the buffered data path
- Releases the buffered data path
- Releases map registers
- Releases the controller
- Saves the status code, transfer count, and device-dependent status that is to be returned to the user process in an I/O status block
- Returns control to the operating system

Chapter 10 discusses the first three steps listed above because they relate to UNIBUS DMA transfers. The sections that follow describe the remaining three steps.

12.1.2.1 Releasing the Controller - To release the controller channel, the driver code invokes the VAX/VMS macro RELCHAN. RELCHAN calls the VAX/VMS routine IOC$RELCHAN. If another driver is waiting for the controller channel, IOC$RELCHAN grants that driver fork process the channel, restores its driver fork context from its UCB fork block, and transfers control to the saved PC. When no more drivers are awaiting the channel, IOC$RELCHAN returns control to the driver fork process that released the channel.

Drivers for single-unit controllers need not release the controller data channel (as discussed in Sections 9.3.1 and 13.1). Through code in the unit initialization routine, these drivers set up the device's unit control block to own the controller permanently.

Drivers must be executing at driver fork IPL when they invoke RELCHAN or call IOC$RELCHAN.

12.1.2.2 Saving Status, Count, and Device-Dependent Status - To save the status code, transfer count, and device-dependent status, the driver performs the following steps:

- It loads a success status code (SS$_NORMAL) into bits 0 through 15 of R0.
- If the I/O operation performed by the device is a transfer function, the driver loads the number of bytes transferred into the high-order 16 bits of R0, that is, into bits 16 through 31.
- The driver then loads device-dependent status information, if any, into R1. R0 and R1 are the status values that VAX/VMS returns to the user process in the I/O status block specified in the original Queue I/O Request system service. If the user specifies no I/O status block, VAX/VMS does not use R0 and R1.

12.1.2.3 Returning to the Operating System - Finally, the driver returns to the system by invoking the VAX/VMS macro REQCOM to complete the I/O request. REQCOM calls the VAX/VMS routine IOC$REQCOM. IOC$REQCOM locates the address of the I/O request packet corresponding to the I/O operation in the device's UCB (UCB$L_IRP). It then writes the two longwords of completion status contained in R0 and R1 into the media field of the I/O request packet (IRP$L_MEDIA and IRP$L_MEDIA+4).

IOC$REQCOM then inserts the I/O request packet in the I/O postprocessing queue. If the packet is the only entry in the postprocessing queue, IOC$REQCOM requests a software interrupt at IPL$_IOPOST so the postprocessing begins when IPL drops below IPL$_IOPOST.

If the error logging bit is set in the device's unit control block (UCB$V_ERLOGIP in UCB$W_STS), IOC$REQCOM obtains the address of the error message buffer from the unit control block (UCB$L_EMB). It then writes the following information into the error buffer:

- Final device status (UCB$W_DEVSTS)
- Final error count (UCB$B_ERTCNT)
- Two longwords of completion status (R0 and R1)

To release the error message buffer, IOC$REQCOM calls ERL$RELEASEMB. Section 13.3 describes error logging in more detail.

If any I/O request packets are awaiting driver processing, IOC$REQCOM performs the following steps:

- Dequeues a packet
- Creates a new driver fork process
- Activates the driver at the driver's start I/O routine

Otherwise, IOC$REQCOM clears the unit busy bit in the device's UCB status word (UCB$V_BSY in UCB$W_STS) and transfers control to IOC$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 VAX/VMS I/O postprocessing.

12.2 TIMEOUT HANDLERS

VAX/VMS transfers control to the driver's timeout handler if a device unit does not request an interrupt within the time limit specified in the wait for interrupt macro. The VAX/VMS timer routine scans device unit control blocks once every second to determine whether a device has timed out.

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

- It disables expected interrupt and timeout on the device by clearing bits in the device's UCB status field (UCB$V_INT and UCB$V_TIM in UCB$W_STS).
- It sets the device timeout bit in the UCB status field (UCB$V_TIMOUT in UCB$W_STS).
- It sets IPL to hardware device interrupt IPL (UCB$B_DIPL).
- It restores the saved R3 and R4 of the driver fork process from the UCB fork block (UCB$L_FR3 and UCB$L_FR4).
- It restores R5 (address of the UCB fork block).
- It computes the address of the driver's timeout handler from the saved PC in the UCB fork block (UCB$L_FPC).
- It calls the driver's timeout handler with a JSB instruction.

The driver's timeout handler executes in following context:

- R0 through R5 are saved on the stack.
- R5 contains the address of the UCB for the device that timed out.
- System address space is mapped.
- The processor is running in kernel mode.
- The processor is running on the interrupt stack.
- IPL is at hardware device interrupt level.

VAX/VMS invoked the timeout handler through an interrupt at IPL$_TIMER. Thus, the driver can lower from device IPL to driver fork IPL to process the timeout. (The driver should lower IPL with SETIPL to preserve the contents of the stack.)

When the driver fork process regains control, R3 and R4 are restored to their previous state from UCB$L_FR3 and UCB$L_FR4 respectively.

During power failure recovery, VAX/VMS forces a device timeout by altering the timeout field (UCB$L_DUETIM) of a unit control block if that device's UCB records that the unit is waiting for an interrupt or timeout (UCB$V_INT and UCB$V_TIM set in UCB$W_STS). The timeout handler can perceive that a power failure recovery is occurring by examining the power bit (UCB$V_POWER in UCB$W_STS) in the unit control block.

A timeout handler usually performs either of three functions:

- Retries the I/O operation unless a retry count is exhausted
- Aborts the I/O request
- Sends a message to an operator mailbox and resumes waiting for a subsequent interrupt or timeout

12.2.1 Retrying the I/O Operation

Some devices may retry an I/O operation after a timeout. For example, a disk driver might take the following steps after a transfer timeout:

- Invoke the following VAX/VMS macro to lower IPL to driver fork level:

  ```
  SETIPL UCB$B_FIPL(R5)
  ```

  The resulting IPL must not drop below IPL$_TIMER.

- Release map registers, data path, and controller data channel.
- If a power failure occurred, load the I/O request packet address into R3 and reload the following I/O request packet fields into the corresponding UCB fields and branch to the start I/O routine:

  ```
  UCB$W_BCNT
  UCB$W_BOFF
  UCB$L_SVAPTE
  ```

  The above steps result in a total retry of the transfer.

- If no power failure has occurred and the device driver supports error logging, call ERL$DEVICTMO to log the device timeout.
- If the retry count is not exhausted, decrease the count, clear the UCB timeout bit in UCB$W_STS, and retry the operation.
- If the retry count is exhausted, set the error code, perform a normal abort I/O clean-up operation, and invoke REQCOM.

12.2.2 Aborting the I/O Request

A driver's timeout handler aborts the I/O request when it exhausts its retry count, or when it determines, upon timeout, that a cancel I/O was requested. If the cancel I/O bit in the UCB status word (UCB$V_CANCEL in UCB$W_STS) is set, a cancel I/O was requested and the timeout handler can abort the request.

To abort an I/O request, a device driver timeout handler can perform the following sequence of steps:

- If appropriate to the device and controller, the handler clears the device control/status register.

- The handler then invokes the following VAX/VMS macro to lower IPL to driver fork level:

  ```
  SETIPL UCB$B_FIPL(R5)
  ```

  The resulting IPL must not drop below IPL$_TIMER.

- The handler releases UNIBUS adapter resources and the controller data channel, if necessary.

- It loads abort status code (SS$_ABORT) into the low word of R0.

- It clears bits 16 through 31 in R0 to indicate that no data was transferred.

- It invokes the VAX/VMS macro REQCOM, described in Section 12.1.2.3, to complete the I/O request processing.

Since the device can interrupt driver timeout processing at fork IPL, the 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.

12.2.3 Sending a Message to the Operator

The following sequence describes a timeout handler that sends a message to the operator mailbox and then goes back into a wait for interrupt or timeout state:

- It invokes the following VAX/VMS macro to lower IPL to driver fork level:

  ```
  SETIPL UCB$B_FIPL(R5)
  ```

  The resulting IPL must not drop below IPL$_TIMER.

- It checks the cancel I/O bit in the UCB status word (UCB$V_CANCEL in UCB$W_STS). If UCB$V_CANCEL is not set, the timeout handler performs the following:

  -- Saves R3 and R4 on the stack

  -- Loads an OPCOM message code, such as MSG$_DEVOFFLIN, into R4

  -- Loads the address of the operator mailbox (SYS$GL_OPRMBX) into R3

  -- Calls a VAX/VMS routine to place the message in the operator mailbox, as follows:

  ```
  JSB     G^EXE$SNDEVMSG
  ```

  -- Restores R3 and R4

  (If the cancel I/O bit is set, the timeout handler can abort the request.)

- The timeout handler then invokes the VAX/VMS macro DSBINT to raise IPL to IPL$_POWER, thereby locking out all interrupts from software and hardware.

- Finally, the timeout handler invokes the VAX/VMS macro WFIKPCH to wait for another interrupt or timeout.

When the OPCOM process reads the message in its mailbox, it sends the requested message, in this case "device-offline", to all operator terminals.

CHAPTER 13

WRITING INITIALIZATION, CANCEL I/O, AND ERROR-LOGGING ROUTINES

Drivers normally contain initialization, cancel I/O, and error-logging routines. The driver prologue table and the driver dispatch table specify the addresses of initialization routines. The driver dispatch table contains the addresses of the cancel I/O and error-logging routines. Whether these routines are required depends on the type of device.

13.1 INITIALIZATION ROUTINES

Most device controllers and device units require initialization under the following circumstances:

- When the driver loading procedure loads a device driver for the controller and device units
- During recovery from a power failure

Initialization routines ready controllers and device units for operation. Depending on the device characteristics, initialization routines perform any of the actions listed below:

- Enable controller interrupts
- Clear error status bits in device registers
- Initiate a device operation such as clearing a drive or acknowledging a pack
- Store values in UCB fields that cannot be addressed with a DPT_STORE macro; that is, fields more than 256 bytes from the start of the unit control block
- Permanently allocate UNIBUS adapter resources, as described in Chapter 10
- Set the online bit (UCB$V_ONLINE in UCB$W_STS) in the unit control block
- Fill in IDB$L_OWNER for single-unit devices such as a line printer

13.1.1 Initialization During Driver Loading

The initialization performed during driver loading depends upon whether the driver is being loaded for the first time or replacing a driver that was previously loaded.

The SYSGEN commands AUTOCONFIGURE, CONNECT, and LOAD add new drivers to the configuration. The LOAD command loads the driver into nonpaged system memory but does not call any driver-specific routines or execute any initialization requests specified in DPT_STORE macro invocations. AUTOCONFIGURE and CONNECT create the I/O data structures associated with the device driver, call driver-specific initialization routines, and perform requests specified in DPT_STORE macro invocations.

For each new device they add to the system, AUTOCONFIGURE and CONNECT carry out the following steps:

- Create a unit control block for the device. If this is the first occurrence of device-name and controller, the commands create a device data block, a channel request block, and an interrupt dispatch block.
- Perform the initialization operations specified by the DPT_STORE macros within the initialization and reinitialization portions of the driver prologue table.
- Relocate all addresses in the driver dispatch table and function decision table to system virtual addresses.
- Call the controller initialization routine specified in the channel request block, if the CRB was created.
- Call the unit initialization routine (if any) specified in the driver dispatch table. If no routine exists in the DDT, call the unit initialization routine (if any) specified in the CRB.

The AUTOCONFIGURE and CONNECT command operations raise IPL to IPL$_POWER to prevent interruption of the initialization routines.

The RELOAD command replaces an existing driver with a new driver. The command loads the new driver code into nonpaged system memory. Unlike the other SYSGEN commands for driver loading, RELOAD assumes that the I/O data structures associated with the driver already exist, and thus updates the data base to reflect the modified code and its different location in system virtual address space.

The RELOAD command performs the following functions:

- Executes requests specified by DPT_STORE macro invocations in only the reinitialization section of the driver prologue table
- Relocates all addresses in the function decision table and driver dispatch table to system virtual addresses
- Calls the controller initialization routine

Chapter 14 contains detailed descriptions of all SYSGEN commands related to device drivers.

13.1.2 Initialization During Recovery from a Power Failure

During powerfail recovery procedures, the operating system locates every unit control block in the I/O data base. Each unit control block points to a channel request block for the device's controller. The channel request block contains the address of the controller

initialization routine, if one was specified. The system uses the following chain of pointers to locate the address of the initialization routine:

DDB ⟶ UCB ⟶ CRB ⟶ controller initialization routine

The operating system calls the initialization routine for each controller if one was specified in a DPT_STORE macro for the CRB$L_INTD+VEC$L_INITIAL of the channel request block.

Next, the system checks for a device unit initialization routine. First, the system examines the unit initialization field in the driver dispatch table (DDT$L_UNITINIT). If the field does not contain an address, the system checks the channel request block using the following chain of pointers:

DDB ⟶ UCB ⟶ CRB ⟶ device unit initialization routine

MASSBUS drivers store unit initialization routines addresses only in the driver dispatch table.

If either the channel request block or the driver dispatch table contains a nonzero address for such a routine, the system calls the routine to initialize the device unit. The system calls only one routine; if the driver dispatch table contains an address, the CRB address is ignored.

13.1.3 Initialization Context

The VAX/VMS operating system always calls controller and unit initialization routines with IPL raised to IPL$_POWER. The high IPL prevents any interrupts from reaching the processor while initialization is occurring. The initialization routines must not lower IPL. The system calls initialization routines with a JSB instruction; the routines return by executing an RSB instruction.

Controller initialization routines are device-dependent. For example, a card reader controller initialization routine might enable interrupts from the device by setting the interrupt enable bit in the device's control/status register. A disk controller initialization routine, on the other hand, might enable interrupts and initialize all unit status registers.

At the time of a call to a controller initialization routine, the registers contain the following values:

Register	Value
R4	Address of the control/status register
R5	Address of the interrupt data block that describes the controller
R6	Address of the device data block associated with the controller
R8	Address of the channel request block for the controller

Device unit initialization routines are useful for initializing device-dependent fields in the unit control block. For example, disk initialization routines can also set disk drive parameters (such as

number of cylinders) in the unit control block and wait for online units to spin up to speed. Unit initialization routines must set the online bit in the unit control block (UCB$V_ONLINE) to declare the unit to be online.

If a device needs permanently allocated UNIBUS adapter resources, a unit initialization routine can call VAX/VMS UNIBUS adapter resource management routines to allocate the resources. Then, the initialization routine can set bits in the CRB UNIBUS adapter resource description fields, for example, VEC$V_PATHLOCK in CRB$L_INTD+VEC$B_DATAPATH.

At the time of a call to a device unit initialization routine, the registers contain the following values:

Register	Value
R3	Address of the primary control/status register
R4	Address of the secondary control/status register; R4 is equal to R3 if there is no secondary CSR
R5	Address of the device's unit control block

If driver initialization routines modify R4 through R11, the routines must save the contents of the registers before use and restore them before returning control to the operating system.

13.2 CANCEL I/O ROUTINE

VAX/VMS routines call the cancel I/O routine in a device driver under the following circumstances:

- When a process issues a Cancel I/O on Channel system service
- When a process deallocates a device and no process I/O channels are assigned to the device
- When a process deassigns a channel from a device
- 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

The VAX/VMS routine EXE$CANCEL locates the unit control block for the device associated with a process I/O channel from a pointer in the channel request block, as follows:

channel index number ⟶ CCB ⟶ UCB address

EXE$CANCEL takes the following steps:

- Raises IPL to fork level
- Removes all I/O request packets associated with the process from the device's I/O request packet wait queue
- Sets the status code SS$_CANCEL in IRP$L_MEDIA
- For buffered I/O read operation, clears the buffered read function bit (IRP$V_FUNC) in IRP$W_STS

- Inserts the I/O packets removed from the packet wait queue into the I/O postprocessing queue
- If the I/O postprocessing queue is empty, requests a software interrupt

Then, EXE$CANCEL calls the cancel I/O routine specified in the driver dispatch table of the associated device driver. EXE$CANCEL locates the routine using the following chain of pointers:

UCB ⟶ DDT ⟶ address of the cancel I/O routine

The cancel I/O routine gives the driver an opportunity to prevent further device-specific processing of the I/O request currently being processed on the device.

13.2.1 Context of a Cancel I/O Routine

When EXE$CANCEL calls the cancel I/O routine, IPL is at driver fork IPL so that the routine can read and modify the device's unit control block. Registers at the time of the call contain the following values:

Register	Value
R2	Channel index number
R3	Address of the current I/O request packet
R4	Address of the process control block of the process for which the Cancel I/O on Channel system service is being performed
R5	Address of the device's unit control block
R8	Reason for the call to cancel the I/O request. Reason codes are defined by the $CANDEF macro. Possible values for R8 include:

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

If a cancel I/O routine uses registers other than R0 through R3, it must save the registers and restore them before exiting.

Device drivers may want to base their cancel I/O operation on whether the cancel I/O request is the result of a channel deassignment (CAN$_DASSGN). For example, the terminal driver cancels out-of-band AST requests only if the call to its cancel I/O routine results from a Deassign I/O Channel ($DASSGN) system service call.

13.2.2 Drivers that Need No Cancel I/O Routine

Some devices do not need any device-dependent processing performed for an I/O request; you can omit the CANCEL argument from the DDTAB

macro. In this case, the DDTAB macro expansion loads the address of the VAX/VMS routine IOC$RETURN into the appropriate position in the driver dispatch table. The routine IOC$RETURN executes a single RSB instruction.

13.2.3 Device-Independent Cancel I/O Routine

Drivers can specify the VAX/VMS routine IOC$CANCELIO as the value of the CANCEL argument in the DDTAB macro invocation. IOC$CANCELIO cancels I/O to a device in the following device-independent manner:

- It confirms that the device is busy by examining the device busy bit in the UCB status word (UCB$V_BSY in UCB$W_STS).
- It locates the process identification field in the I/O packet currently being processed on the device using the following chain of pointers:

  UCB ⟶ IRP ⟶ process identification field

  IOC$CANCELIO confirms that the field (IRP$L_PID) contains the same value as the corresponding field in the process control block (PCB$L_PID).
- It confirms that the specified channel index number is the same as the value stored in the I/O request packet channel index field (IRP$W_CHAN).
- It sets the cancel I/O bit in the UCB status word (UCB$V_CANCEL in UCB$W_STS).

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

13.2.4 Device-Dependent Cancel I/O Routines

Drivers that include their own cancel I/O routines must perform the first three steps of IOC$CANCELIO listed in Section 13.2.3 to determine whether the I/O request being processed originates from the process canceling I/O on a channel. If the three checks succeed, the cancel routine can proceed in a device-specific manner.

13.3 ERROR-LOGGING ROUTINES

The operating system supplies two routines that drivers can call to allocate and fill error-logging buffers after a device error or timeout occurs:

- ERL$DEVICERR
- ERL$DEVICTMO

Both routines expect to find the address of the device unit control block in R5. Drivers must call them at fork IPL. Each routine performs the following steps:

- It allocates an error log buffer of the length specified in the device's driver dispatch table. It uses the following chain of pointers to locate the buffer length:

  UCB ⟶ DDT ⟶ length of error log buffer

- It loads into the buffer fields from the unit control block, the I/O request packet, and the device data block.

- It loads the address of the error message buffer location where device register contents are to be stored.

- It calls a register dump routine in the device driver. It locates the routine using the following chain of pointers:

  UCB ⟶ DDT ⟶ register dump routine address

Specify the address of a register dump routine with the value of the REGDMP argument to the DDTAB macro invocation.

The register dump routine can expect the following registers to be loaded:

Register	Content
R0	Address of the buffer
R4	Address of the control/status register if the driver used the WFIKPCH macro to wait for an interrupt or timeout
R5	Address of the device's unit control block

The dump routine should save and restore R3 through R11 if the routine requires their use.

The driver register dump routine should fill the buffer as follows:

- Write a longword value representing the number of device registers to be written into the buffer

- Move device register longword values into the buffer following the register count longword

The routine must store the contents of each device register to be logged in a longword in the buffer. For example, the following instruction stores the contents of the device register:

```
MOVZWL  TD_STATUS(R4),(R0)+
```

A driver that supports error logging must satisfy the following prerequisites:

- It must use the error log extension to the unit control block.

- It must ensure that DDT$W_ERRORBUF is large enough to accommodate EMB$L_DV_REGSAV+4 plus one longword for each register to be dumped

- Its driver prologue table must set the device characteristic DEV$V_ELG in UCB$_DEVCHAR.

CHAPTER 14

LOADING A DEVICE DRIVER

You can load a user-written device driver any time after the system is bootstrapped. If the driver contains an error and the error does not crash or corrupt the operating system, you can correct the error and reload a new version of the driver.

14.1 PREPARATION FOR LOADING

To prepare a device driver for loading, take the following steps:

- Write the device driver in one or more source files. If the driver comprises multiple source files, you must insert a .PSECT directive before any generated code in all files except the file that contains the DPTAB and DDTAB macro invocations. The following .PSECT must be used:

  ```
  .PSECT   $$$115_DRIVER
  ```

  If a single source file contains the driver, you must not specify any .PSECT directives. The declaration of the DPTAB and DDTAB macros establish driver program sections correctly.

- Assemble the source file(s) with the system macro library (SYS$LIBRARY:LIB.MLB). For example:

  ```
  $ MACRO MYDRIVER.MAR+SYS$LIBRARY:LIB.MLB/LIBRARY
  ```

- Link the object file with the VAX/VMS global symbol table, which is located in SYS$SYSTEM and called SYS.STB. If the driver consists of multiple source files, you must specify the file that contains the driver prologue table as the first file in the list. The linker options file must contain a BASE statement specifying a zero base for the executable image. The following is an example of the creation of the options file and the LINK command used to link a driver:

  ```
  $ CREATE MYDRIVER.OPT
  BASE=0
  CTRL/Z
  $ LINK /NOTRACE MYDRIVER1[,MYDRIVER2,...],-
         MYDRIVER.OPT/OPTIONS,-
         SYS$SYSTEM:SYS.STB/SELECTIVE_SEARCH
  ```

  The resulting image must consist of a single image section. The linker will report that the image has no transfer address.

14.2 LOADING THE DRIVER

Once the driver has linked correctly, it is ready to be loaded. To load the driver into system virtual memory, run the System Generation Utility (SYSGEN) from the system manager's account or from an account having Change Mode to Kernel privilege using the following command:

```
$ RUN SYS$SYSTEM:SYSGEN
```

SYSGEN responds with a prompt and waits for further input:

```
SYSGEN>
```

The VAX-11 Utilities Reference Manual describes the full set of SYSGEN commands. The sections that follow describe those commands SYSGEN uses to load drivers:

- LOAD (requires Change Mode to Kernel (CMKRNL) privilege)
- CONNECT (requires CMKRNL privilege)
- RELOAD (requires CMKRNL privilege)
- SHOW/ADAPTER (requires CMEXEC privilege)
- SHOW/CONFIGURATION (requires CMEXEC privilege)
- SHOW/DEVICE (requires CMEXEC privilege)

In addition, you should understand SYSGEN's automatic configuration feature, as described in Section 14.3.

14.2.1 LOAD Command

To load a device driver, issue the LOAD command. If the controller has only a single unit attached to it, issue the CONNECT command.

Format

```
LOAD driver-file-spec
```

driver-file-spec
: The file specification of the image file containing the I/O driver to be loaded. The LOAD command obtains the driver name from the DPT$T_NAME field in the driver prologue table. If the name of the driver being loaded matches the name of any driver already in the configuration, the LOAD command will not load the driver.

  SYS$SYSTEM is the default device and directory name. EXE is the default file type.

Description

The driver loading procedure compares the name field in the driver prologue table of the driver being loaded with the name field in the driver prologue tables of the drivers already loaded into system memory. If no match is found, the procedure loads the new driver into contiguous locations in nonpaged pool and links the driver prologue table into the DPT linked list. If the procedure finds a match, it takes no further action.

Example

```
SYSGEN> LOAD CRDRIVER
```

This command loads the driver found in SYS$SYSTEM:CRDRIVER.EXE (the card reader driver).

14.2.2 CONNECT Command

The CONNECT command creates I/O data base control blocks for devices. The CONNECT command can also load the driver if it has not been previously loaded into system memory.

Format

```
CONNECT device-name required-quals [optional-quals]
```

Command Qualifiers

```
/[NO]ADAPTER=nexus
/CSR=csr-address
/VECTOR=vector-address
/DRIVERNAME=driver-name (optional)
/NUMVEC=number (optional)
/ADPUNIT=unit-number (optional)
/MAXUNITS=number (optional)
```

Parameter

device-name

The name of the device for which control blocks are to be added to the I/O data base. Specify the device name in the following format:

```
ddcu

dd = device code (up to 9 alphabetic characters)
c  = controller designation (alphabetic)
u  = unit number
```

For example, LPA0 specifies the line printer (dev) on controller A (c) at unit 0 (u). When specifying the device name, do not follow it with a colon (:).

The device code and controller specification must be a unique and accurate device name and controller combination. If control blocks for the specified device/controller already exist, the driver loading procedure does not create any control blocks or perform any initialization operations. If the device/controller name does not accurately name a device, the procedure will create spurious control blocks.

Required Qualifiers

/[NO]ADAPTER=nexus

The nexus value of the UNIBUS adapter, MASSBUS adapter, or other controller to which the device unit is attached. The nexus can be a number or a generic name listed by the /ADAPTER qualifier to the SYSGEN command SHOW. See Section 14.2.4 for a discussion of SHOW/ADAPTER.

Specify a nexus number in the range 0 through 15. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X).

Nexus values for VAX-11 processors are listed below:

		VAX-11/730	VAX-11/750	VAX-11/780
UNIBUS adapter	0	3	8	3
	1	-	9	4
	2	-	-	5
	3	-	-	6
MASSBUS adapter	0	-	4	8
	1	-	5	9
	2	-	6	10
	3	-	-	11

Issue the CONNECT command with the /NOADAPTER qualifier to connect drivers associated with software devices. The mailbox driver is an example of this type of driver.

/CSR=csr-address
The UNIBUS address of the control/status register for the device. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X).

/VECTOR=vector-address
The UNIBUS address of the interrupt vector for the device. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X). Section 14.3 provides additional information on vector and CSR assignments.

Optional Qualifiers

/NUMVEC=number
The number of interrupt vectors for the device. If this qualifier is omitted, the number of vectors defaults to 1. The number specified by the /VECTOR qualifier is the address of the lowest vector. Vectors must be contiguous.

/DRIVERNAME=driver-name
The name of the driver that handles the device being connected. If this qualifier is omitted, CONNECT follows one of two procedures to supply a default name. If the device to be connected is the first unit on the controller, CONNECT concatenates the first two characters of the device code with "DRIVER", for example, LPDRIVER. Otherwise, CONNECT obtains the driver name from the DDB$T_DRVNAME field in the controller's device data block.

Consult the SYSGEN device table in Section 14.3.2 for the driver names of the devices supported by VAX/VMS.

/ADPUNIT=unit-number
The unit number of a device on the MASSBUS adapter. The unit number for a disk drive is the number of the plug on the drive. For magnetic tape drives, the unit number corresponds to the tape controller number.

/MAXUNITS=number
The maximum number of units attached to the controller. This number determines the size of the UCB list appended to the

interrupt dispatch block. If specified, this value overrides the maximum number of units designated in the driver prologue table. The maximum number of units is stored in the IDB field IDB$W_UNITS.

Description

The I/O data base contains a linked list of driver prologue tables. The CONNECT command looks for a device driver by scanning the driver prologue tables and comparing the DPT$T_NAME field in each DPT with the specified or defaulted driver name. If no match is found, the driver loading procedure loads the driver image SYS$SYSTEM:drivername.EXE; see Section 14.2.1.

Then the loading procedure examines the I/O data base for control blocks that support the specified device. The procedure creates the following control blocks if they do not exist:

- Device data block -- the procedure creates a device data block for the generic device name/controller string specified if such a device data block does not exist.

  When the procedure creates a device data block for a UNIBUS device, it also creates a channel request block and an interrupt dispatch block.

- Unit control block -- the procedure creates a unit control block if it has just created a device data block or if a unit control block for the specified device does not exist. If a unit control block already exists, the procedure continues to execute but makes no more modifications to the I/O data base.

After creating the control blocks, the driver loading procedure performs the following initialization operations:

- Performs the initialization operations specified by the DPT_STORE macros in the initialization and reinitialization portions of the driver prologue table.

- Relocates all addresses in the driver dispatch table and function decision table to absolute system virtual addresses.

- Raises IPL to IPL$_POWER so that initialization is not interrupted.

- If a new channel request block was created, the procedure calls the controller initialization routine (if one exists) specified by CRB$L_INTD+VEC$L_INITIAL.

- Calls the unit initialization routine (if one exists) specified by DDT$L_UNITINIT. If the DDT contains no unit initialization routine, the procedure calls the unit initialization routine (if any) specified by CRB$L_INTD+VEC$L_UNITINIT.

You should specify CONNECT commands with extreme caution. The driver and data base loading procedure does little error checking. If you specify a vector that has already been defined, the procedure rejects the CONNECT command. However, if the CONNECT command specifies an incorrect CSR address, the I/O data base is apt to become corrupted. The result is a system failure.

If the CONNECT command specifies an existing controller and a new device unit, the procedure creates a unit control block for the new unit and calls a unit initialization routine for the unit.

A CONNECT command that specifies a device name with a new controller causes the driver loading procedure to create a device data block, channel request block, interrupt dispatch block, and unit control block and to call controller and unit initialization routines.

Example

```
SYSGEN> CONNECT LPA0 /ADAPTER=UB0/CSR=%O777514/VECTOR=%O200
```

This command loads the driver LPDRIVER, if it is not already loaded, and creates the device data base (DDB, CRB, IDB, and UCB) needed to describe LPA0.

14.2.3 RELOAD Command

The RELOAD command loads a driver and removes a previously-loaded version of that driver. The RELOAD command provides all of the functions of LOAD, except that it loads the driver regardless of whether it is already loaded.

If any of the units associated with the driver are busy, the driver cannot be reloaded; SYSGEN issues an error message.

Format

RELOAD driver-file-spec

driver-file-spec
: The file specification of the image file containing the driver to be loaded.

Description

To reload the driver, the driver loading procedure compares the name field in the driver prologue table of the driver being loaded with the name field in the driver prologue tables of drivers already loaded into system memory. If no match is found, RELOAD loads the driver as described in Section 14.2.1.

If the procedure finds a match, it first confirms that the current driver can be replaced by the new driver in the following steps:

- Confirms that the DPT$M_NOUNLOAD flag in the driver prologue table of the current driver is not set
- Calls the current driver's unload routine, if one exists, and confirms that the returned status is a success code
- Ensures that no devices that use the current driver are busy, as indicated by the UCB$V_BSY bit set in UCB$W_STS

If these checks succeed, the procedure replaces the current driver with the new driver. The procedure loads the new driver into contiguous locations in nonpaged system memory and searches the I/O data base for references to the driver. If any device data block refers to the driver being reloaded, the procedure

reinitializes fields of the device and controller control blocks according to the reinitialization instructions in the new driver's prologue table; Chapter 7 describes the DPT reinitialization fields.

Fields that must be reinitialized when a driver is reloaded include those that contain relative addresses within the driver:

- Addresses of driver interrupt service routines
- Addresses of device unit and controller initialization routines
- Address of the driver dispatch table

Once the loading procedure has reinitialized fields, it calls the driver controller initialization routine. (It does not call the unit initialization routine.) The procedure then removes the newly replaced driver from the DPT list and deallocates the nonpaged system space the old driver occupied. Finally, the loading procedure links the address of the new driver prologue table to the DPT list.

Use the RELOAD command only when all devices supported by the driver are inactive. The activity checks made by the RELOAD command may not detect all device activity, and changing a driver while an I/O request is being processed will cause a system failure.

14.2.4 SHOW/ADAPTER

The SHOW/ADAPTER command displays a list of nexus values of adapters in the system configuration. Use of the SHOW/ADAPTER command requires Change Mode to Executive (CMEXEC) privilege.

Format

```
SHOW/ADAPTER
```

Description

The SHOW/ADAPTER command displays nexus numbers and generic names of UNIBUS and MASSBUS adapters, memory controllers, and device interconnects such as the DR32.

Example

```
SYSGEN> SHOW/ADAPTER

CPU Type:  11/780
Hardware Revision #96

  Nexus   Generic Name or Description

     1    16K memory, non-interleaved
     4    UB0
     5    UB1
     8    MB0
     9    MB1
```

This example shows a VAX-11780 that uses one memory controller composed of 16K-bit chips, two UNIBUS adapters, and two MASSBUS adapters.

14.2.5 SHOW/CONFIGURATION

The SHOW/CONFIGURATION command displays information about the system configuration.

Format

```
SHOW/CONFIGURATION  [/ADAPTER=nexus]
                    [/COMMAND_FILE]
                    [/OUTPUT=file-spec]
```

nexus
The nexus value of the UNIBUS adapter, MASSBUS adapter, or other interconnect to be displayed.

file-spec
The file specification of an optional output file.

Descripton

The SHOW/CONFIGURATION command displays the device name, number of units, nexus number and type and shows the CSR and vector addresses of devices connected or autoconfigured to the system. You can direct the display to an output file with the /OUTPUT qualifier. If you combine the /OUTPUT and /COMMAND_FILE qualifiers, SYSGEN formats all the device data into CONNECT commands and copies them to the output file you specify. In this way, you can remove a device from floating address space without completely rejumpering the CSR and vector addresses of the remaining devices. See the VAX-11 Utilities Reference Manual for more details.

Example

```
SYSGEN> SHOW/CONFIGURATION/ADAPTER=UB1


     System CSR and Vectors on  4-JAN-1982 14:58:26.08

Name: LPA  Units: 1  Nexus:4  (UBA) CSR: 777514  Vector1: 200  Vector2: 000
Name: DYA  Units: 2  Nexus:4  (UBA) CSR: 777170  Vector1: 264  Vector2: 000
Name: XMA  Units: 1  Nexus:4  (UBA) CSR: 760070  Vector1: 300  Vector2: 304
Name: XMB  Units: 1  Nexus:4  (UBA) CSR: 760100  Vector1: 310  Vector2: 314
Name: XMC  Units: 1  Nexus:4  (UBA) CSR: 760110  Vector1: 320  Vector2: 324
Name: TTA  Units: 8  Nexus:4  (UBA) CSR: 760130  Vector1: 330  Vector2: 334
Name: TTB  Units: 8  Nexus:4  (UBA) CSR: 760140  Vector1: 340  Vector2: 344
Name: TTC  Units: 8  Nexus:4  (UBA) CSR: 760150  Vector1: 350  Vector2: 354
Name: TTD  Units: 8  Nexus:4  (UBA) CSR: 760160  Vector1: 360  Vector2: 364
Name: TTE  Units: 8  Nexus:4  (UBA) CSR: 760170  Vector1: 370  Vector2: 374
```

14.2.6 SHOW/DEVICE

The SHOW/DEVICE command displays the location of a driver and the I/O data base describing its devices in system virtual memory. This command requires Change Mode to Executive (CMEXEC) privilege.

Format

```
SHOW/DEVICE [=driver-name]
```

driver-name
Name of the driver for which the information is to be displayed. If a driver name is not specified, the command displays information about all drivers and devices known to the system.

Description

The SHOW/DEVICE command displays the following information:

- Name of the driver
- The driver's starting and ending virtual addresses; the starting address is the address of the driver prologue table
- The generic device/controller name associated with the driver
- The addresses of the device data block, channel request block, and interrupt data block for the generic device/controller supported by the driver
- The unit numbers and UCB addresses for each device unit associated with the driver

Example

```
SYSGEN> SHOW/DEVICE=TMDRIVER

__DRIVER___START____END___DEV___DDB_____CRB_____IDB_____UNIT___UCB

TMDRIVER 8009DF00 8009F020
                          MTA 800BA660 800BA6C0 800BA360
                                                          0 8009F020
                                                          1 8009F0C0
```

14.3 AUTOCONFIGURATION

The standard VAX/VMS system start-up file runs SYSGEN to create and initialize an I/O data base that describes all supported DIGITAL peripherals in the configuration. The following command requests SYSGEN to prepare a data base for all supported DIGITAL devices attached to every UNIBUS and MASSBUS:

```
SYSGEN>  AUTOCONFIGURE ALL
```

To configure devices attached to the UNIBUS, SYSGEN goes through the steps described in subsequent sections of this chapter.

DIGITAL-supported devices are attached to the UNIBUS according to a table found in Appendix A of the PDP-11 Peripherals Handbook. The basic rules follow:

- A device of type A is always at a fixed and predefined CSR address; the device always interrupts at a fixed and predefined vector address; only one example of device A can be configured in each system.
- A device of type B is identical to type A except that 1 through n examples can be configured in a single system. Examples 2 through n are also located at fixed and predefined CSRs and vector addresses.
- Devices of type C (1 through n of them) are always at fixed and predefined CSR addresses; however, the interrupt vector addresses vary according to what other devices are present on the system.
- Devices of type D (1 through n of them) are at CSR addresses and vector addresses that vary according to what other devices are present on the system.

The CSR and vector addresses that vary are called floating addresses. The devices must be located in floating CSR and vector space according to the order in which the devices appear in the SYSGEN device table. This table, shown in Section 14.3.2, lists all the type A and type B devices supported by VAX/VMS. It also lists the type C and type D devices that are recognized by SYSGEN's autoconfiguration procedure.

The base of floating vector space is 300 (octal). The base of floating CSR space is 760010 (octal).

14.3.1 The SYSGEN Autoconfiguration Facility

The SYSGEN utility automatically configures a UNIBUS adapter as follows:

- It initializes the base of floating space to 300 (octal) and 760010 (octal) for vectors and CSRs, respectively.
- It tests fixed and floating CSR address space for all known DIGITAL devices.
- When a device is found at a CSR, SYSGEN reserves floating CSR and vector space for that device, if necessary.
- It searches for the name of the driver associated with the device by checking the SYSGEN device table (shown in Section 14.3.2) and the directory SYS$SYSTEM. If the driver has already been loaded or exists as an image file in SYS$SYSTEM, SYSGEN creates and initializes the I/O data base for that device and loads the driver image if necessary. If the device at the CSR is supported by VAX/VMS and SYSGEN cannot locate its associated driver image, it generates an error message. If the device is unsupported and has no corresponding driver image, SYSGEN ignores the condition.

14.3.2 The SYSGEN Device Table

The SYSGEN device table lists the characteristics of all DIGITAL devices. This table indicates the following information for each device type:

- The device controller name
- The name of the device driver, and whether it is supported
- The name of the device recognized by VAX/VMS
- The interrupt vector
- The number of interrupt vectors per controller
- The address of the first device register for each controller recognized by SYSGEN (the first register is usually, but not always, the CSR)
- The number of registers per controller

Currently, the SYSGEN device table lists the following devices:

Device	Name	Vector	#Vectors	Alignment	CSR/Rank	#Registers	Driver	Support
CRA	CR11	230			777160		CRDRIVER	yes
DMA	RK611	210			777440		DMDRIVER	yes
LPA	LP11	200 170 174 270 274			777514 764004 764014 764024 764034		LPDRIVER	yes
DLA	RL11	160			774400		DLDRIVER	yes
MSA	TS11	224			772520		TSDRIVER	yes
DYA	RX211	264			777170		DYDRIVER	yes
DQA	RB730	250			775606		DQDRIVER	no
PUA	UDA	154			772150		PUDRIVER	yes
OMA	DC11	float	2	8	774000 774010 774020 774030 . . . (maximum of 32 units)		OMDRIVER	no
DDA	TU58	float	2	8	776500 776510 776520 776530 . . . (maximum of 16 units)		DDDRIVER	yes

Device	Name	Vector	#Vectors	Alignment	CSR/Rank	#Registers	Driver	Support
OBA	DN11	float	1	4	775200 775210 775220 775230 . . . (maximum of 16 units)		OBDRIVER	no
YMA	DM11B	float	1	4	770500 770510 770520 770530 . . . (maximum of 16 units)		YMDRIVER	no
OAA	DR11C	float	2	8	767600 767570 767560 767550 . . . (maximum of 16 units)		OADRIVER	no
PRA	PR611	float	1	8	772600 772604 772610 772614 . . . (maximum of 8 units)		PRDRIVER	no

Device	Name	Vector	#Vectors	Alignment	CSR/Rank	#Registers	Driver	Support
PPA	PP611	float	1	8	772700 772704 772710 772714 . . . (maximum of 8 units)		PPDRIVER	no
OCA	DT11	float	2	8	777420 777422 777424 777426 . . . (maximum of 8 units)		OCDRIVER	no
ODA	DX11	float	2	8	776200 776240		ODDRIVER	no
YLA	DL11C	float	2	8	775610 775620 775630 775640 . . . (maximum of 31 units)		YLDRIVER	no
YJA	DJ11	float	2	8	float	4	YJDRIVER	no
YHA	DH11	float	2	8	float	8	YHDRIVER	no
OEA	GT40	float	4	8	772000 772010		OEDRIVER	no

Device	Name	Vector	#Vectors	Alignment	CSR/Rank	#Registers	Driver	Support
LSA	LPS11	float	6	8	770400		LSDRIVER	no
XQA	DQ11	float	2	8	float	4	XQDRIVER	no
OFA	KW11W	float	2	8	772400		OFDRIVER	no
XUA	DU11	float	2	8	float	4	XUDRIVER	no
XWA	DUP11	float	2	8	float	4	no driver[1]	no
XVA	DV11	float	3	8	775000 775040 775100 775140		XVDRIVER	no
OGA	LK11	float	2	8	float	4	OGDRIVER	no
XMA	DMC11	float	2	8	float	4	XMDRIVER	yes
TTA	DZ11	float	2	8	float	4	DZDRIVER	yes
XKA	KMC11	float	2	8	float	4	XKDRIVER	no
OHA	LPP11	float	2	8	float	4	OHDRIVER	no
OIA	VMV21	float	2	8	float	4	OIDRIVER	no
OJA	VMV31	float	2	8	float	8	OJDRIVER	no
OKA	DWR70	float	2	8	float	4	OKDRIVER	no
DLB	RL11	float	1	4	float	4	DLDRIVER	yes
MSB	TS11	float	1	4	772524 772530 772534		TSDRIVER	yes

1. Because there are multiple drivers for this device, AUTOCONFIGURE does not load any driver. These devices must be connected.

Device	Name	Vector	#Vectors	Alignment	CSR/Rank	#Registers	Driver	Support
LAA	LPA11	float	2	8	770460		LADRIVER	yes
LAB	LPA11	float	2	8	float	8	LADRIVER	yes
OLA	KW11C	float	2	8	float	4	OLDRIVER	no
RSVA	RSV	float	1	8	float	4	RSVDRIVER	no
DYB	RX211	float	1	4	float	4	DYDRIVER	yes
XAA	DR11W	float	1	4	float	4	XADRIVER	yes
XBA	DR11B	124			772410		XBDRIVER	no
XBB	DR11B	float	1	4	772430	4	XBDRIVER	no
XBC	DR11B	float	1	4	float	4	XBDRIVER	no
XDA	DMP11	float	2	8	float	4	XDDRIVER	yes
ONA	DPV11	float	2	8	float	4	ONDRIVER	no
ISA	ISB11	float	2	8	float	4	ISDRIVER	no
OOA	DMV11	float	2	8	float	8	OODRIVER	no
UNA	UNA	float	1	4	float	4	XEDRIVER	no
PUB	UDA	float	1	4	float	2	PUDRIVER	yes
TXA	DMF32	float	8	4	float	16	YCDRIVER	yes
XGA							XGDRIVER	yes
LCA							LCDRIVER	yes
XIA							XIDRIVER	no
XSA	KMS11	float	3	8	float	8	XSDRIVER	no
XPA	PCL11	float	2	8	764200 764240 764300 764340		XPDRIVER	no

Devices not listed in the SYSGEN device table include:

- Non-DIGITAL-supplied devices with fixed CSR and vector addresses. These devices have no effect on autoconfiguration. Customer-built devices should be assigned CSR and vector addresses beyond the floating address space reserved for DIGITAL-supplied devices.
- Those DIGITAL-supplied, floating vector devices that the AUTOCONFIGURE command does not recognize. Use the CONNECT command to attach these devices to the system.

14.3.3 Device Driver Control of Autoconfiguration

The SYSGEN autoconfiguration facility provides two features that drivers can use to control the automatic configuration of the devices they operate. These features are invoked through the DEFUNITS and DELIVER arguments to the DPTAB macro.

The DEFUNITS argument to the DPTAB macro specifies a default number of units to be configured into the system. The DPTAB macro copies this value to the DPT$W_DEFUNITS field in the driver prologue table. The SYSGEN autoconfiguration facility reads this field and creates unit control blocks numbered zero through the default unit number minus one. The default value of DEFUNITS is one.

The DELIVER argument to the DPTAB macro specifies the address of a driver-specific unit delivery action routine. An offset to this routine is stored in the DPT$W_DELIVER field within the driver prologue table. When the DELIVER argument is present, the SYSGEN autoconfiguration facility calls the action routine once for each unit for the number of units specified in the DEFUNITS argument. If the action routine returns a true status in R0, the unit is configured. If the status in R0 is false, the autoconfiguration facility does not configure the device. If the DELIVER argument is not used, the unit delivery feature is disabled.

SYSGEN calls the unit delivery action routine with a JSB instruction in the following context:

- Interrupt priority level is at IPL$_POWER (31).
- R0 through R2 are available for use.
- R3 contains the address of the interrupt dispatch block, of one exists. If none exists, the value contained in R3 is zero.
- R4 contains the address of the control/status register for the controller.
- R5 contains the number of the unit that the routine must decide whether or not to configure.
- R6 contains the base address of UNIBUS adapter I/O space.
- R7 contains the address of the configuration control block (ACF).
- R8 contains the address of the UNIBUS adapter control block.

The configuration control block is described in Appendix A.

The VAX/VMS DZ11 device driver specifies a default unit number of eight and no action routine to configure eight terminal units automatically for each DZ11 CSR. The RK611 device driver gives eight as the default number of units and also spcifies the address of a unit delivery action routine that is called once for each of the eight possible devices on the controller. The unit delivery routine prevents the creation of unit control blocks for devices that do not respond to a request that tests for their presence.

14.3.4 Floating Vector Address Calculation

To calculate the floating vector address of a device, the SYSGEN utility rounds the current floating vector base (CFVB) up to the next valid vector address boundary for the next device in the table.

If a device is present, SYSGEN reserves floating vector space for the device by computing a new CFVB:

```
CFVB + (4 * number_of_vectors) ⟶ CFVB
```

14.3.5 Floating CSR Address Calculation

To calculate the floating CSR address of a device, SYSGEN rounds the current floating CSR base (CFCB) up to the next valid floating CSR address. Floating CSR addresses must fall on an 8-byte boundary.

SYSGEN tests the CSR address (CFCB) for the next device in the device table by executing a test word (TSTW) instruction on the address and noting whether there is a response at that address.

If the device is present, SYSGEN reserves floating CSR address space for the device by computing a new CFCB:

```
CFCB + bytes_in_register_set ⟶ CFCB
```

When all devices of a particular type have been located and their floating CSR space reserved, SYSGEN reserves an extra block of CSR space to indicate a change to a new device type:

```
CFCB + 8 ⟶ CFCB
```

If the device is not present, SYSGEN reserves an extra block of CSR space to indicate a change to a new device type by adding eight to the rounded CFCB:

```
CFCB + 8 ⟶ CFCB
```

14.3.6 Rules for Configuration

The formulas described in Sections 14.3.4 and 14.3.5 reduce to the following maxims:

- Devices with fixed CSR addresses and fixed vector addresses must be attached according to the SYSGEN device table settings.

- Devices with floating CSR or vector addresses must be attached in the order in which they are listed in the SYSGEN device table.
- An 8-byte gap must be reserved between each different type of device that is located in floating CSR address space.
- An 8-byte gap must be reserved in floating CSR address space for each device type that has no controller in its configuration.
- An extra 8-byte gap must be reserved between the KW11C and the RX11 in floating CSR address space.

14.3.7 Example of a UNIBUS Configuration

This example shows the correct configuration for UNIBUS devices with floating CSR and vector addresses. Controllers flagged with an asterisk (*) are not supported by DIGITAL.

Controller	Vector(s)	CSR (first register)
1 DN11*	300	775200
1 DU11*	310	760040
1 DV11*	320	775000
1 DMC11	340	760100
2 DZ11s	350	760120
	360	760130
2 TS11s	224	772520
	370	772524
3 DR11Bs*	124	772410 (CSR is third register)
	400	772430
	410	760300
1 customer device	420 (or higher)	760320 (or higher)

When assigning floating vector addresses and registers to devices not supplied by DIGITAL, be sure to leave a generous gap between these addresses and those of DIGITAL devices, since subsequent VAX/VMS maintenance updates may add new devices to the SYSGEN device table.

CHAPTER 15

DEBUGGING A DEVICE DRIVER

DELTA and XDELTA are debugging tools that can be used to monitor the execution of user programs and the VAX/VMS operating system. When you link DELTA with a user image that runs in a nonprivileged process, DELTA is a user-mode debugging tool. When run in a privileged process, however, DELTA acts as a multimode debugger; it allows you to debug in user mode or to change to kernel mode for debugging. DELTA does not support debugging at elevated IPLs.

XDELTA is syntactically identical to DELTA but also allows you to debug code that executes at an elevated IPL. XDELTA is used for stand-alone debugging of driver code and the executive.

In the command syntaxes and dialogues contained in this chapter, red ink indicates the commands typed by the user and black ink indicates the system prompts and responses.

15.1 BOOTSTRAPPING THE SYSTEM WITH XDELTA

Under VAX/VMS, drivers are part of the operating system. You normally bootstrap the system with two boot flags set to allow you to debug with XDELTA. One flag causes the bootstrapping procedure to include XDELTA in the system. The other boot flag indicates a stop at a breakpoint in VAX/VMS initialization. Execution of the breakpoint instruction causes control to transfer to a fault handler located in XDELTA. The procedures for bootstrapping the system with XDELTA differ depending on which processor the operating system is running.

15.1.1 Bootstrapping the System with XDELTA on a VAX-11/780

In addition to the normal system bootstrap command files, the VAX/VMS console floppy diskette for a VAX-11/780 contains two command files that bootstrap the system with XDELTA:

- DMAXDT
- DBAXDT

To bootstrap the system with XDELTA, follow the procedures in the VAX-11/780 Software Installation Guide with two exceptions:

- Deposit the unit number of the device in R3.
- Specify one of the command files listed above instead of the command files listed in the installation guide.

The dialogue in Figure 15-1 is an example of bootstrapping the system with XDELTA on a VAX-11/780.

>>>DEPOSIT R3 0	Deposit the unit number 0 in R3.
>>>@DMAXDT	Boot the system from DMA0. The procedure boots the processor and prompts the user from SYSBOOT:
SYSBOOT>	Enter any SYSBOOT command. If you did not set or load system parameters with the USE command, the system uses the parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, you can set the system parameter BUGREBOOT to zero.
SYSBOOT> CONTINUE	Continue with the bootstrapping operation.

Figure 15-1: Bootstrapping the System with XDELTA on a VAX-11/780

15.1.2 Bootstrapping the System with XDELTA on a VAX-11/750

If the VAX/VMS operating system is running on a VAX-11/750, you must issue the following command in order to bootstrap the system with XDELTA:

```
>>>B[/f] device-name
```

Command Parameters and Qualifiers

B

The console BOOT command. See the VAX-11/750 Software Installation Guide for further details on this command.

/f

The 32-bit hexadecimal integer value loaded into R5 as an input value to VMB.EXE, the primary bootstrap program. The /f qualifier may have the following values:

Value	Meaning
f=0	Normal nonstop bootstrap (default)
f=1	Stop in SYSBOOT (equivalent to @DxyGEN on the VAX-11/780)
f=2	Include XDELTA with the system but do not take the initial breakpoint
f=6	Include XDELTA with the system and take the initial breakpoint
f=7	Include XDELTA with the system, stop in SYSBOOT and take the initial breakpoint at system initialization (equivalent to @DxyXDT on the VAX-11/780)

device-name
Indicates the name of the device that contains the volume to be bootstrapped. Specify the device name using the format ddcu (refer to the VAX-11/750 Software Installation Guide for a complete description of device name format). Both controller and unit identifiers must be specified; there are no defaults. If you do not use the device-name parameter, the /f qualifier is ignored.

The dialogue in Figure 15-2 is an example of bootstrapping the operating system with XDELTA on a VAX-11/750.

>>>B/7 DMA0	Bootstrap the system from DMA0. The command boots the processor and prompts the user from SYSBOOT.
SYSBOOT>	Enter any SYSBOOT commands. If you did not set or load system parameters with the USE command, the system uses the parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, you can set the system parameter BUGREBOOT to zero.
SYSBOOT> CONTINUE	Continue with the boostrapping operation.

Figure 15-2: Bootstrapping the System with XDELTA on a VAX-11/750

To bootstrap the system from the console TU58, see the VAX-11/750 Software Installation Guide. The console TU58 contains the command files DMAXDT and DBAXDT which are analogous to the files on the VAX-11/780 console floppy diskette.

15.1.3 Bootstrapping the System with XDELTA on a VAX-11/730

In addition to the normal system bootstrap command files, the VAX/VMS console DECtape for a VAX-11/730 contains two command files that bootstrap the system with XDELTA:

- DQAXDT
- DQ0XDT

To bootstrap a VAX-11/730 with XDELTA, follow the procedures outlined in the VAX-11/730 Software Installation Guide and specify one of the command files listed above. The dialogue in Figure 15-3 is a general example of boostrapping the system with XDELTA on a VAX-11/730.

When the boot device is DQA0, you can omit the first step in Figure 15-3 and execute the command procedure DQ0XDT:

```
>>> @DQ0XDT
```

>>>D/G/L 3 1	Deposit the unit number 1 in R3
>>>@DQAXDT	Boot the system from DQA1. The procedure boots the processor and prompts the user from SYSBOOT:
SYSBOOT>	Enter any SYSBOOT command. If you did not set or load any system parameters with the USE command, the system uses the system parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, you can set the system parameter BUGREBOOT to zero.
SYSBOOT> CONTINUE	Continue with the bootstrapping operation.

Figure 15-3: Bootstrapping the System with XDELTA on a VAX-11/730

15.1.4 Proceeding from the Initial Breakpoint

After being bootstrapped, the system displays its welcoming message and halts in XDELTA, as follows:

```
1 BRK AT nnnnnnnn
address/NOP
```

XDELTA is waiting for input. (XDELTA never issues explicit prompts.) Usually, you proceed from this point with the following command:

```
;P (RET)
```

All of the XDELTA commands are described in Section 15.10.

If the operating system halts with a fatal bugcheck, the system prints the bugcheck information on the console terminal. Then, because the system parameter BUGREBOOT was set to zero, XDELTA prompts. Bugcheck information consists of the following:

- Type of bugcheck
- Register values
- Dump of one or more stacks

PC and stack content indicate how an experimental driver crashed the system. You can then examine the system state further by issuing XDELTA commands.

15.2 LOADING THE DRIVER

Once the system is running, you can log in to the system as the system manager and load the experimental driver.

To load the driver, run SYSGEN and issue the appropriate LOAD and CONNECT commands. Figure 15-4 provides a sample dialogue.

The first SHOW command in Figure 15-4 causes SYSGEN to display the location of the device driver in system memory. You then define the device to the operating system. The second SHOW command causes SYSGEN to display the driver's location and the addresses of the device's DDB, CRB, IDB, and UCB.

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN>   LOAD DMA0:[YOUR.DIRECTORY]YRDRIVER.EXE

SYSGEN>   SHOW /DEVICE=YRDRIVER
  Driver     Start    End     Dev   DDB      CRB      IDB     Unit  UCB
YRDRIVER   80060E50 80061070

SYSGEN>   CONNECT YR /ADAP=3/VEC=%O274/CSR=%O776240

SYSGEN>   SHOW /DEVICE=YRDRIVER
  Driver     Start    End     Dev   DDB      CRB      IDB     Unit  UCB
YRDRIVER   80060E50 80061070
                            YRA 8005FDC0 80060B70 8005FE00
                                                        0 80060BB0

SYSGEN>   EXIT
```

Figure 15-4: Loading a Driver

15.3 INSERTING BREAKPOINTS IN THE SOURCE CODE

The SYSGEN command CONNECT calls controller initialization and unit initialization routines. To begin debugging the driver, you should ensure that the kernel mode debugging utility XDELTA gains control of the driver before these routines execute. This is accomplished by placing calls to the special system routine INI$BRK within the source code of either the controller or unit initialization routines. To call INI$BRK, give the following instruction:

```
JSB     G^INI$BRK
```

The INI$BRK routine contains two instructions:

```
BPT
RSB
```

When the processor executes the BPT instruction, XDELTA gains control and reports the address of the breakpoint:

```
1 BRK AT nnnnnnnn
```

You can use INI$BRK as a debugging tool and place calls to it within any part of the driver source code.

To determine the last driver PC before the breakpoint, examine the kernel stack. The stack register is register RE (hexadecimal format):

```
RE/address /address
```

Display RE to find the address of the current top of stack. Another display command (/) reveals the contents of the stack top, that is, the return address to the driver that called INI$BRK.

15.4 CALCULATING THE BASE OF DRIVER CODE

Before you debug the driver, it is a good idea to calculate the base address of driver code, as follows:

- Run SYSGEN and issue the SHOW/DEVICE command. The resulting display lists the location in nonpaged pool at which SYSGEN loaded the driver.

- Consult the load map for the driver (obtained at driver link time). The driver resides in two program sections (PSECTs):

  $$$105_PROLOGUE driver prologue table

  $$$115_DRIVER driver code

  The locations given in the driver code listing are offsets from $$$115_DRIVER. Thus, you can calculate the base address of the driver by adding the address at which the driver was loaded to the offset associated with the PSECT $$$115_DRIVER shown in the map.

If you do not have the load map, consult the driver prologue table in the driver listing. Look for the address of DPT_STORE_END, which generates a 2-byte entry that terminates the DPT. To get the base address of driver code, add the address of DPT_STORE_END + 2 to the address at which the driver was loaded. You can set an XDELTA base register to the base of driver code; Section 15.7 describes this procedure.

15.5 REQUESTING AN XDELTA SOFTWARE INTERRUPT

Once the controller and unit initialization routines complete execution, you will need to set breakpoints in order to debug the driver. You can set a breakpoint in the driver source code by inserting calls to INI$BRK, as described in Section 15.3. You can also invoke XDELTA to set breakpoints interactively by requesting an XDELTA software interrupt.

The procedures described in the following sections will issue a software interrupt to the processor at IPL 5. The IPL 5 interrupt service routine handles the interrupt by calling the routine INI$BRK, which in turn executes the first XDELTA breakpoint. XDELTA then issues the message:

```
1 BRK AT nnnnnnnn
address/NOP
```

15.5.1 Requesting an XDELTA Interrupt on a VAX-11/780

To request an XDELTA software interrupt on a VAX-11/780, issue the following commands at the console terminal:

```
$ CTRL/P
>>>HALT
>>>DEPOSIT/I 14 5
>>>CONTINUE
```

15.5.2 Requesting an XDELTA Interrupt on a VAX-11/750

To request an XDELTA software interrupt on a VAX-11/750, issue the following commands at the console terminal:

```
$ CTRL/P
>>>D/I 14 5
>>>C
```

The VAX-11/750 accepts only one-character commands.

15.5.3 Requesting an XDELTA Interrupt on a VAX-11/730

To request an XDELTA interrupt on a VAX-11/730, issue the following commands at the console terminal:

```
$ CTRL/P
>>>D/I 14 5
>>>C
```

The VAX-11/730 accepts only one-character commands.

15.6 LOOKING AT THE VECTOR JUMP TABLE

To gain familiarity with the I/O data base, you may wish to look for the address of the location in the channel request block that contains a JSB instruction to the driver's interrupt service routine. You can do this at a controller initialization breakpoint because one of the inputs is the IDB address. The procedures for locating the driver interrupt service routine on nondirect and direct vector UNIBUS adapters are shown below.

Nondirect Vector Procedure

```
R5/IDB-address  Q+10/ADP-address
Q+10/vector-table-address
Q+vector-address-in-hex/address-of-JSB-instruction-in-CRB
Q!JSB-instruction
```

Direct Vector Procedure

```
R5/IDB-address      Q+10/ADP-address
Q+10/vector-table-address
Q+vector-address-in-hex+2/address-of-JSB-instruction-in-CRB
Q!JSB-instruction
```

Finding the driver interrupt service routine address at the expected vector does not guarantee that an interrupt from the device will dispatch to the driver's interrupt service routine. If the device's physical vector is set to some other address, an interrupt from the device may dispatch to some other interrupt service routine, or dispatch to an unassigned vector.

See the SYSGEN device table shown in Chapter 14 for a list of vectors. Consult DIGITAL field service for help with any problem similar to the one described above.

15.7 SETTING AN XDELTA BASE REGISTER

During a driver debugging session, you can use an XDELTA relocation register as a base from which to examine driver code and set breakpoints within the driver. Use one of the methods outlined in Section 15.4.2 to determine the base address of driver code, then set a relocation register by issuing the following command:

```
driver-base-address,0;X  RET
```

This command sets relocation register X0 to the base of driver code. Now you can examine offsets into the code using X0 as a base:

```
X0 + offset/nnnnnnnn

     or

X0 + offset!instruction
```

XDELTA also uses the base register to display address values in the base register plus offset format. Suppose, for example, that your driver contains the code shown below:

```
        50   81   90   00D3   132 10$:     MOVB   (R1)+,R0
             10   13   00D6   133          BEQL   20$
        20   50   91   00D8   134          CMPB   R0,#^A/ /
             F6   19   00DB   135          BLSS   10$
   7A   8F   50   91   00DD   136          CMPB   R0,#^A/Z/
             F0   14   00E1   137          BGTR   10$
        82   50   90   00E3   138          MOVB   R0,(R2)+
             EB   11   00E6   139          BRB    10$
```

If base register 0 contains the base address of your driver, the following XDELTA dialogue is possible:

```
X0+D3,X0+E6!X0+D3/MOVB (R1)+,R0
X0+D6/BEQL     X0+E8
X0+D8/CMPB     R0,#20
X0+DB/BLSS     X0+D3
X0+DD/CMPB     R0,#7A
X0+E1/BGTR     X0+D3
X0+E3/MOVB     R0,(R2)+
X0+E6/BRB      X0+D3
```

To set breakpoints in driver code, give the command:

```
X0 + offset;B  RET
```

To display a driver instruction and set a breakpoint, add the instruction's offset to the base register, for example:

```
X0+1C!instruction    .;B  RET
```

The last XDELTA command sets a breakpoint at the displayed location. See Section 15.10 for a detailed discussion of XDELTA commands.

15.8 DESTROYING REGISTER CONTENTS

Since the driver frequently calls VAX/VMS I/O routines, you must be careful to anticipate the register usage of these routines. Most VAX/VMS common I/O support routines use R0 through R3 freely. A frequent driver bug is to load a value into R3 and expect to find it intact after a call to allocate or load UNIBUS adapter resources.

Other VAX/VMS I/O routines write into R4. In some cases, the use of R4 is obvious; for example, IOC$REQSCHANL writes the device's CRB address into R4. In other cases, you might not anticipate the use of R4.

For example, EXE$IOFORK saves the calling code's R4 in a fork block, and then writes the device's IPL into R4. Since the normal flow of events is that an interrupt service routine restores a driver with a JSB instruction and the driver then calls EXE$IOFORK which returns to the interrupt service routine, the instructions following the JSB in the interrupt service routine can only assume R5 is still untouched. The coding sequence is as follows:

```
MOVQ    UCB$L_FR3(R5),R3        ; Restore R3-R4.
JSB     @UCB$L_FPC(R5)          ; Restore the driver process.
.
.
.
```

Between these instructions, the interrupt service routine can make no assumptions about the contents of R0 through R4

```
.
.
.

POPR    #M^<R0,R1,R2,R3,R4,R5> ; Restore interrupt registers.
REI                            ; Return from the interrupt.
```

15.9 EXAMINING UCB, IRP, AND PSL

In addition to using XDELTA to debug drivers, you also can examine the contents of the unit control block and the associated I/O request packet.

It also is useful to examine the contents of the PSL at the time of a system failure. The PSL, for example, indicates the IPL at the time. When the system fails it prints the PSL and other register contents on the console terminal.

While the system is running, the following command can be used to examine the PSL in XDELTA:

```
RF+4/
```

That is, the PSL location is stored in the longword following the PC.

15.10 XDELTA COMMANDS

Table 15-1 summarizes XDELTA commands. The sections that follow detail the commands.

Table 15-1: XDELTA Command Summary

Command	Function
/	Open location (display contents in current mode)
!	Open location (display contents as instructions)
<RET>	Close current location
<LF>	Close current location; open next
<TAB>	Open location specified by current value
<ESC>	Display previous location
=	Display value of expression; set Q
+	Add
-	Subtract
space	Add
*	Multiply
@	Shift
%	Divide
,	Field separator
Q	Last quantity displayed
Rn	Register n
Xn	Base register n
Pn	Processor register n
G	Add ^X80000000 to subsequent or preceding value
H	Add ^X7FFE0000 to subsequent or preceding value
.	Current location
S	Execute one instruction, step into subroutine call
O	Execute one instruction, step over subroutine call (on CALLx, JSB, or BSBx)
;P	Proceed from breakpoint
;B	Set/clear/display breakpoint
;E	Execute command string
;G	Go to location and proceed
;X	Set base register
[B	Set byte mode
[W	Set word mode
[L	Set longword mode
[I	Set instruction mode
"	Set ASCII mode
'string'	Deposit string at current dot, autoincrementing dot. A single quote terminates a string; every <RET> and <LF> typed will be stored.

15.10.1 Values and Expressions

All numeric values are interpreted in hexadecimal radix. Expressions are strings of alternating values and binary operators, where the first and last items in the string are always values, as in the following example:

```
G4A32 + 24 - .
```

Trailing operators are ignored.

15.10.2 Special Symbols

XDELTA defines the following special symbols:

Current location; set by slash (/), exclamation point (!) and TAB operations

Q	Last quantity displayed
X0⟶XF	Base registers; used for remembering values
R0⟶RF	General register names
P0⟶Pnn	Internal processor registers
RF+4	PSL
G	^X80000000; prefix for system space addresses; for example, G2E is equivalent to ^X8000002E
H	^X7FFE0000; prefix for control region prefix; for example, H2E is equivalent to ^X7FFE002E

15.10.3 Operators

XDELTA recognizes the following operators:

+ or space	add
-	negate, subtract
*	multiply
%	divide
@	shift (arithmetic)

Evaluation of expressions is left to right with no precedence.

15.10.4 Open and Display Value Command

Syntax

```
address-expression/old-value [new-value-expression]
```

Type an address expression followed by a slash (/) character. XDELTA displays the contents of the location (old-value above) followed by a space character. You can change the value at the location by typing a new value and then pressing RETURN. If you press RETURN without preceding it with a value, the old contents remain unchanged.

The display and the value deposited default to longword hexadecimal values. The length can be changed to byte or word with the set mode commands.

A slash preceded by a null address expression uses the displayed value (Q) as the address value. This feature is convenient for following address linked chains.

```
address-expression/old-value /old-value /old-value
```

15.10.5 Display Instruction Command

Syntax

```
address-expression!decoded-instruction
```

Type an address-expression followed by an exclamation point (!). XDELTA displays the contents of memory as a VAX-11 MACRO instruction starting with the address you specify.

XDELTA does not make any distinction between reasonable . and unreasonable instructions or instruction streams; the decoding always begins at the specified address. The display instruction command does not allow you to modify the displayed location. The command sets a flag that causes subsequent close and display next or indirect location commands to perform instruction decoding. You can reset the flag with the open and display value command.

Whenever an address appears as an instruction operand, XDELTA sets the last quantity displayed (Q) to that address. XDELTA changes Q only for operands that use program counter or branch displacement addressing modes; Q is not altered for literal and register addressing modes. This feature is useful for following branches, as shown below.

```
address-expression!BRW address-2  !instruction-at-address-2
```

15.10.6 Close and Display Next Location Command

Syntax

```
address/old-value
```

Press LINE FEED. XDELTA closes the current open location, then opens and displays the value in the next location according to the current display mode.

If instruction display is the current mode, XDELTA does not deposit a value in the open location. The next location is the first location after the instruction currently displayed. If value display is the current mode, you can deposit a value into the open location. In this case, the next location is the current location incremented by the current data width (byte, word, or longword).

15.10.7 Display Range Command

Syntax

```
start-addr-expression,end-addr-expression/contents-of-start

    or

start-addr-expression,end-addr-expression!contents-of-start
```

Type two address expressions separated by a comma and followed by a slash (/) or exclamation point (!) character. XDELTA displays the range of addresses using the specified display mode (value or instruction). If you specify instruction display, XDELTA decodes one more more instructions. Otherwise, XDELTA displays the contents of each location in the current data type (byte, word, or longword).

15.10.8 Indirect Command

Syntax

```
TAB
address/old-value
```

Press TAB. XDELTA uses the last quantity displayed (Q) as an address and displays that address and its contents using the current display mode. This command opens locations in the same way as the slash (/) and exclamation point (!) commands, but prints the information on a new line and displays the address value before showing the address's contents.

15.10.9 Display Previous Location Command

Syntax

```
ESC
address/old-value
```

Press ESC. Unless the current display mode is instruction, XDELTA decreases the location counter by the current data width, and displays the contents of the resulting location using the current data width and type. This command is ignored in instruction display mode.

15.10.10 Show Value Command

Syntax

```
expression=value-of-expression
```

Type an expression followed by an equal sign (=). The expression can be composed of a series of values and operators from the set of operators listed in the command summary. XDELTA shows the value of the expression according to the current display data type. The last quantity (Q) is set to the value of the computed expression.

15.10.11 Step Instruction Command

Syntax

```
S
```

Type an S. XDELTA causes one instruction to be executed, then displays the address of the next instruction and decodes that instruction.

This command also sets a flag that causes subsequent close and display next or indirect location commands to perform instruction decoding. The open and display value command resets the flag.

If the next instruction is BSBB, BSBW, JSB, CALLG, or CALLS, this command steps into the subroutine and displays the first instruction within the routine.

15.10.12 Step Instruction Over Subroutine Command

Syntax

```
O
```

Type an O. XDELTA causes one instruction to be executed, then displays the address of the next instruction and decodes that instruction.

This command also sets a flag that causes subsequent close and display next or indirect location commands to perform instruction decoding. The open and display value command resets the flag.

If the next instruction is BSBB, BSBW, JSB, CALLG or CALLS, XDELTA executes the entire subroutine and displays the instruction that immediately follows the subroutine call; that is, this command steps over subroutines.

15.10.13 Setting Breakpoints

Syntax

```
address-expression;B (RET)
```

Type an address followed by a semicolon (;) the letter B, then press RETURN. XDELTA sets a breakpoint at the specified location and assigns it the first available breakpoint number.

Alternate syntax:

```
address-expression,n;B (RET)
```

Type an address, followed by a comma, a single digit between 2 and 8, a semicolon (;), the letter B, and then press RETURN. XDELTA sets a breakpoint at the specified location and assigns it the specified breakpoint number. Breakpoint 1 is reserved for INI$BRK.

Before XDELTA executes the instruction as a breakpoint, it suspends normal instruction processing, sets a flag that causes subsequent close and display next or indirect location commands to perform instruction decoding, and displays the following message:

```
n BRK at address
address/decoded-instruction
```

You may now enter XDELTA commands. You can reset the flag that controls instruction display mode by issuing the open and display value command.

15.10.14 Clearing Breakpoints

Syntax

```
0,n;B (RET)
```

Type zero (0), followed by a comma, a single digit between 2 and 8, a semicolon (;), the letter B, and then press RETURN. XDELTA clears the specified breakpoint. Never clear breakpoint 1.

15.10.15 Displaying Breakpoint List

Syntax

```
;B (RET)
```

Type a semicolon (;) followed by the letter B. XDELTA shows the current setting of all breakpoints. For each breakpoint, XDELTA displays the following information:

- Breakpoint number
- Address at which the breakpoint is set
- Display address (for complex breakpoints; see Section 15.10.19)
- Command string address (for complex breakpoints)

15.10.16 Setting Base Registers

Syntax

```
address-expression,n;X (RET)
```

Type an expression followed by a comma (,), a single digit between 0 and D (hexadecimal), a semicolon (;), and the letter X. XDELTA assigns the specified expression to the base register selected by n. XDELTA confirms that the base register is set by displaying the value deposited in the base register.

Whenever XDELTA displays an address closely located to an address stored in a base register, XDELTA displays the base register identifier (Xn) followed by an offset that gives the address's location in relation to the address stored in the base register. For example, if base register 2 contains 800D046A and the address XDELTA needs to display is 800D052E, XDELTA displays X2+C4. XDELTA computes relative addresses for opened or displayed locations and addresses that are instruction operands.

XDELTA displays an address in base register plus offset format to a distance of 800 (hex) from the base register. If the address falls outside this range, XDELTA displays it as a hexadecimal value. Sections 15.10.22 and 15.10.23 describe several predefined base registers.

15.10.17 Proceeding from Breakpoints

Syntax

```
;P (RET)
```

Type a semicolon (;) followed by the letter P and then press RETURN. XDELTA continues executing at the current PC.

15.10.18 Loading PC and Continuing

Syntax

```
address-expression;G (RET)
```

Type an address, a semicolon, and G, then press RETURN. XDELTA loads the address into PC and continues executing at the new PC.

15.10.19 Display Mode Control

Syntax

```
[B    Byte width
[W    Word width
[L    Longword width
[I    Instruction display (using longword width)
"     ASCII display (using current width)
```

Type a left square bracket ([) followed by one of the letters B, W, or L to change the current display width to byte, word, or longword respectively. The default value is longword. The setting remains in effect until another display mode control command is given. For example, the following command displays the least significant byte contained at the specified address and deposits the new value to that byte only.

```
address-expression [B/ old-value new-value
```

Type a left square bracket ([) followed by the letter I to change the current display mode to instruction format. This command is equivalent to the exclamation point (!) command and, similarly, is canceled by typing a slash (/) or a double quotation mark ("). Instruction mode sets display mode storage units to longword values. For an example of instruction display, see Section 15.10.5.

You can display contents of memory locations in ASCII characters by typing an address expression followed by a double quotation mark (").

```
address-expression" old-value-in-ASCII
```

Pressing LINE FEED displays the next location in ASCII.

The display mode remains set to ASCII until the next slash (/) or exclamation point (!) command. At this point, the display mode reverts to hexadecimal. Width remains unchanged.

15.10.20 The EXECUTE STRING Command

Syntax

```
address-expression;E (RET)
```

Type an address expression followed by a semicolon, the letter E, then press RETURN. This command executes the ASCII commands found at the specified address expression. If you terminate the ASCII commands with a semicolon followed by the letter P, XDELTA will proceed with program execution. If you terminate the string with null (1 byte of 0), XDELTA waits for a new command.

To create command strings, open the address of the start of the string and deposit ASCII text as follows:

```
address/old-contents 'XDELTA-command' RET
```

You can use any XDELTA command, including RETURN, LINE FEED, and TAB.

To terminate the string with a null, follow the above command with:

```
./old-contents 0 RET
```

You can deposit command strings into nonpaged system patch space. To determine the size of patch space and its starting address, locate the symbol PAT$A_NONPGD in the system map file (SYS$SYSTEM:SYS.MAP). This symbol contains a descriptor of the address and size of patch space remaining in the system, as shown below:

```
PAT$A_NONPGD::

     .LONG     size-in-bytes
     .LONG     patch-space-start-address
```

You can also preassemble command strings with your experimental driver. Locate the addresses of these strings as you would any other address within your driver.

15.10.21 Setting Complex Breakpoints

Syntax

```
address-expression,n,display-addr-expression,command-string-address;B RET
```

Type an address expression, followed by a comma, a single digit between 2 and 8, another address expression, and the address of a command string. The first address is the breakpoint address; the digit equals the breakpoint number. XDELTA shows the contents of the display address in the current display mode when the breakpoint is reached. The command string address specified in the last command parameter executes after automatic display.

15.10.22 XDELTA Stored Commands

XDELTA contains two predefined command strings whose addresses are contained in base registers XE and XF. You can use these commands during general system debugging as well as driver debugging; they perform the following functions:

XE — Use the value of base register X0 as a page frame number and display the PFN data base for that page.

XF — Set base register X0 to the value (PFN) in R0 and perform the same function as XE.

You must initialize the stored commands to set the relocation registers they use (X6-XD). Issue the following commands:

```
XE;E RET
XF;E RET
```

Now you can use the stored commands to obtain the following information about a page frame number:

- Specified physical page number (PFN)
- PFN state
- PFN type
- PFN reference count
- PFN backward link/working set list index
- PFN forward link/share count
- Page table entry (PTE) pointer to PFN
- PFN backing store address
- Virtual block number in process swap image

15.10.23 Stored Base Registers

XDELTA defines two base registers useful in system debugging: X4 and X5. Base register X4 corresponds to the global symbol SCH$GL_CURPCB. This symbol contains the address of the current process's software process control block (PCB). Base register X5 corresponds to the global symbol SCH$GL_PCBVEC, which contains the starting address of the list of PCB slots.

15.11 DELTA

DELTA is a debugging tool that can be linked with a user program to examine that program's execution. To link and run DELTA, issue the following commands:

```
$ LINK program-name
$ DEFINE LIB$DEBUG SYS$LIBRARY:DELTA
$ RUN/DEBUG program-name
```

DELTA accepts all the XDELTA commands, plus two additional commands described in the following sections.

15.11.1 The EXIT Command

Syntax

```
EXIT (RET)
```

Typing EXIT causes DELTA to return control to the command interpreter.

15.11.2 Examining and Modifying Locations in Process Space

Syntax

```
process_id:address_expression/old_contents
```

DELTA displays the current contents at the specified address expression within the specified process. The modify flag controls the ability to modify locations opened by this command. To examine the flag, type:

```
;M (RET)
```

Modify access is inhibited by default (M=0).

To open, examine and change a location, type the commands:

```
1;M (RET)
process_id:address_expression/old_contents new_contents
```

15.12 GUIDELINES FOR DEBUGGING DEVICE DRIVERS

The following sections discuss errors commonly made during debugging sessions and describe additional debugging techniques.

15.12.1 References to System Addresses

References by drivers to system addresses within the executive must use general addressing (G^) mode. For example, use

```
JSB     G^INI$BRK
```

15.12.2 Opening Device Registers in XDELTA

References to 16-bit device registers must be word instructions; references to 8-bit device registers must be byte instructions. These restrictions apply to the XDELTA EXAMINE command; therefore, be sure to set the correct mode control before examining device registers. For example, if the address of the device CSR is in R4, give the following command:

```
R4/csr_address [W/csr_contents
```

15.12.3 Incorrect References to Device Registers

A common driver error is to access a nonexistent device register or to access the correct register with an instruction of incorrect word length. On VAX-11 processors that use direct vector interrupts, these references cause a fatal machine check exception. On VAX-11 processors using nondirect vector interrupts, these references cause a UNIBUS adapter error interrupt. The system logs the adapter error and continues. When debugging a device driver, it is a good idea to catch this type of driver error as early as possible. Set an XDELTA breakpoint at the place in the system where it detected a UNIBUS adapter error interrupt. Follow the steps outlined below:

- Consult the system map file. Read the value of EXE$DW780_INT.
- Enter XDELTA and set a breakpoint at the address of EXE$DW780_INT. When a UNIBUS adapter error interrupt occurs, XDELTA executes the breakpoint at EXE$DW780_INT.

- Examine the stack as follows:

```
RE/current_stack_pointer/saved_R2 (LF)
                         saved_R3 (LF)
                         saved_R4 (LF)
                         saved_R5 (LF)
                         saved_PC (LF)
                         saved_PSL
```

In many cases, the saved PC on the stack is the address of the instruction that caused the error. In other cases (for example, when the offending instruction is executed at IPL 31), the saved PC is not the address of this instruction but an address some number of instructions later, when the system actually services the interrupt.

15.12.4 XDELTA and System Failures

Driver errors can cause the operating system to suspend activity in such a way that you cannot invoke XDELTA. In this case, the only recourse is to induce a system failure. Follow the procedure described in the VAX/VMS System Dump Analyzer Reference Manual; the system will signal a fatal bugcheck.

To gain control in XDELTA following a fatal bugcheck, stop in SYSBOOT while initializing the system and set the BUGREBOOT parameter to zero. The system will stop in XDELTA, thereby allowing you to examine the device unit control block and other driver data to determine the driver error.

Another, more thorough, way to determine the cause of a system failure is to leave the BUGREBOOT parameter set to 1, allow the system to reboot, and then invoke the System Dump Analyzer (SDA) to examine the condition of the I/O data structures at the time of the fatal bugcheck. The VAX/VMS System Dump Analyzer Reference Manual provides detailed information on fatal bugcheck stack format and how SDA can help debug a device driver.

PART III
APPENDIXES

APPENDIX A

THE I/O DATA BASE

The I/O data base is a collection of control blocks allocated in nonpaged system memory. This data base provides the following information:

- I/O request packets describing in-progress I/O requests
- Device characteristics of each device type
- Number and type of each device unit
- Current activity on each device unit
- External entry points to all device drivers
- Entry points for controller and device unit initialization routines
- Interrupt vector dispatch code
- Addresses of device registers
- UNIBUS adapter data path bit map

Much of this I/O data base is created and used only by VAX/VMS routines. Other parts are the primary source of data for the device drivers. The sections that follow identify all I/O data base control blocks and describe their fields. Field descriptions are in the order in which they appear in the control blocks. Driver code must consider fields flagged with asterisks (*) as read-only fields. Fields marked by "spare" or "unused" are reserved for future use by DIGITAL unless otherwise specified.

A.1 CONFIGURATION CONTROL BLOCK (ACF)

The configuration control block is used by the SYSGEN autoconfiguration facility to describe the device it is adding to the system. Device drivers may gain access to this data structure only if they have specified a unit-delivery routine in the driver prologue table and only when that routine is executing. Under certain conditions, the information stored in the configuration control block may be useful to a unit-delivery routine.

The fields described in the configuration control block are illustrated in Figure A-1 and described in Table A-1.

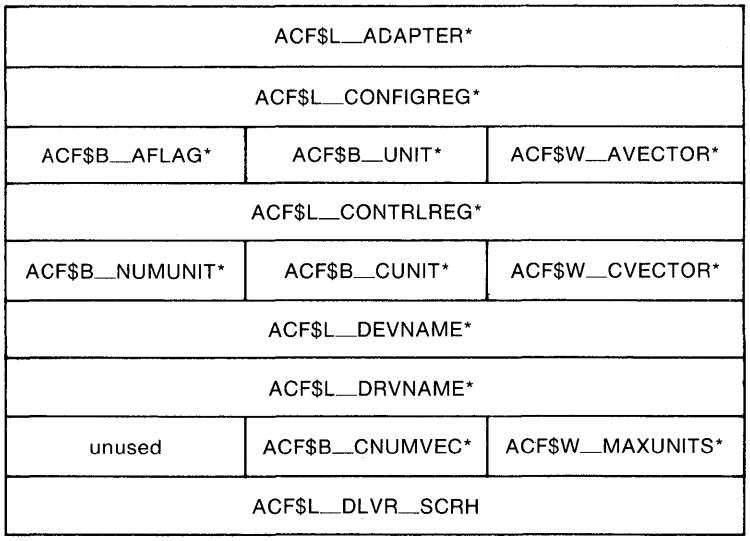


Figure A-1: Configuration Control Block

Table A-1: Contents of the Configuration Control Block

Field Name	Contents
ACF$L_ADAPTER*	Address of the adapter control block for the adapter currently being configured.
ACF$L_CONFIGREG*	Address of the configuration register for the adapter currently being configured.
ACF$W_AVECTOR*	Offset from the base of the system control block (SCB) to the interrupt vector of the adapter currently being configured.
ACF$B_AUNIT*	Adapter unit number of the device or controller currently being configured.
ACF$B_AFLAG*	Flags associated with the automatic configuration operation. Flags defined in this field include the following: ACF$V_RELOAD — Reloading driver code ACF$V_CRBBLT — CRB and IDB for this device already built ACF$V_SCBVEC — CVECTOR is an offset into the SCB ACF$V_NOLOAD_DB — Do not load I/O data base, only load driver ACF$V_SUPPORT — VAX/VMS supported device

(continued on next page)

Table A-1 (Cont.): Contents of the Configuration Control Block

Field Name	Contents
ACF$L_CONTROLREG*	Address of control/status for the controller currently being configured
ACF$W_CVECTOR*	Offset in the adapter control block vector table to the longword that contains the transfer address of the interrupt vector used by the controller currently being configured (if ACF$V_SCBVEC is not set). If ACF$V_SCBVEC is set, this field is the offset from the SCB base to the interrupt vector of the controller currently being configured.
ACF$B_CUNIT*	Unit number of the device currently being configured.
ACF$B_NUMUNIT*	Number of units to be configured for the controller currently being configured.
ACF$L_DEVNAME*	Address of a counted ASCII string that gives the name of the controller currently being configured.
ACF$L_DRVNAME*	Address of a counted ASCII string that gives the driver name for the controller currently being configured.
ACF$W_MAXUNITS*	Maximum number of units that can be connected to the controller currently being configured.
ACF$BCNUMVEC*	Number of interrupt vectors to configure for the controller currently being configured.
ACF$L_DLVR_SCRH	Field available for use by the unit delivery routine. SYSGEN never alters this field.

A.2 ADAPTER CONTROL BLOCK (ADP)

Each MASSBUS and UNIBUS adapter configured in the system is represented to VAX/VMS and driver routines by an adapter control block. The adapter control block stores adapter-specific static and dynamic data such as the adapter CSR address and map register wait queues.

The fields of the ADP for a UNIBUS adapter are illustrated in Figure A-2 and described in Table A-2.

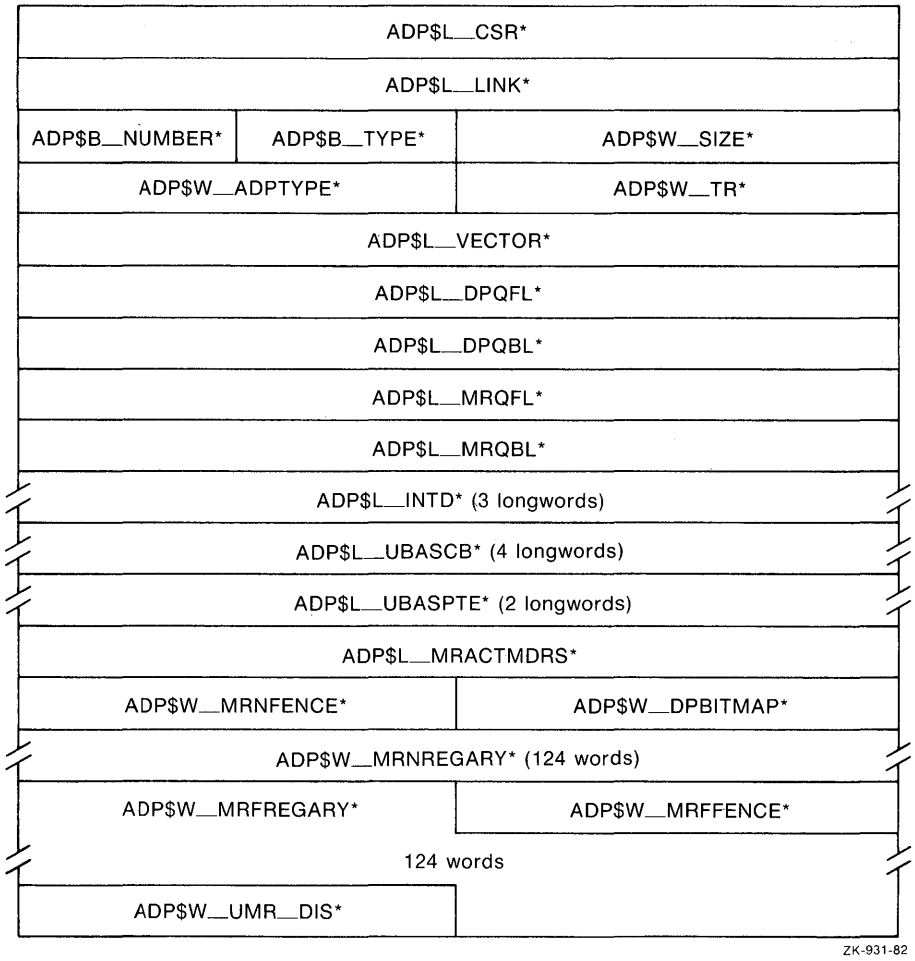


Figure A-2: Adapter Control Block

Table A-2: Contents of Adapter Control Block

Field Name	Contents
ADP$L_CSR*	Virtual address of the adapter configuration register. The CPU initialization sets this field. The configuration register marks the base of adapter register space, an area that contains data path registers, map registers, or any other registers appropriate to the implementation of the adapter.
ADP$L_LINK*	Address of next ADP. The CPU initialization routine writes this field. A value of 0 indicates that this is the last ADP.
ADP$W_SIZE*	Size of the ADP control block. The CPU initialization routine writes this field when the routine creates the ADP. For the UNIBUS adapter, this includes the UNIBUS interrupt service code and device vector table.

(continued on next page)

Table A-2 (Cont.): Contents of Adapter Control Block

Field Name	Contents
ADP$B_TYPE*	Type of control block. The CPU initialization routine writes the symbolic constant DYN$C_ADP into this field when the routine creates the ADP.
ADP$B_NUMBER*	Number of this type of adapter (for example, the number for a third MASSBUS adapter is 2). The CPU initialization routine writes this field when the routine creates the ADP.
ADP$W_TR*	Nexus number of the adapter. The CPU initialization routine writes this field when the routine creates the ADP. The driver loading procedure compares the nexus number specified in a CONNECT command with this field of each ADP in the system to determine to which adapter a device is attached.
ADP$W_ADPTYPE*	Type of adapter. The CPU initialization routine writes the symbolic constant AT$_UBA into this field when the routine creates an ADP for a UNIBUS adapter. AT$_MBA is the type code for a MASSBUS adapter.
ADP$L_VECTOR*	Address of vector table. The table is 512 bytes of longword vectors that correspond to UNIBUS device interrupt vectors (0-%0777). On VAX-11 processors that handle direct vector interrupts, ADP$L_VECTOR points to the second (or third) page of the system control block (SCB). The CPU uses this page when it dispatches the device interrupt to the driver interrupt service routine. Each vector entry that corresponds to a vector in use contains the address of the controller's interrupt dispatcher (CRB$L_INTD). On VAX-11 processors that handle nondirect vector interrupts, ADP$L_VECTOR points to a page allocated from nonpaged pool. Each longword in the page that corresponds to a vector in use contains the address of the controller's interrupt dispatcher (CRB$L_INTD+2). When the UNIBUS adapter interrupts on behalf of a UNIBUS device, the UNIBUS adapter interrupt service routine saves R0 through R5, determines the vector address of the interrupting device, indexes into the vector table, and executes the instruction at CRB$L_INTD+2. For both types of VAX-11 processor, vector table entries that correspond to unused vectors contain the address of a UNIBUS adapter unexpected interrupt service routine.

(continued on next page)

Table A-2 (Cont.): Contents of Adapter Control Block

Field Name	Contents
ADP$L_DPQFL*	Data path wait queue forward link. IOC$REQDATAP and IOC$RELDATAP read and write this field. When a driver fork process requests a buffered data path and none is currently available, IOC$REQDATAP saves driver context in the device's UCB fork block, inserts the fork block address in the data path wait queue, and suspends the driver fork process. When another driver calls IOC$RELDATAP to release a buffered data path, the routine dequeues a UCB fork block address from the data path wait queue, allocates a data path to the driver, and reactivates that driver fork process.
ADP$L_DPQBL*	Data path wait queue backward link. IOC$REQDATAP and IOC$RELDATAP read and write this field.
ADP$L_MRQFL*	Map register wait queue forward link. IOC$REQMAPREG and IOC$RELMAPREG read and write these fields. When a driver fork process requests a set of map registers and the set is not currently available, IOC$REQMAPREG saves driver fork context in the device's UCB fork block, inserts the fork block address in the map register wait queue, and suspends the driver fork process. When another driver calls IOC$RELMAPREG to release a set of map registers, the routine dequeues a UCB fork block address from the map register wait queue, allocates the requested set of map registers to the driver, and reactivates that driver fork process.
ADP$L_MRQBL*	Map register wait queue backward link. IOC$REQMAPREG and IOC$RELMAPREG read and write this field.
ADP$L_INTD*	Interrupt transfer vector. When a device attached to the UNIBUS adapter requests a hardware interrupt, the processor transfers control to the ADP$L_INTD field of the UNIBUS adapter's control block. The field contains code that dispatches the interrupt to the proper driver interrupt service routine. **The interrupt transfer vector is only used for UNIBUS adapters that directly generate interrupts.**
ADP$L_UBASCB*	Series of four longwords that contain system control block entry values, one for each bus request (BR) level or interrupt vector. The UNIBUS adapter power failure recovery procedure uses these values.

(continued on next page)

Table A-2 (Cont.): Contents of Adapter Control Block

Field Name	Contents
ADP$L_UCBSPTE*	Page table entry values for the base of UNIBUS adapter register space and the base of UNIBUS I/O register space. These values are used during UNIBUS adapter power failure recovery.
ADP$L_MRACTMDRS*	The number of active map register descriptors in the arrays pointed to by ADP$L_MRNREARY and ADP$L_MRFREGARY. IOC$REQMAPREG and IOC$RELMAPREG use these fields when allocating and deallocating UNIBUS map registers.
ADP$W_DPBITMAP*	Data path allocation bit map. IOC$REQDATAP and IOC$RELDATAP read and write this field. The CPU initialization routine sets the bit map to show as available all the buffered data paths supported by the UNIBUS adapter. The state of each of the available buffered data paths (whether in use or available) is recorded in the data path allocation bit map. One data path corresponds to each bit in the field. If a bit is clear, the related data path is currently allocated to a driver fork process.
ADP$W_MRNFENCE*	Contains negative one. This field acts as a boundary marker for the array specified by ADP$L_MRNREGARY.
ADP$W_MRNREGARY*	Map register "number of registers" array of 124 words. The number of words, or cells, that are active in this array is contained in ADP$L_MRACTMDRS. Each active cell gives a number of free map registers. For each active cell in this array, there is a corresponding first free map register number in the array specified by ADP$W_MRFREGARY. Together, these values give the base map register and number of free map registers for a block of free map registers. This information is used to allocate and deallocate UNIBUS map registers.
ADP$W_MRFFENCE*	Word that contains negative one. This field acts as a boundary marker for the array specified by ADP$W_MRFREGARY.
ADP$W_MRFREGARY*	Map register "first register" array of 124 words. The number of currently active cells in this array is contained in ADP$L_MRACTMDRS. Each active cell gives a number of the first free map register within a block of free map registers. For each active cell in this array, there is a corresponding cell in the number of registers array that gives a number of free map registers. Together, these values give the base map register and number of free map registers for a block of free map registers. This information is used to allocate and deallocate UNIBUS map registers.

(continued on next page)

Table A-2 (Cont.): Contents of Adapter Control Block

Field Name	Contents
ADP$W_UMR_DIS*	The number of disabled map registers. During system initialization, some map registers may be disabled so that their corresponding UNIBUS addresses can be accessed directly through backplane interconnect physical addresses.

A.3 CHANNEL CONTROL BLOCK (CCB)

When a process assigns an I/O channel to a device unit with the Assign I/O Channel system service, EXE$ASSIGN locates a free block among the process's preallocated channel control blocks. EXE$ASSIGN then writes a description of the device attached to the channel in the CCB.

The fields of a channel control block are illustrated in Figure A-3 and described in Table A-3.

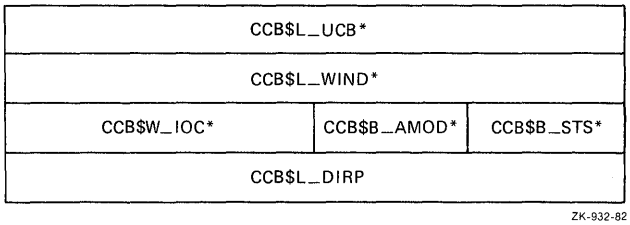


Figure A-3: Channel Control Block

Table A-3: Contents of Channel Control Block

Field Name	Contents
CCB$L_UCB*	Address of the unit control block of the assigned device unit. EXE$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 a window control block (WCB) for a file-structured device assignment. This field is written by an ACP and read by EXE$QIO. A file-structured device's ACP creates a window control block when a process accesses a file on a device assigned to a process channel. The window control block maps the virtual block numbers of the file to a series of physical locations on the device.
CCB$B_STS*	Channel status.

(continued on next page)

Table A-3 (Cont.): Contents of Channel Control Block

Field Name	Contents
CCB$B_AMOD*	Access mode plus 1 of the process at the time of the channel assignment. EXE$ASSIGN writes the process access mode value into this field.
CCB$W_IOC*	Number of outstanding I/O requests on the channel. EXE$QIO increases this field when it begins to process an I/O request that specifies the channel. During I/O postprocessing, the kernel mode AST routine decrements this field. Some FDT routines and EXE$DEASSIGN read this field.
CCB$L_DIRP*	Address of deaccess I/O request packet. 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.

A.4 CHANNEL REQUEST BLOCK (CRB)

The activity of each controller in a configuration is described in a channel request block. This control block contains pointers to the wait queue of drivers ready to gain access to a device through the controller. It also stores the entry points to the driver's interrupt service routines and device/controller initialization routines.

The fields of the channel request block are illustrated in Figure A-4 and described in Table A-4.

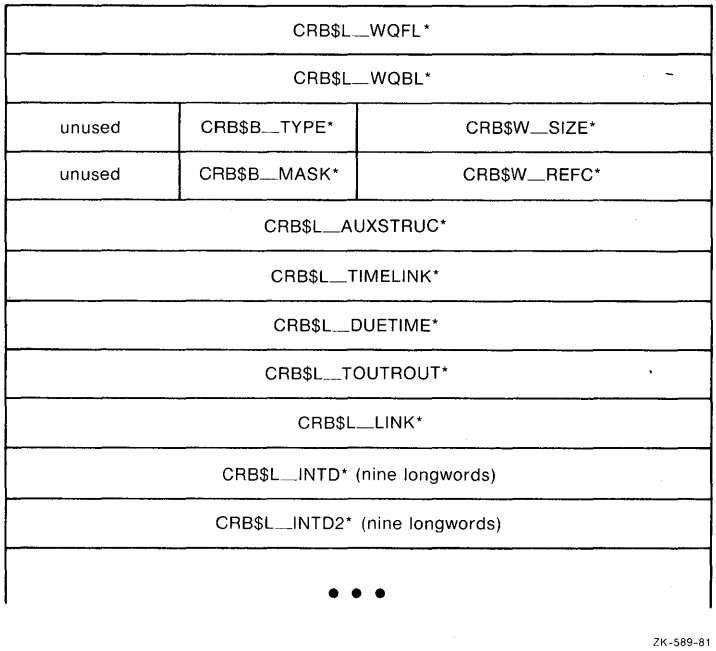


Figure A-4: Channel Request Block

Table A-4: Contents of Channel Request Block

Field Name	Contents
CRB$L_WQFL*	Controller data channel wait queue forward link. IOC$REQxCHANx and IOC$RELxCHAN insert and remove driver fork block addresses in this field. A channel wait queue contains addresses of driver fork blocks that record the context of suspended drivers waiting to gain control of a controller data channel. If a channel is busy when a driver requests access to the channel, IOC$REQxCHANx suspends the driver by saving the driver's context in the device's UCB fork block and inserting the fork block address in the channel wait queue. When a driver releases a channel because an I/O operation no longer needs the channel, IOC$RELxCHAN dequeues a driver fork block, allocates the channel to the driver, and reactivates the suspended driver fork process. If no drivers are awaiting the channel, IOC$RELxCHAN clears the channel busy bit.
CRB$L_WQBL*	Controller channel wait queue backward link. IOC$REQxCHANx and IOC$RELxCHAN read and write this field.
CRB$W_SIZE*	Size of the CRB. The driver loading procedure writes this field when the procedure creates the CRB.
CRB$B_TYPE*	Type of control block. The driver loading procedure writes the symbolic constant DYN$C_CRB into this field when the procedure creates the CRB.
CRB$W_REFC*	Unit control block reference count. The driver loading procedure increases the value in this field each time the procedure creates a UCB for a device attached to the controller.
CRB$B_MASK*	Mask that describes the status of the controller. At present, only one bit, CRB$V_BSY, is defined in this field. IOC$REQxCHANx 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$RELxCHAN clears the busy bit if no driver is waiting to acquire the channel.
CRB$L_AUXSTRUC*	Address of an auxiliary data structure that the device driver uses to store special controller information. A device driver that wishes to use this field can contain a controller initialization routine that allocates a block of nonpaged dynamic memory and sets this field to point to it. Use of this field is reserved to DIGITAL.

(continued on next page)

Table A-4 (Cont.): Contents of Channel Request Block

Field Name	Contents
CRB$L_TIMELINK*	Forward link in a queue of channel request blocks waiting for periodic wakeups. This field points to the CRB$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$L_DUETIME*	The time in seconds, relative to EXE$GL_ABSTIM, that the next periodic wakeup associated with this channel request block is to be delivered. Compute this value by raising IPL to IPL$_POWER, adding the desired number of seconds to the contents of EXE$GL_ABSTIM, and storing the result in this field. Use of this field is reserved to DIGITAL.
CRB$L_TOUTROUT*	The address of a routine to be called when the periodic wakeup associated with this CRB becomes due. The routine must compute and reset the value in CRB$L_DUETIME if another periodic wakeup request is desired. Use of this field is reserved to DIGITAL.
CRB$L_LINK*	Address of secondary CRB (for MASSBUS devices only). This field is written by the driver loading procedure and read by IOC$REQSCHANx and IOC$RELSCHAN.
CRB$L_INTD*	Interrupt transfer vector. The driver prologue table in every driver for an interrupting device specifies the address of a driver interrupt service routine. The driver loading procedure writes two instructions in this field: PUSHR #^M<R0,R1,R2,R3,R4,R5> JSB @#^driver_isr_address Direct vector UNIBUS adapters transfer control to CRB$L_INTD, which causes the processor to execute the PUSHR instruction to save R0 through R5 on the stack. Next, the processor executes the JSB instruction to transfer control to the driver interrupt service routine. On nondirect vector UNIBUS adapters, the UNIBUS adapter interrupt service routine transfers control to CRB$L_INTD+2, which contains the JSB instruction to the driver interrupt service routine. Since the UNIBUS adapter service routine has already saved R0 through R5, the PUSHR instruction is bypassed. The CRB$L_INTD field is nine longwords long. Figure A-5 and Table A-5 describe the contents of the rest of block.

(continued on next page)

Table A-4 (Cont.): Contents of Channel Request Block

Field Name	Contents
CRB$L_INTD2*	Second interrupt transfer vector for devices with multiple interrupt vectors. If the driver prologue table in a device driver specifies the address of a second driver interrupt service routine, the driver loading procedure creates a CRB long enough to contain two INTDx fields of nine longwords each. The first two longwords of the CRB$L_INTD2 field contain a PUSHR and JSB instruction to the second driver interrupt service routine. There are as many interrupt transfer vector blocks as there are device vectors. The number of device vectors is determined by the value specified in the /NUMVEC= qualifier to the SYSGEN command CONNECT.

The interrupt transfer vector blocks contained in the CRB store executable code, driver entry points, and UNIBUS adapter information. The fields of the CRB$L_INTD block are illustrated in Figure A-5 and described in Table A-5.

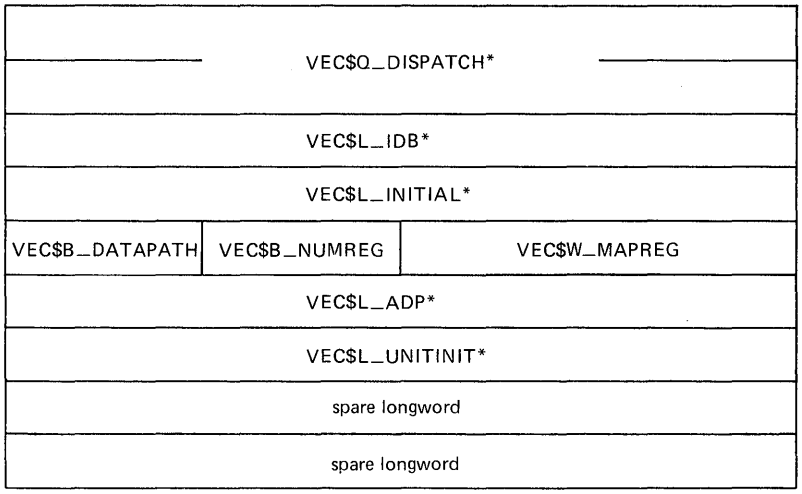


Figure A-5: Contents of CRB$L_INTD

Table A-5: Fields of CRB$L_INTD

Field Name	Contents
VEC$Q_DISPATCH*	Contains the two interrupt dispatching instructions described above in the CRB$L_INTD field. This field is written by the driver loading procedure.

(continued on next page)

Table A-5 (Cont.): Fields of CRB$L_INTD

Field Name	Contents
VEC$L_IDB*	Address of the interrupt dispatch block for the controller. The driver loading procedure creates an IDB for each CRB and loads the address of the IDB in this field. Device drivers use the IDB address to obtain the virtual addresses of device registers. When a driver interrupt service routine gains control, the top of the stack contains a pointer to this field.
VEC$L_INITIAL*	Address of the controller initialization routine. If a device controller requires initialization at driver loading time and during recovery from a power failure, the driver specifies a value for this field in the driver prologue table. The driver loading procedure calls this routine each time the procedure loads the driver. The VAX/VMS powerfail recovery procedure also calls this routine to initialize a controller after a power failure.
VEC$W_MAPREG	Number of the first UNIBUS adapter map register allocated to the driver that owns the controller data channel. IOC$REQMAPREG writes this field when the routine allocates a set of map registers to a driver fork process for a DMA transfer. IOC$RELMAPREG reads the field to deallocate a set of map registers. If the high bit (VEC$V_MAPLOCK) of this field is set, the map register set is permanently allocated. Device drivers read this field to calculate the starting address of a UNIBUS transfer.
VEC$B_NUMREG	Number of UNIBUS adapter map registers allocated to a driver. IOC$REQMAPREG writes this field when the routine allocates a set of map registers. IOC$RELMAPREG reads this field to deallocate a set of map registers.
VEC$B_DATAPATH	The data path specifier. The bits that make up this field are used as follows: 0 ⟶ 4 — The number of the data path used in a DMA transfer. The routine IOC$REQDATAP sets this field when a buffered data path is allocated and clears the field when the data path is released.

(continued on next page)

Table A-5 (Cont.): Fields of CRB$L_INTD

Field Name	Contents	
VEC$B_DATAPATH (Cont.)	0 ⟶ 4 (Cont.)	The routine IOC$LOADUBAMAP copies the contents of this field into the UNIBUS adapter map registers. These bits also serve as implicit input to the IOC$PURGDATAP routine.
	VEC$V_LWAE	Longword access enable (LWAE) bit. Drivers set this bit when they wish to limit the data path to longword-aligned random access mode. The routine IOC$LOADUBAMAP copies the value in this field to the UNIBUS adapter map registers.
	6	Reserved to DIGITAL.
	VEC$V_PATHLOCK	Buffered data path allocation indicator. Drivers set this bit to specify that the buffered data path is permanently allocated.
VEC$L_ADP*	Address of the UNIBUS adapter control block (ADP). The SYSGEN command CONNECT must specify the nexus number of the UNIBUS adapter used by a controller. The driver loading procedure writes the address of the ADP for the specified UBA into the VEC$L_ADP field. IOC$REQMAPREG and IOC$RELMAPREG read and write fields in the ADP to allocate and deallocate UNIBUS adapter map registers.	
VEC$L_UNITINIT*	Address of the device unit initialization routine. If a device unit requires initialization at driver loading time and during recovery from a power failure, the driver specifies a value for this field in the driver prologue table. The driver loading procedure calls this routine for each device unit each time the procedure loads the driver. The VAX/VMS powerfail recovery procedure also calls this routine to initialize device units after a power failure. MASSBUS drivers that support mixed device types must not use this field. Instead, they should specify unit initialization in the (DDT$L_UNITINIT) unit initialization field of the driver dispatch table. Other drivers may use either field.	

A.5 DEVICE DATA BLOCK (DDB)

The device data block is a variable-length 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 device data block for each controller during autoconfiguration at system startup and dynamically creates additional device data blocks for new controllers as they are added to the system using the SYSGEN command CONNECT. The procedure initializes all fields in the device data block. All the device data blocks in the I/O data base are linked together in a single-linked list. The contents of IOC$GL_DEVLIST points to the first entry in the list.

VAX/VMS routines and device drivers refer to the device data block.

Fields of the device data block are illustrated in Figure A-6 and described in Table A-6.

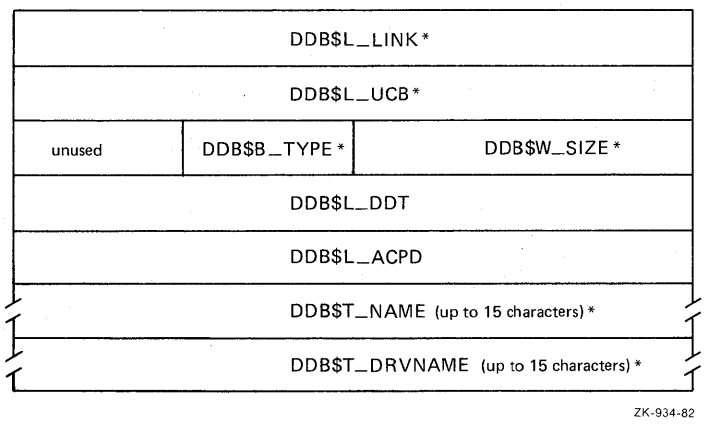


Figure A-6: Device Data Block

Table A-6: Contents of Device Data Block

Field Name	Contents
DDB$L_LINK*	Address of the next DDB. A zero indicates that this is the last DDB in the DDB chain.
DDB$L_UCB*	Address of the unit control block for the first unit attached to the controller.
DDB$W_SIZE*	Size of the DDB.
DDB$B_TYPE*	Type of control block. The driver loading procedure writes the constant DYN$C_DDB into this field when the procedure creates the DDB.
DDB$L_DDT	Address of the driver dispatch table. VAX/VMS can transfer control to a device driver only through addresses listed in the DDT, the CRB, and the UCB fork block. The driver prologue table of every device driver must specify a value for this field.

(continued on next page)

Table A-6 (Cont.): Contents of Device Data Block

Field Name	Contents
DDB$L_ACPD	Name of the default ACP for the controller. ACPs that control access to file-structured devices use the high-order byte of this field, DDB$B_ACPCLASS, to indicate the class of the file-structured device. If the SYSGEN parameter ACP_MULT is set to one, 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: DDB$K_CART — cartridge disk pack DDB$K_PACK — standard disk pack DDB$K_SLOW — floppy disk DDB$K_TAPE — magnetic tape that simulates a file-structured device
DDB$T_NAME*	Generic name of the devices attached to the controller. The first byte of this field is the number of characters in the generic name. The remainder of the field consists of a string of up to 15 characters in length that, suffixed by a device unit number, identifies devices on the controller.
DDB$T_DRVNAME*	Name of the device driver for the controller. The first byte of this field is the number of characters in the driver name. The remainder of the field contains a string of up to 15 characters in length taken from the driver prologue table in the driver.

A.6 DRIVER DISPATCH TABLE (DDT)

Each device driver contains a driver dispatch table. The table lists entry points in the driver that various VAX/VMS routines call. An example is the entry point for the driver routine that starts an I/O operation on a device.

A device driver creates a driver dispatch table by invoking the VAX/VMS macro DDTAB. The fields in the driver dispatch table are illustrated in Figure A-7 and described in Table A-7.

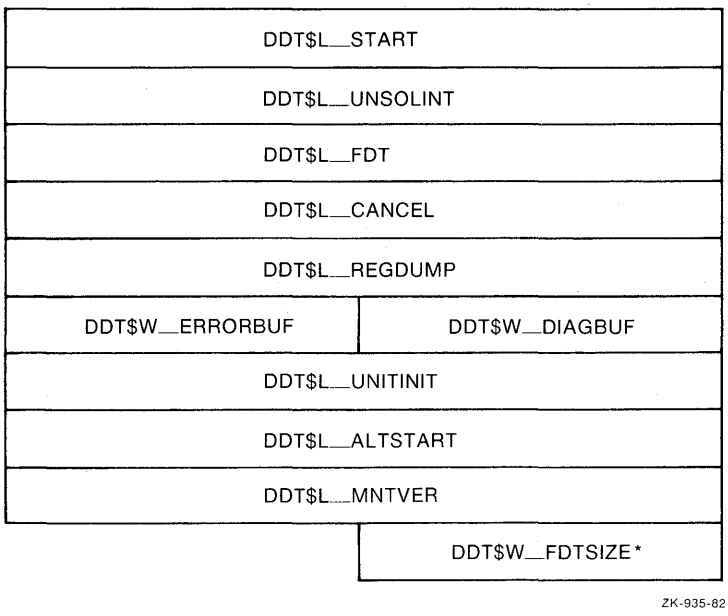


Figure A-7: Driver Dispatch Table

Table A-7: Contents of Driver Dispatch Table

Field Name	Contents
DDT$L_START	Entry point to the driver start I/O routine. Every driver must specify this field with the value of the START argument in the DDTAB macro invocation. When a device unit is idle and an I/O request is pending for that unit, IOC$INITIATE transfers control to the address contained in this field.
DDT$L_UNSOLINT	Entry point to the MASSBUS driver's unsolicited interrupt service routine. The driver specifies this field with the value of the UNSOLIC argument in the DDTAB macro invocation. This field contains the address of a routine that analyzes unexpected interrupts from a device. The standard driver interrupt service routine, the address of which is stored in the CRB, determines whether an interrupt was solicited by a driver. If the interrupt is unsolicited, the service routine may call the unsolicited interrupt service routine.
DDT$L_FDT	Address of the driver's function decision table. Every driver must specify this field with the value of the FUNCTB argument in the DDTAB macro invocation.

(continued on next page)

Table A-7 (Cont.): Contents of Driver Dispatch Table

Field Name	Contents
DDT$L_FDT (Cont.)	EXE$QIO refers to the FDT to validate I/O function codes, decide which functions are buffered, and call FDT action routines associated with function codes.
DDT$L_CANCEL	Entry point to the driver cancel I/O routine. The driver specifies this field with the value of the CANCEL argument in the DDTAB macro invocation. Some devices require special clean-up processing when a process or a VAX/VMS routine cancels an I/O request before the I/O operation completes or when the last channel is deassigned. The $DASSGN, $DALLOC, and $CANCEL system services cancel I/O requests.
DDT$L_REGDUMP	Entry point to the driver register dump routine. The driver specifies this field with the value of the REGDMP argument in the DDTAB macro invocation. IOC$DIAGBUFILL, ERL$DEVICERR, and ERL$DEVICTMO call the address contained in this field to write device register contents into a diagnostic or error-logging buffer.
DDT$W_DIAGBUF	Size of the diagnostic buffer. The driver specifes this field with the value of the DIAGBF argument in the DDTAB macro invocation. The value is the size in bytes of a diagnostic buffer for the device. When EXE$QIO preprocesses an I/O request, the routine allocates a system buffer of the size recorded in this field if the user process has diagnostic privileges, specifies a diagnostic buffer in the I/O request, and this field of the DDT contains a nonzero value. IOC$DIAGBUFILL fills the buffer after the I/O operation completes.
DDT$W_ERRORBUF	Size of the error log buffer. The driver specifies this field as the value of the ERLGBF argument in the DDTAB macro invocation. The value is the size in bytes of an error-logging buffer for the device. If error logging is enabled and an error occurs during an I/O operation, the driver calls ERL$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 error message buffer if an error has occurred.

(continued on next page)

Table A-7 (Cont.): Contents of Driver Dispatch Table

Field Name	Contents
DDT$L_UNITINIT	Address of the device unit initialization routine, if one exists. Drivers for MASSBUS devices use this field rather than CRB$L_INTD+VEC$L_UNITINIT. Drivers for UNIBUS devices may use either field.
DDT$L_ALTSTART	Address of the alternate start I/O routine. The VAX/VMS routine EXE$ALTQUEPKT initiates the alternate start I/O routine at this address.
DDT$L_MNTVER	Address of the VAX/VMS routine IOC$MNTVER called at the beginning and end of a mount verification operation. The MNTVER argument to the DPTAB macro defaults to this routine. Use of the MNTVER argument to call any routine other than IOC$MNTVER is reserved to DIGITAL.
DDT$W_FDTSIZE*	The number of bytes in the function decision table. The driver loading procedure uses this field to relocate addresses in the FDT to system virtual addresses.

A.7 DRIVER PROLOGUE TABLE (DPT)

When loading a device driver and its data base into virtual memory, the driver loading procedure finds the basic description of the driver and its device in a driver prologue table. This table provides the length, name, adapter type, and loading and reloading specifications for the driver.

A device driver creates a driver prologue table by invoking the VAX/VMS macros DPTAB and DPT_STORE. The fields of the DPT are illustrated in Figure A-8 and described in Table A-8.

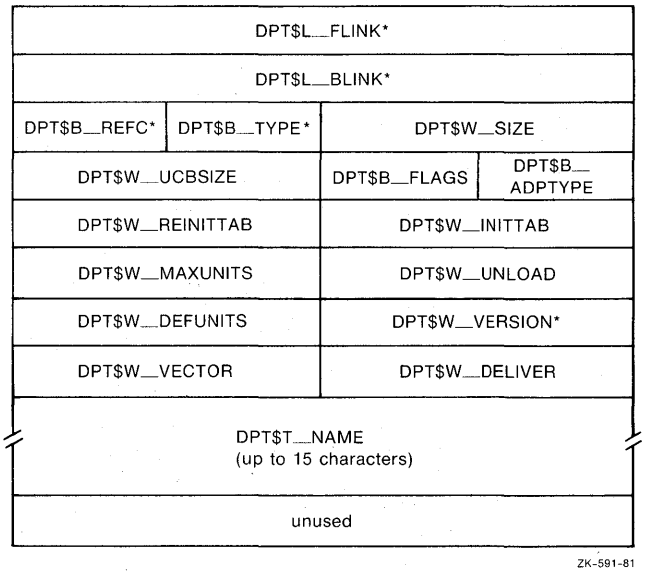


Figure A-8: Driver Prologue Table

Table A-8: Contents of Driver Prologue Table

Field Name	Contents
DPT$L_FLINK*	Forward link to the next DPT. The driver loading procedure writes this field. The procedure links all driver prologue tables in the system in a doubly linked list.
DPT$L_BLINK*	Backward link to the previous DPT. The driver loading procedure writes this field.
DPT$W_SIZE	Size in bytes of the device driver. The DPTAB macro writes this field by subtracting the address of the beginning of the DPT from the address specified as the END argument in the invocation of the DPTAB macro. The driver loading procedure uses this value to determine the space needed in nonpaged system memory to load the driver.
DPT$B_TYPE*	Type of control block. The DPTAB macro always writes the symbolic constant, DYN$C_DPT, into this field.
DPT$B_REFC*	Number of device data blocks that refer to this 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$B_ADPTYPE	Type of adapter used by devices driven by this driver. Every driver must specify the string "UBA" or "MBA" as value of the argument ADAPTER in the invocation of the DPTAB macro. The macro writes the value AT$_UBA or AT$_MBA in this field.
DPT$B_FLAGS	Driver loader flags. The driver can specify any of a set of flags as the value of the argument FLAGS in the invocation of the DPTAB macro. The driver loading procedure modifies the loading and reloading algorithm followed based on the settings of these flags. Flags defined in the flag field include the following: DPT$M_SUBCNTRL Device is a subcontroller DPT$M_SVP Device requires permanent system page; allocated during driver loading DPT$M_NOUNLOAD Driver cannot be reloaded
DPT$W_UCBSIZE	Size in bytes of unit control blocks created for device units driven by this driver. Every driver must specify a value for this field as the value of the argument UCBSIZE in the invocation of the DPTAB macro.

(continued on next page)

Table A-8 (Cont.): Contents of Driver Prologue Table

Field Name	Contents
DPT$W_UCBSIZE (Cont.)	The driver loading procedure allocates blocks of nonpaged system memory of the specified size when creating UCBs for devices associated with the driver.
DPT$W_INITTAB	Offset to driver initialization table. Every driver must specify a list of control block fields and values to be written into the fields at the time that the driver loading procedure creates the control blocks. The driver invokes the VAX/VMS macro DPT_STORE to specify these fields and their values. Every driver must specify the following fields: UCB$B_FIPL — Fork interrupt priority level UCB$B_DIPL — Device interrupt priority level Other commonly initialized fields are: UCB$L_DEVCHAR — Device characteristics UCB$B_DEVCLASS — Class of device UCB$B_DEVTYPE — Type of device UCB$W_DEVBUFSIZ — Default buffer size UCB$L_DEVDEPEND — Device-dependent parameters
DPT$W_REINITTAB	Offset to driver reinitialization table. Every driver must specify a list of control block fields and values to be written into fields at the time that the driver loading procedure creates the control blocks or loads the driver. The driver invokes the VAX/VMS macro DPT_STORE to specify these fields and their values. Every driver must specify the following field: DDB$L_DDT — Driver dispatch table Other commonly initialized fields are: CRB$L_INTD+4 — Interrupt service routine CRB$L_INTD2+4 — Second interrupt service routine VEC$L_INITIAL — Controller initialization routine VEC$L_UNITINIT — Unit initialization routine
DPT$W_UNLOAD	Relative address of a driver action routine to be called when a driver is reloaded. The driver specifes this field with the value of the UNLOAD argument in the invocation of the macro DPTAB.

(continued on next page)

Table A-8 (Cont.): Contents of Driver Prologue Table

Field Name	Contents
DPT$W_UNLOAD (Cont.)	If the driver requires special clean-up processing such as buffer or map register deallocation before the driver can be reloaded, the driver must specify this field. The driver loading procedure calls the driver unloading routine before reinitializing all device units associated with the driver.
DPT$W_MAXUNITS	Maximum number of units on a controller that this driver supports. Specify this value in the MAXUNITS argument to the DPTAB macro. If no value is specified, the default is 8 units.
DPT$W_VERSION*	The version number that identifies the format of the driver prologue table. The DPTAB macro automatically inserts a value in this field. SYSGEN checks its copy of the version number against the value stored in this field. If the values do not match, an error is generated. To correct the error, reassemble and relink the driver.
DPT$W_DEFUNITS	The number of unit control blocks that the autoconfigure facility will automatically create. Drivers specify this number with the DEFUNITS arugment to the DPTAB macro. If the driver also gives a value to DPT$W_DELIVER, this field is also the number of times that the autoconfiguration facility calls the unit delivery action routine.
DPT$W_DELIVER	Relative address of the device driver unit delivery routine that the autoconfiguration facility calls once for the number of unit control blocks specified in DPT$W_DEFUNITS. The driver gives the relative address in the DELIVER argument to the DPTAB macro.
DPT$W_VECTOR	Relative address of a driver-specific vector. Use of this field is reserved to DIGITAL.
DPT$T_NAME	Name of the device driver. Field is 12 bytes in length. One byte records the length of the name string; the name string can be up to 11 characters in length. Drivers specify this field as the value of the NAME argument in the invocation of the DPTAB macro. 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.

A.8 INTERRUPT DISPATCH BLOCK (IDB)

The interrupt dispatch block records controller characteristics. The driver loading procedure creates and initializes this block when the

procedure creates a channel request block. The interrupt dispatch block points to the physical controller by storing the virtual address of the control/status register. This register is the indirect pointer to all device unit registers.

The fields of the interrupt dispatch block are illustrated in Figure A-9 and detailed in Table A-9.

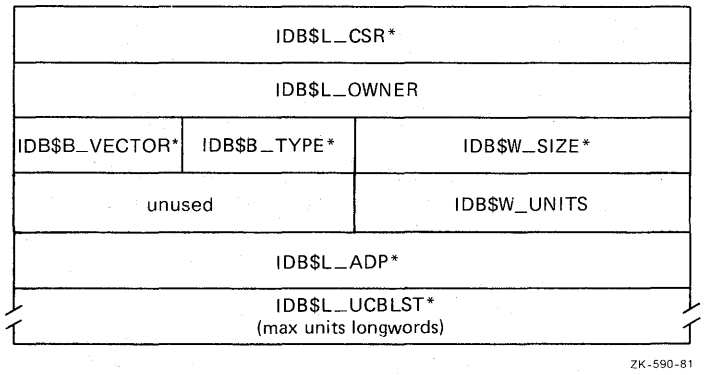


Figure A-9: Interrupt Dispatch Block

Table A-9: Contents of Interrupt Dispatch Block

Field Name	Contents
IDB$L_CSR*	Address of the control/status register (CSR). The SYSGEN command CONNECT must specify the address of a device's control/status register. The driver loading procedure writes the system virtual equivalent of this address into the IDB$L_CSR field. Device drivers set and clear bits in device registers by referencing all device registers at fixed offsets from the CSR address.
IDB$L_OWNER	Address of the unit control block of the device that owns the controller data channel. IOC$REQxCHANx writes a UCB address into this field when the routine allocates a controller data channel to a driver. IOC$RELxCHAN confirms that the proper driver fork process is releasing a channel by comparing the driver's UCB with the UCB stored in the IDB$L_OWNER field. If the UCB addresses are the same, IOC$RELxCHAN allocates the channel to a waiting driver by writing a new UCB address into the field. If no driver fork processes are waiting for the channel, IOC$RELxCHAN 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.

(continued on next page)

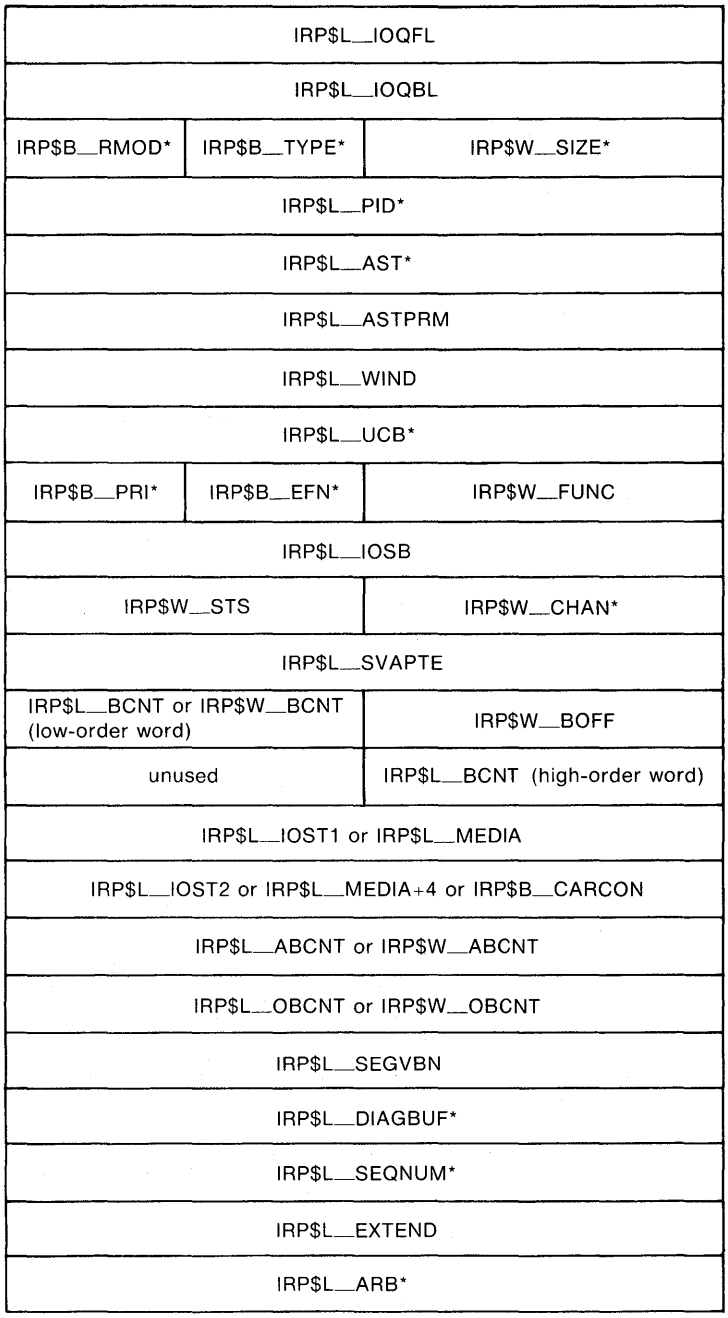
Table A-9 (Cont.): Contents of Interrupt Dispatch Block

Field Name	Contents
IDB$W_SIZE*	Size of the IDB. The driver loading procedure writes the constant IDB$K_LENGTH into this field when the procedure creates the IDB.
IDB$B_TYPE*	Type of control block. The driver loading procedure writes the symbolic constant DYN$C_IDB into this field when the procedure creates the IDB.
IDB$B_VECTOR*	For UNIBUS devices, the interrupt vector number of the device right-shifted by 2 bits. SYSGEN writes a value to this field using either the autoconfiguration data base or the value specified in the /VECTOR qualifier to the CONNECT command. Drivers for devices that define the interrupt vector address through a device register must use this field to load that register during unit initialization and reinitialization after a power failure.
IDB$W_UNITS*	Maximum number of units connected to the controller. The maximum number of units is specified in the driver prologue table and may be overridden at driver loading time.
IDB$L_ADP*	Address of the UNIBUS adapter control block (ADP). The SYSGEN command CONNECT must specify the nexus number of the UNIBUS adapter used by a device. The driver loading procedure writes the address of the ADP for the specified UNIBUS adapter into the IDP$L_ADP field.
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 driver prologue table. The maximum specified in the DPT can be overridden at driver load time. The driver loading procedure writes a UCB address into this field every time the routine creates a new UCB associated with the controller.

A.9 I/O REQUEST PACKET (IRP)

When a user process queues a valid I/O request by issuing a Queue I/O Request or Queue I/O Request and Wait system service, the service (EXE$QIO) creates an I/O request packet. This packet contains a description of the request and receives the status of the I/O processing as it proceeds.

The fields of an I/O request packet are illustrated in Figure A-10 and detailed in Table A-10.



ZK-587-81

Figure A-10: I/O Request Packet

Table A-10: Contents of an I/O Request Packet

Field Name	Contents
IRP$L_IOQFL	I/O queue forward link. EXE$INSERTIRP reads and writes this field when the routine inserts I/O packets into an I/O request packet wait queue. IOC$REQCOM reads and writes this field when the

(continued on next page)

Table A-10 (Cont.): Contents of an I/O Request Packet

Field Name	Contents
IRP$L_IOQFL (Cont.)	routine dequeues I/O packets from an I/O request packet wait queue in order to send the packet to a device driver.
IRP$L_IOQBL	I/O queue backward link. EXE$INSERTIRP and IOC$REQCOM read and write these fields.
IRP$W_SIZE*	Size of the I/O request packet. EXE$QIO writes the symbolic constant, IRP$C_LENGTH, into this field when the routine allocates and fills an I/O packet.
IRP$B_TYPE*	Type of control block. EXE$QIO writes the symbolic constant DYN$C_IRP into this field when the routine allocates and fills an I/O packet.
IRP$B_RMOD*	Access mode of the process at the time of the I/O request. EXE$QIO obtains the processor access mode from the PSL 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 process control block and writes the value into this field.
IRP$L_AST*	Address of the AST routine specified by the user in the I/O request. If the process specifies an AST routine address in the QIO call, EXE$QIO writes the address in this field. During I/O postprocessing, the kernel mode AST routine queues a user mode AST to the requesting process if this field contains the address of an AST routine.
IRP$L_ASTPRM	Address of a parameter to be sent as an argument to the AST routine specified by the user in the I/O request. If the process specifies an AST routine and a parameter to that AST routine in the QIO call, EXE$QIO writes the parameter in this field. During I/O postprocessing, the kernel mode AST routine queues a user mode 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$L_WIND	Address of a window control block (WCB) that describes the file being accessed in an I/O request. EXE$QIO writes this field if the I/O request refers to a file-structured device. The ACP reads this field.

(continued on next page)

Table A-10 (Cont.): Contents of an I/O Request Packet

Field Name	Contents
IRP$L_WIND (Cont.)	When a process gains access to a file on a file-structured device or creates a logical link between a file and a process I/O channel, the device ACP creates a window control block that describes the virtual-to-logical mapping of the file data on the disk. EXE$QIO stores the address of this WCB in the IRP$L_WIND field.
IRP$L_UCB*	Address of the unit control block for the device assigned to the process I/O channel. EXE$QIO copies this value from the channel control block.
IRP$W_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.
IRP$B_EFN*	Event flag number and group specified in the I/O request. If the I/O request call does not specify an event flag number, EXE$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$B_PRI*	Base priority of the process when the I/O request was issued. EXE$QIO obtains a value for this field from the process control block. EXE$INSERTIRP reads this field to insert an I/O request packet into a priority-ordered I/O request packet wait queue.
IRP$L_IOSB	Virtual address of the process I/O status block that receives the final status of the I/O request at I/O completion. EXE$QIO writes a value into this field if the I/O request call specifies an IOSB address. The I/O postprocessing kernel mode AST routine writes two longwords of I/O status into the IOSB block after the I/O operation is complete. When an FDT routine aborts an I/O request by calling EXE$ABORTIO, EXE$ABORTIO zeroes the IRP$L_IOSB field so that I/O postprocessing does not write status into the block.

(continued on next page)

Table A-10 (Cont.): Contents of an I/O Request Packet

Field Name	Contents
IRP$W_CHAN*	Index number of the process I/O channel for the request. EXE$QIO writes this field.
IRP$W_STS	Status of the I/O request. EXE$QIO initializes this field to 0. EXE$QIO, FDT routines, and driver fork processes modify this field according to the current status of the I/O request. I/O postprocessing reads this field to determine what sort of postprocessing is necessary (for example, deallocate system buffers and adjust quota usage). Bits in the IRP$W_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$V_SWAPIO — Swapping I/O funtion IRP$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$L_SVAPTE	For a direct I/O operation, specifies the virtual address of the first page table entry (PTE) of the I/O transfer buffer. FDT routines that lock pages in memory for a direct I/O transfer write the PTE address in this field. For a buffered I/O operation, specifies the address of the buffer in system address space. FDT routines that allocate system buffers for a buffered I/O transfer write this field. IOC$INITIATE copies the field into the device unit control block field 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 operation or to unlock pages locked for a direct I/O operation.

(continued on next page)

Table A-10 (Cont.): Contents of an I/O Request Packet

Field Name	Contents
IRP$W_BOFF	Byte offset into first page of a direct I/O transfer. FDT routines calculate this offset and write the field. For buffered I/O operations, FDT routines must write the number of bytes to be charged to the process in this field because these bytes are being used for a system buffer. IOC$INITIATE copies the field into the device unit control block field UCB$W_BOFF before calling a device driver start I/O routine. I/O postprocessing uses IRP$W_BOFF in conjunction with IRP$W_BCNT and IRP$L_SVAPTE to unlock pages locked for direct I/O. For buffered I/O, I/O postprocessing adds the value of IRP$W_BOFF to the process byte count quota.
IRP$L_BCNT or IRP$W_BCNT	Byte count of I/O transfer. FDT routines calculate the count value and write the field. IOC$INITIATE copies the low-order word of this field into the device unit control block field UCB$W_BCNT before calling a device driver start I/O routine. For a buffered I/O read function, I/O postprocessing uses IRP$L_BCNT to determine how many bytes of data to write to the user's buffer. The field UCB$W_BCNT points to the low-order word of this field to provide compatibility with previous versions of VAX/VMS.
IRP$L_IOST1 (also called IRP$L_MEDIA)	First I/O status longword. IOC$REQCOM and EXE$FINISHIO(C) write the contents of R0 into this field. The I/O postprocessing routine copies the contents of this field into the user I/O status block. EXE$ZEROPARM copies a 0 and EXE$ONEPARM copies P1 into this field. This field is a good place to put a Queue I/O Request argument (P1 through P6) or a computed value.
IRP$L_IOST2 (also called IRP$L_MEDIA+4 or IRP$B_CARCON)	Second I/O status longword. IOC$REQCOM and EXE$FINISHIO(C) write the contents of R1 into this field. The I/O postprocessing routine copies the contents of this field into the user I/O status block. 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 A-10 (Cont.): Contents of an I/O Request Packet

Field Name	Contents
IRP$L_ABCNT or IRP$W_ABCNT	Accumulated bytes transferred in a virtual I/O transfer. Read and written by IOC$IOPOST after a partial virtual transfer. The field UCB$W_ABCNT points to the low-order word of this field to provide compatibility with previous versions of VAX/VMS.
IRP$L_OBCNT or IRP$W_OBCNT	Original transfer byte count in a virtual I/O transfer. Read by IOC$IOPOST to determine whether a virtual transfer is complete, or whether another I/O request is necessary to transfer the remaining bytes. The field UCB$W_OBCNT points to the low-order word of this field to provide compatibility with previous versions of VAX/VMS.
IRP$L_SEGVBN	Virtual block number of the current segment of a virtual I/O transfer. Written by IOC$IOPOST after a partial virtual transfer.
IRP$L_DIAGBUF*	Address of a diagnostic buffer in system address space. If the I/O request call specifies this address, and if a diagnostic buffer length is specified in the driver dispatch table, and if the process has diagnostic privilege, EXE$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 kernel mode AST routine copies diagnostic data from the system buffer into the process diagnostic buffer.
IRP$L_SEQNUM*	I/O transaction sequence number. If an error is logged for the request, this field contains the universal error log sequence number.
IRP$L_EXTEND	Address of the I/O request packet extension linked to this packet. FDT routines write an extension address to this field when a device requires more context than the I/O request packet can accommodate. This field is read by IOC$POST. IRP$V_EXTEND in IRP$W_STS is set if this extension address is used.
IRP$L_ARB*	Address of the access rights block. This block is located in the process control block and contains the process privilege mask and UIC, which are set up as follows: ARB$Q_PRIV — Quadword containing process privilege mask

(continued on next page)

Table A-10 (Cont.): Contents of an I/O Request Packet

Field Name	Contents	
IRP$L_ARB* (Cont.)	SPARE$L	Unused longword
	ARB$L_UIC	Longword containing process UIC

A.10 I/O REQUEST PACKET EXTENSION (IRPE)

I/O request packet extensions hold additional I/O request information for devices that require more context than the standard I/O request packet 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 64K bytes. An IRPE provides space for two buffer regions, each with a 32-bit byte count.

FDT routines allocate IRPEs by calling EXE$ALLOCIRP. Driver routines link the IRP extension to the I/O request packet, store the extension's address in IRP$L_EXTEND and set the bit field IRP$V_EXTEND in IRP$W_STS to show that an extension exists for the packet. The FDT routine initializes the contents of the IRPE. Any fields within the extension not described in Table A-11 can store driver-dependent information.

If the IRP extension specifies additional buffer regions, the FDT routine must use those buffer locking routines that perform coroutine calls back to the driver if the locking procedure fails (EXE$READLOCKR, EXE$WRITELOCKR, and EXE$MODIFYLOCKR). If an error occurs during the locking procedure, the driver must unlock all previously locked regions and deallocate the I/O request packet extension before returning to the buffer locking routine.

IOC$IOPOST automatically unlocks the pages in region 1 (if defined) and region 2 (if defined) for all the IRP extensions linked to the packet being completed. IOC$IOPOST also deallocates all the IRPEs.

The fields of the I/O request packet extension are illustrated in Figure A-11 and described in Table A-11.

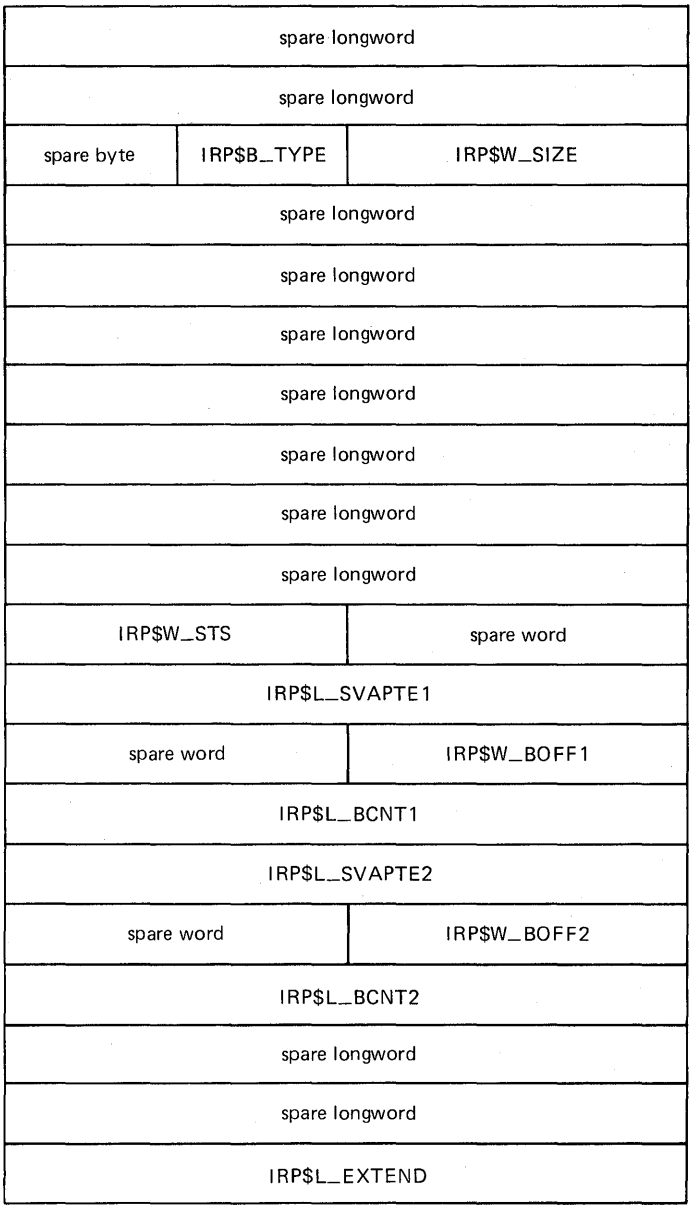


Figure A-11: Request Packet Extension

Table A-11: Contents of the I/O Request Packet Extension

Field Name	Contents
IRPE$W_SIZE	Size of the I/O request packet extension. EXE$ALLOCIRP writes the constant IRP$C_LENGTH to this field.
IRPE$B_TYPE	Type of control block. EXE$ALLOCIRP writes the constant DYN$C_IRP to this field.

(continued on next page)

Table A-11 (Cont.): Contents of the I/O Request Packet Extension

Field Name	Contents
IRPE$W_STS	IRP extension status field. Bits in the status field describe the following conditions: IRPE$V_EXTEND Another IRPE is linked to this one
IRPE$L_SVAPTE1	System virtual address of the page table entry mapping the start of region 1. FDT routines write this field. If the region is not defined, this field is zero.
IRPE$W_BOFF1	Byte offset of region 1. FDT routines write this field.
IRPE$L_BCNT1	Size in bytes of region 1. FDT routines write this field.
IRPE$L_SVAPTE2	System virtual address of the page table entry mapping the start of region 2. Set by FDT routines. This field contains a value of zero if region 2 is not defined.
IRPE$W_BOFF2	Byte offset of region 2. This field is set by FDT routines.
IRPE$L_BCNT2	Size in bytes of region 2. FDT routines write this field.

A.11 UNIT CONTROL BLOCK (UCB)

The unit control block is a variable-length block that describes a single device unit. Each device unit on the system has its own unit control block. The block 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 unit control block for each device unit in the system. A privileged system user can request the driver loading procedure to create unit control blocks for additional devices with the SYSGEN command CONNECT as described in Chapter 14. The procedure creates unit control blocks of the length specified in the driver prologue table of the device's driver. The driver uses UCB storage located beyond the standard UCB fields for device-specific data and temporary driver storage.

The driver loading procedure initializes some static unit control block fields when it creates the block. VAX/VMS and device drivers can read and modify all nonstatic fields of the unit control block.

The fields of the unit control block that are present for all devices are illustrated in Figure A-12 and described in Table A-12.

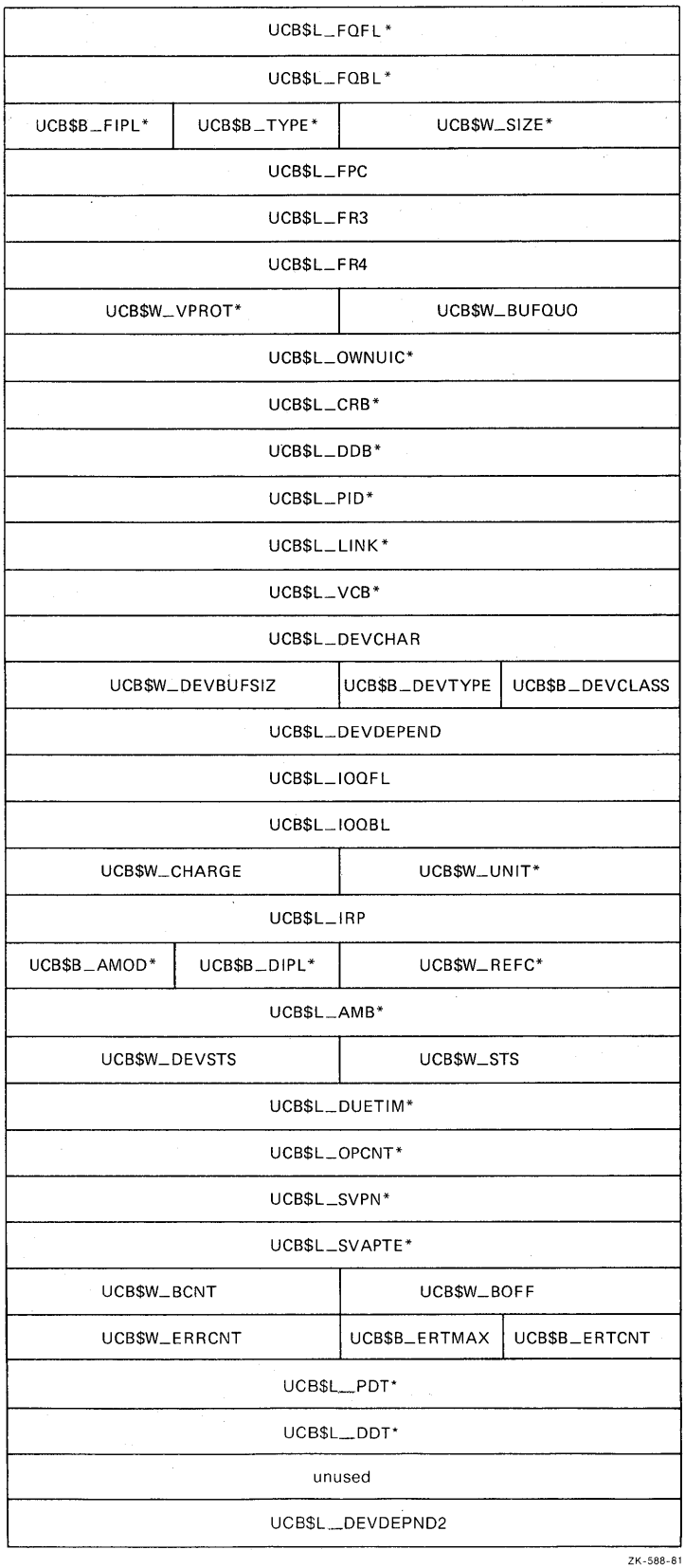


Figure A-12: Unit Control Block

Table A-12: Contents of Unit Control Block

Field Name	Contents
UCB$L_FQFL*	Fork queue forward link. The link points to the next entry in the fork queue. EXE$IOFORK and VAX/VMS 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$L_FQBL*	Fork queue backward link. The link points to the previous entry in the fork queue. EXE$IOFORK and VAX/VMS resource management routines write this field.
UCB$W_SIZE*	Size of the UCB. The driver prologue table of every driver must specify a value for this field. The driver loading procedure uses the value to allocate space for a UCB and stores the value in each UCB created. Extra space beyond the standard bytes in a UCB (UCB$K_LENGTH) is for device-specific data and temporary storage.
UCB$B_TYPE*	Type of the control block. The driver loading procedure writes the constant DYN$C_UCB into this field when the procedure creates the UCB.
UCB$B_FIPL*	Fork interrupt priority level (IPL) at which the driver of the device usually executes. The driver prologue table of every driver must specify a value for this field. The driver loading procedure writes the value in the UCB when the procedure creates the UCB. VAX/VMS creates a driver fork process that gains control in a driver start I/O routine at this IPL. When the driver creates a fork process after an interrupt, VAX/VMS inserts the fork block into a fork queue based on this IPL. A VAX/VMS fork dispatcher executing at UCB$B_FIPL dequeues the fork block and restores control to the suspended driver fork process. All devices that are attached to one UNIBUS adapter and actively compete for shared UNIBUS adapter resources and/or a controller data channel must specify the same value for the fork IPL field.
UCB$L_FPC	Fork process driver PC address. When a VAX/VMS routine saves driver fork context in order to suspend driver execution, the routine stores the address of the next driver instruction to be executed in this field. A VAX/VMS routine that reactivates a suspended driver transfers control to the saved PC address. VAX/VMS routines that suspend driver processing include EXE$IOFORK, IOC$REQxCHANx IOC$REQMAPREG,

(continued on next page)

Table A-12 (Cont.): Contents of Unit Control Block

Field Name	Contents
UCB$L_FPC (Cont.)	IOC$REQDATAP, and IOC$WFIKPCH. Routines that reactivate suspended drivers include IOC$RELCHAN, IOC$RELMAPREG, IOC$RELDATAP, EXE$FORKDSPTH, and driver interrupt service routines. When a driver interrupt service routine determines that a device is expecting an interrupt, the routine restores control to the saved PC address in the device's UCB.
UCB$L_FR3	Value of R3 at the time that a VAX/VMS routine suspends a driver fork process. The value of R3 is restored just before a suspended driver regains control.
UCB$L_FR4	Value of R4 at the time that an operating system routine suspends a driver fork process. The value of R4 is restored just before a suspended driver regains control.
UCB$W_BUFQUO*	Buffered I/O quota if this UCB represents a mailbox.
UCB$W_VPROT*	Description of the volume protection if a volume is mounted on this device. This field is written by the MOUNT command when a volume is mounted. It is read by EXE$QIO to check logical or physical access to a device and by the device's ACP. It is written by the SET PROTECTION/DEVICE command.
UCB$L_OWNUIC*	User identification code of volume owner. This field is written by the MOUNT command when a volume is mounted. It is read by EXE$QIO to check logical or physical access to a device and by the device's ACP. It is also written by the SET PROTECTION/DEVICE command.
UCB$L_CRB*	Address of the primary channel request block associated with the device. The driver loading procedure writes this field after it creates the associated CRB. Driver fork processes read this field to gain access to device registers. VAX/VMS routines use UCB$L_CRB to locate interrupt dispatching code and initialization routine addresses.
UCB$L_DDB*	Address of the device data block associated with the device. The driver loading procedure writes this field when the procedure creates the associated UCB. VAX/VMS routines generally read the DDB field in order to locate device driver entry points, the address of a driver function decision table, or the ACP associated with a given device.

(continued on next page)

Table A-12 (Cont.): Contents of Unit Control Block

Field Name	Contents
UCB$L_PID*	Process identification code of the process that has allocated the device. Written by $ALLOC.
UCB$L_LINK*	Address of the next UCB in the chain of UCBs attached to a single controller and associated with a device data block. The driver loading procedure writes this field when the procedure adds the next UCB. Any VAX/VMS routines that examine the status of all devices on the system read this field. Such routines include EXE$TIMEOUT, IOC$SEARCHDEV, and power failure recovery routines.
UCB$L_VCB*	Address of the volume control block (VCB) that describes the volume mounted on the device. This field is written by the device's ACP and read by EXE$QIOACPPKT and ACPs.
UCB$L_DEVCHAR	Device characteristics bits. The driver prologue table of every driver should specify symbolic constant values (defined by the $DEVDEF macro) for this field. The driver loading procedure writes the field when the procedure creates the UCB. The Queue I/O Request system service reads the field to determine whether a device is spooled, file-structured, shared, has a volume mounted, and so on. The system defines the following device characteristics: DEV$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$V_SQD Sequential block-oriented device (e.g., magtape) DEV$V_SPL Device is being spooled DEV$V_NET Network device DEV$V_FOD Files-oriented device (e.g., disk and magtape) DEV$V_SHR Shareable device (used by more than one program simultaneously) DEV$V_GEN Generic device DEV$V_AVL Device is available for use DEV$V_MNT Device is mounted DEV$V_MBX Mailbox device DEV$V_DMT Device is marked for dismount DEV$V_ELG Error-logging is enabled on device DEV$V_ALL Device is allocated DEV$V_FOR Device is mounted foreign (i.e., non-file-structured)

(continued on next page)

Table A-12 (Cont.): Contents of Unit Control Block

Field Name	Contents
UCB$L_DEVCHAR (Cont.)	DEV$V_SWL Device is software write-locked DEV$V_IDV Device is capable of providing input DEV$V_ODV Device is capable of providing output DEV$V_RND Device allows random access DEV$V_RTM Real time device DEV$V_RCK Read-checking is enabled on device DEV$V_WCK Write-checking is enabled on device
UCB$B_DEVCLASS	Device class. The driver prologue table of every driver should specify a symbolic constant (defined by the $DCDEF macro) for this field. The driver loading procedure writes this field when the UCB is created. Drivers with set mode and device characteristics functions rewrite the value in this field with data supplied in an I/O request. The VAX/VMS system defines the following device classes: DC$_DISK Disk device DC$_TAPE Tape device DC$_SCOM Synchronous communications device DC$_CARD Card reader device DC$_TERM Terminal device DC$_LP Line printer device DC$_REALTIME Real time device DC$_MAILBOX Mailbox device Note that the definition of a device as a real-time device is somewhat subjective; it implies no special treatment by VAX/VMS.
UCB$B_DEVTYPE	Device type. The driver prologue table of every driver should specify a symbolic constant (defined by the $DCDEF macro) for this field. The driver loading procedure writes the field when the procedure creates the UCB. Drivers with set mode and device characteristics functions rewrite the value in this field with data supplied in an I/O request.
UCB$W_DEVBUFSIZ	Default buffer size. The driver prologue table can specify a value for this field if relevant. The driver loading procedure writes the field when the procedure creates the UCB. Drivers with set mode and device characteristics functions rewrite the value in this field with

(continued on next page)

Table A-12 (Cont.): Contents of Unit Control Block

Field Name	Contents
UCB$W_DEVBUFSIZ (Cont.)	data supplied in an I/O request. This field is used by VAX-11 RMS for record I/O on non-file-oriented devices.
UCB$L_DEVDEPEND	Device-dependent data. Contains device-descriptive data that only the device driver can interpret. The driver prologue table can specify a value for this field. The driver loading procedure writes this field when the procedure creates the UCB. Drivers with set mode and device characteristics functions rewrite the value in this field with data supplied in an I/O request.
UCB$L_IOQFL*	I/O queue listhead forward link. The queue contains the addresses of I/O request packets waiting for processing on a device. EXE$INSERTIRP inserts I/O request packets into the I/O request packet wait queue when a device is busy. IOC$REQCOM dequeues I/O request packets when the device is idle. The queue is a priority queue that has the highest priority packets at the front of the queue. Priority is determined by the base priority of the requesting process. Packets with the same priority are processed first-in/first-out.
UCB$L_IOQBL*	I/O queue listhead backward link. EXE$INSERTIRP and IOC$REQCOM modify the I/O request packet wait queue.
UCB$W_UNIT*	Number of the physical device unit. Stored as a binary value. The driver loading procedure writes a value into this field when the UCB is created. Drivers for multiunit controllers read this field during unit initialization to identify a unit to the controller.
UCB$W_CHARGE*	Mailbox byte count quota charge, if the device is a mailbox.
UCB$L_IRP	Address of the I/O request packet currently being processed on the device unit by a driver fork process. IOC$INITIATE writes an I/O request packet address into this field before the routine creates a driver fork process to handle an I/O request. A driver fork process obtains the address of the I/O request packet being processed from this field. The value contained in this field is valid if the UCB$V_BSY bit in UCB$W_STS is set.

(continued on next page)

Table A-12 (Cont.): Contents of Unit Control Block

Field Name	Contents
UCB$W_REFC*	Reference count of processes that currently have process I/O channels assigned to the device. Incremented by the $ASSIGN and $ALLOC system services. Decremented by the $DASSGN and $DALLOC system services.
UCB$B_DIPL	Device interrupt priority level at which the device requests hardware interrupts. The driver prologue table of every driver must specify a value for this field. The driver loading procedure writes the field when the procedure creates the UCB. Some device drivers raise IPL to this value before reading or writing device registers.
UCB$B_AMOD*	If the device unit is allocated, the access mode at which the allocation occurred. Written by the $ALLOC and $DALLOC system services.
UCB$L_AMB*	Associated mailbox UCB pointer. This field is used for spooled devices and mailboxes.
UCB$W_STS	Device unit status. Written by drivers, IOC$REQCOM, IOC$CANCELIO, IOC$INITIATE, IOC$WFIKPCH, IOC$WFIRLCH, EXE$INSIOQ, and EXE$TIMEOUT. This field is read by drivers, the Queue I/O Request system service routines, IOC$REQCOM, IOC$INITIATE, and EXE$TIMEOUT. This status word includes the following bits: UCB$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 online 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_MNTVERIP — Mount verification in progress UCB$V_WRONGVOL — Volume name does not match name in volume control block UCB$V_DEADMO — Deallocate device at dismount UCB$V_VALID — Software believes volume is valid UCB$V_UNLOAD — Unload volume at dismount UCB$V_TEMPLATE — Template unit control block from which other UCBs for this device are

(continued on next page)

Table A-12 (Cont.): Contents of Unit Control Block

Field Name	Contents
UCB$W_STS (Cont.)	UCB$V_TEMPLATE (Cont.) made. The $ASSIGN system service checks this bit in the requested UCB and, if the bit is set, creates a UCB from the template. The new UCB is assigned instead.
UCB$W_DEVSTS	Device-dependent status. Read and written by device drivers.
UCB$L_DUETIM*	Due time for I/O completion. Stored as the low-order 32-bit absolute time (time in seconds since the operating system was booted) at which the device will timeout. IOC$WFIKPCH and IOC$WFIRLCH write this value when they suspend a driver to wait for an interrupt or timeout. EXE$TIMEOUT examines this field in each UCB in the I/O data base 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 the device unit since VAX/VMS was booted. IOC$REQCOM writes this field every time the routine inserts an I/O request packet in the I/O postprocessing queue.
UCB$L_SVPN*	Index to a virtual address of a system page table entry permanently allocated to the device by the driver loading procedure. The system virtual address of the page described by this index can be calculated by the formula (index * 200) + 80000000 (hex) If a driver prologue table specifies DPT$M_SVP in the flags argument to the DPTAB macro, the driver loading procedure allocates a page of nonpaged system memory to the device. The procedure writes the system page table entry index into UCB$L_SVPN when the procedure creates the UCB. This field is used for ECC error correction by disk drivers.
UCB$L_SVAPTE	For a direct I/O operation, the virtual address of the system page table entry (PTE) for the first page that is to be used in an I/O transfer. For a buffered I/O operation, the address of the system buffer used in the transfer. This field is used only in transfer operations.

(continued on next page)

Table A-12 (Cont.): Contents of Unit Control Block

Field Name	Contents
UCB$L_SVAPTE (Cont.)	IOC$INITIATE writes this field from IRP$L_SVAPTE before calling a driver start I/O routine. Drivers read this value to compute the starting address of a transfer.
UCB$W_BOFF	For direct I/O operations, byte offset in first page of the transfer buffer. For buffered I/O operations, the number of bytes charged to a process for a transfer. IOC$INITIATE copies this field from the I/O request packet. Drivers read the field in calculating the starting address of a DMA transfer. If only part of a DMA transfer succeeds, the driver adjusts the value in this field to be the byte offset in the first page of the data that was not transferred.
UCB$W_BCNT	Count of bytes in I/O transfer. IOC$INITIATE copies this field from the I/O request packet. Drivers read this field to determine how many bytes to transfer in an I/O operation.
UCB$B_ERTCNT	Error retry count of 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. If error-logging is occurring, IOC$REQCOM copies the value into the error message buffer.
UCB$B_ERTMAX	Maximum error retry count allowed for a single I/O transfer. The driver prologue table of some drivers specifies a value for this field. The driver loading procedure writes the field when the procedure creates the UCB. If error-logging is occurring, IOC$REQCOM copies the value into the error message buffer.
UCB$W_ERRCNT	Number of errors that have occurred on the device since the system was bootstrapped. 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$L_DDT*	Address of the driver dispatch table for this unit. The driver load procedure writes the contents of DDB$L_DDT for the device controller to this field when it creates the UCB.
UCB$L_PDT*	Address of the port descriptor table. This field is reserved for VAX/VMS port drivers.

(continued on next page)

Table A-12 (Cont.): Contents of Unit Control Block

Field Name	Contents
UCB$L_DEVDEPND2	Second longword for device-dependent status. This field is an extension of UCB$L_DEVDEPEND.

Unit control blocks are variable length depending on the type of device and whether the driver performs error-logging for the device. The error log UCB extension, if present, appears directly after the UCB$W_ERRCNT field of the standard UCB.

The fields in the UCB error log extension are illustrated in Figure A-13 and described in Table A-13.

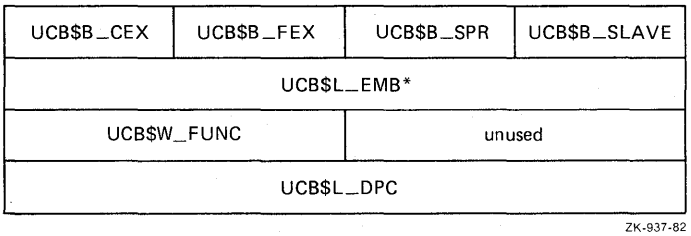


Figure A-13: UCB Error Log Extension

Table A-13: UCB Error Log Extension

Field Name	Contents
UCB$B_SLAVE*	Unit number of slave controller.
UCB$B_SPR	Spare byte. This field is reserved for driver use. MASSBUS adapter drivers use this field to store a fixed offset to the MASSBUS adapter registers for the unit.
UCB$B_FEX	Device-specific field. This field is reserved for driver use.
UCB$B_CEX	Device-specific field. This field is reserved for driver use.
UCB$L_EMB*	Address of the error message buffer. If error logging is enabled and a device/controller error or timeout occurs, the driver calls ERL$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$W_FUNC	I/O function modifiers. This field is read and written by drivers that log errors.

(continued on next page)

Table A-13 (Cont.): UCB Error Log Extension

Field Name	Contents
UCB$L_DPC	Device-specific field. This field is reserved for driver use.

Another extension of the unit control block is the disk extension block. This UCB extension is present for all disk devices. It follows the error log extension. A driver that supports a disk must allow space in the UCB for both the error log and disk extensions.

Disk drivers use three bits in UCB$W_DEVSTS as follows:

UCB$V_ECC	ECC correction made
UCB$V_DIAGBUF	Diagnostic buffer specifed
UCB$V_NOCNVRT	No logical block number to media address conversion

The fields are illustrated in Figure A-14 and described in Table A-14.

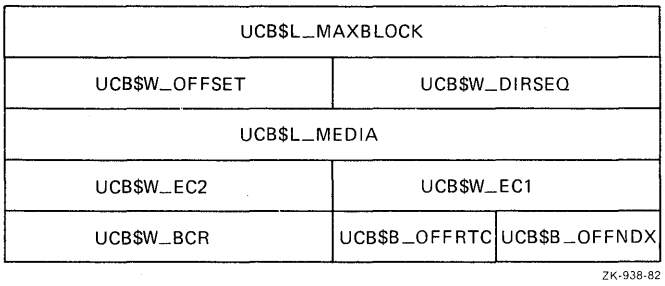


Figure A-14: UCB Disk Extension

Table A-14: UCB Disk Extension

Field Name	Contents
UCB$L_MAXBLOCK	Maximum number of logical blocks on a random access device. This field is written by a disk driver during unit initialization and power recovery.
UCB$W_DIRSEQ	Directory sequence number.
UCB$W_OFFSET	Current offset register contents.
UCB$L_MEDIA	Media address.
UCB$W_EC1	ECC position register. This field records the starting bit number of an error burst. Disk driver register dump routines copy the contents of this field into an error-logging or diagnostic buffer.

(continued on next page)

Table A-14 (Cont.): UCB Disk Extension

Field Name	Contents
UCB$W_EC1 (Cont.)	The VAX/VMS correction routine IOC$APPLYECC reads the contents of this field to locate the beginning of an error burst in a disk block.
UCB$W_EC2	ECC position register. Records the exclusive OR correction pattern. Disk driver register dump routines copy the contents of this field into an error-logging or diagnostic buffer. The VAX/VMS ECC correction routine IOC$APPLYECC reads the contents of this field to correct disk data.
UCB$B_OFFNDX	Current offset table index. When a disk driver transfer ends in an error, the disk driver can retry the error a number of times with different offsets of the disk head from the centerline. This field is an index into a driver table of offset positions.
UCB$B_OFFRTC	Current offset retry count. This field records the number of times to try a particular offset setting in a disk transfer retry.
UCB$W_BCR	Byte count register. Some disk drivers use this field as an internal count of the number of bytes left to be transferred in an I/O request.

APPENDIX B

VAX/VMS MACROS INVOKED BY DRIVERS

This appendix contains an alphabetical listing of macros that drivers invoke. Default values are provided where applicable.

$DEF	Defines a field within the structure delineated by the $DEFINI and $DEFEND macros
SYM	Name of symbol used to access the field
[ALLOC]	Block storage allocation directives. Possible directives are .BLKB, .BLKW, .BLKL, .BLKQ, .BLKO. You can define multiple symbols for the same field by leaving this parameter blank.
[SIZ]	Number of block storage units to allocate
$DEFEND	Declares the end of a structure definition
STRUC	Name of structure
$DEFINI	Declares the start of a structure definition
STRUC	Name of structure (for example, CRB, UCB, and so on)
GBL=LOCAL	Specifies that symbols within the structure are local (LOCAL value) or global (GLOBAL value). LOCAL is the default.
DOT=0	Declares offset from base of structure
$EQULST	Defines symbol names and values
PREFIX	Prefix for all symbols generated
GBL=LOCAL	Defines symbols within structure as local (default) or global
INIT	Default value for first symbol generated
INCR=1	Default increment for symbol values
LIST	List of elements. Each element takes the form <SYM,VALUE> SYM is concatenated with PREFIX to form the symbol name. If specified, VALUE is the value assigned to the symbol.

$VIELD	Generates symbols whose values are bits within major fields defined by $DEF macros. Symbols take the form: MOD$V_SYM MOD$S_SYM MOD$M_SYM
MOD	Prefix for symbols generated by this macro
INIBIT	Initial bit offset within subfield from which subfield definitions are based
FIELDS	One or more subfields of the form: <SYM,[SIZ],[MSK]>
SYM	Suffix of each symbol defined for this subfield
SIZ=1	Width of subfield in bits
MSK	Generates a symbol whose value is a mask
_VIELD	Generates subfields within major fields defined by $DEF macros. This macro uses the same syntax as $VIELD, but symbols take the form: MOD_V_SYM MOD_S_SYM MOD_M_SYM
CASE	Generates a CASE instruction and CASE table
SRC	Source of CASE index value
DISPLIST	List of destinations for each case (dest1, dest2, dest3)
TYPE=W	Data type (B, W, L)
LIMIT=#0	Lower limit of CASE value
NMODE=S^#	Address mode for number of table entries; the short literal default is good for up to 63 entries
DDTAB	Generates a driver dispatch table named devnam$DDT
DEVNAM	Generic device name
START=IOC$RETURN	Address of start I/O routine
UNSOLIC=IOC$RETURN	Address of unsolicited interrupt service routine for MASSBUS drivers
FUNCTB	Address of function decision table
CANCEL=IOC$RETURN	Address of cancel I/O routine
REGDMP=IOC$RETURN	Address of error-logging register dump routine
DIAGBF=0	Length in bytes of diagnostic buffer
ERLGBF=0	Length in bytes of error logging buffer
UNITINIT=IOC$RETURN	Device unit initialization routine
ALTSTART=IOC$RETURN	Alternate start I/O routine
MNTVER=IOC$MNTVER	Address of mount verification routine

DPTAB	Generates a driver prologue table in PSECT $$$105_PROLOGUE
END	Address of the end of the driver
ADAPTER	Type of adapter (UBA, MBA, DR or NULL)
FLAGS=0	Driver loading flags (DPT$M_SVP and DPT$M_NOUNLOAD)
UCBSIZE	Size in bytes of each device UCB
[UNLOAD]	Optional address of a routine to call if the driver is to be unloaded
MAXUNITS=8	Maximum number of units that can be connected
DEFUNITS=1	Number of UCBs to be created by AUTOCONFIGURE
[DELIVER]	Address of action routine that determines whether a unit is automatically configured
[VECTOR]	Address of a driver-specific transfer vector (use of this field is reserved to DIGITAL)
NAME	Driver name
DPT_STORE	Generates a table containing initialization values for fields in the I/O data base
STR_TYPE	Type of control block (DDB, UCB, CRB, IDB); or table marker (INIT, REINIT, END)
STR_OFF	Offset into control block
OPER	Type of initialization operation (B=byte, W=word, L=long, D=address relative to driver, V=bit field); if an at sign (@) precedes the OPERATION, then the EXPRESSION argument is the address of the initialization data
EXP	Initialization value to be stored in control block
POS	Bit position for OPERATION=V
SIZE	Field size for OPERATION=V
DSBINT	Disables interrupts by raising IPL
[IPL]	IPL value to be loaded into the IPL processor register PR$_IPL (defaults to 31)
[DST]	Location for old IPL value (defaults to top of stack)
ENBINT	Enables interrupts by restoring a saved IPL
[SRC]	Location in which an IPL is saved (defaults to top of stack)
FORK	Calls EXE$FORK to create a fork process
FUNCTAB	Generates a function decision table consisting of two 64-bit entries of function codes, and n 96-bit entries of function codes and action routine addresses
[ACTION]	Address of an FDT routine to call for the function codes listed
CODES	A list of I/O function codes

IFNORD	Branches if a range of addresses is not readable
SIZ	Number of bytes in range (0 - 512)
ADR	Address of first byte in range
DEST	Location to branch to if the range of addresses is not readable
MODE=#0	Access mode at which to probe (defaults to USER)
IFNOWRT	Branches if a range of addresses is not writeable
SIZ	Number of bytes in range (0 - 512)
ADR	Address of first byte in range
DEST	Location to branch to if the range of addresses is not writeable
MODE=#0	Access mode at which to probe (defaults to USER)
IFRD	Branches if a range of addresses is readable
SIZ	Number of bytes in range (0 - 512)
ADR	Address of first byte in range
DEST	Location to branch to if the range of addresses is readable
MODE=#0	Access mode at which to probe (defaults to USER)
IOFORK	Calls EXE$IOFORK to create a device driver fork process
LOADUBA	Calls IOC$LOADUBAMAP to load a preallocated set of UNIBUS adapter map registers
PURDPR	Calls IOC$PURGDATAP to purge a data path
RELCHAN	Calls IOC$RELCHAN to release all controller data channels that are allocated by the driver
RELDPR	Calls IOC$RELDATAP to release a preallocated UNIBUS adapter data path
RELMPR	Calls IOC$RELMAPREG to release a preallocated set of UNIBUS adapter map registers
RELSCHAN	Calls IOC$RELSCHAN to release all secondary controller data channels that are allocated by the driver
REQCOM	Calls IOC$REQCOM to complete an I/O request after driver processing is finished

REQDPR	Calls IOC$REQDATAP to request a UNIBUS adapter data path
REQMPR	Calls IOC$REQMAPREG to request a set of UNIBUS map registers
REQPCHAN	Calls IOC$REQPCHANH or IOC$REQPCHANL to request a primary controller data channel
[PRI]	Priority of request; if PRI=HIGH, calls IOC$REQPCHANH; otherwise calls IOC$REQPCHANL
REQSCHAN	Calls IOC$REQSCHANH or IOC$REQSCHANL to request a secondary controller data channel
[PRI]	Priority of request; if PRI=HIGH calls IOC$REQSCHANH; otherwise calls IOC$REQSCHANL
SAVIPL	Saves the current IPL value as recorded in the processor register PR$_IPL
DST=-(SP)	Location in which to save the current IPL (defaults to a new top of stack)
SETIPL	Sets IPL to a new value
[IPL]	New IPL value (defaults to 31)
SOFTINT	Initiates a software interrupt
IPL	IPL value of the interrupt; loads IPL into the processor register PR$_SIRR
TIMEWAIT	Checks for specific state by testing bit(s) for a given length of time. Returns success or failure status in R0.
TIME	Number of 10 microsecond intervals to wait for state
BITVAL	Mask for bit(s) to test
SOURCE	Address of bit(s) to test
CONTEXT	Bit test context: B, W, or L
SENSE=.TRUE.	If .TRUE., test for one or more bits set; otherwise, test for all bits clear
WFIKPCH	Calls an executive subroutine to wait for an interrupt or a device timeout and keep the controller data channel
EXCPT	Relative address of a device timeout handling routine; writes the address into the two bytes following the call to the executive routine.
[TIME]	Number of seconds to allow before a device timeout (defaults to 65536 seconds)

WFIRLCH	Calls an executive subroutine to wait for an interrupt or a device timeout and release the controller data channel
EXCPT	Relative address of a device timeout handling routine; writes the address into the two bytes following the call to the executive routine.
[TIME]	Number of seconds to allow before a device timeout (defaults to 65536 seconds)

APPENDIX C

OPERATING SYSTEM ROUTINES

This appendix describes the VAX/VMS operating system routines that are used by device drivers. The information given in this section follows the conventions listed below:

- Fields used for both input and output are not specified.
- Registers are assumed preserved unless otherwise specified.
- IPL at execution refers to the interrupt priority level at which the routine executes, not the IPL at which it is called.

COM$DELATTNAST

module: COMDRVSUB

Driver fork processes call this routine to deliver all the AST control blocks linked to the specified AST list.

INPUT TO ROUTINE

Registers	**Contents**
R4	Address of specified listhead
R5	Address of the unit control block
Fields	**Contents**
---	---

IPL at execution: caller's IPL

This routine removes all AST blocks from the specified list and schedules an IPL$_QUEUEAST level fork process to queue each AST to its process.

OUTPUT FROM ROUTINE

Registers	**Contents**
---	---

Fields	Contents
Specified listhead	0

IPL at exit: caller's IPL

COM$DRVDEALMEM

module: COMDRVSUB

Drivers use this routine to deallocate system dynamic memory. COM$DRVDEALMEM can be called from any interrupt priority level.

INPUT TO ROUTINE

Registers	Contents
R0	Address of the block to be deallocated

Fields	Contents
IRP$W_SIZE	Size of the block in bytes

IPL at execution: caller's IPL and IPL$_QUEUEAST

If the block size is smaller than 24 bytes or the block is not properly aligned, a system bugcheck occurs. This routine also calls SCH$RAVAIL to mark the resource free.

IPL at exit: caller's IPL

COM$FLUSHATTNS

module: COMDRVSUB

Driver FDT and fork routines call this routine to flush an attention AST list. Drivers use this routine during cancel I/O operations.

INPUT TO ROUTINE

Registers	Contents
R4	Address of the current PCB
R5	Address of the UCB
R6	Number of the assigned channel
R7	Address of the AST control block listhead

Fields	Contents
UCB$B_DIPL	Device IPL
PCB$L_PID	Process's ID
PCB$W_ASTCNT	ASTs remaining in quota

IPL at execution: device IPL (UCB$B_DIPL)

COM$FLUSHATTNS locates all the control blocks whose channel number and process identification match those specified as input to the routine, removes them from the specified list and deallocates them. This routine exits by returning to its caller.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL
R1	Destroyed
R2	Destroyed
R7	Destroyed

Fields	Contents
PCB$W_ASTCNT	Number of ACBs flushed (added to previous contents)
Specified listhead	Updated

IPL at exit: caller's IPL

COM$POST

module: COMDRVSUB

Drivers call this routine after they have completed all device-dependent I/O postprocessing for an I/O request. This routine inserts the I/O request packet into the I/O postprocessing queue and returns to the driver fork process. COM$POST operates independently of the device unit; it does not attempt to dequeue another packet nor does it change the busy status of the device.

Drivers can use this routine to complete I/O request packets initiated by the routine EXE$ALTQUEPKT.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet
R5	Address of the unit control block

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

IPL at execution: caller's IPL (driver fork level or above)

This routine places the I/O request packet into the queue headed by IOC$GL_PSBL.

OUTPUT FROM ROUTINE

Registers	Contents
R0 - R1	Destroyed
Fields	**Contents**
UCB$L_OPCNT	Incremented by 1

IPL at exit: caller's IPL

COM$SETATTNAST

module: COMDRVSUB

Driver FDT routines call this routine to enable or disable attention ASTs, depending upon the contents of the queue I/O parameter P1. To enable an AST, P1 contains the address of an AST routine. The routine allocates a control block that can double as an AST control block when the AST is delivered.

This control block contains the following information:

- The address of the specified AST routine
- The specified AST parameter
- The specified access mode
- The channel number
- The process identification of the requesting process

COM$SETATTNAST links the control block to the start of the specified linked list of AST control blocks located in the unit control block extension area.

If P1 is clear, the routine disables ASTs by searching through the linked list, extracting each entry, and deallocating it.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the IRP
R4	Address of the current PCB
R5	Address of the UCB
R6	Address of the assigned channel control block
R7	Address of the specified AST control block listhead
AP	Address of the QIO parameter list

Fields	Contents
IRP$W_CHAN	I/O request channel number
UCB$B_DIPL	Device IPL
PCB$W_ASTCNT	Number of ASTs remaining in process quota
PCB$L_PID	Process identification
0(AP)	Process AST address
4(AP)	AST parameter
8(AP)	Access mode for AST

IPL at execution: caller's IPL and device IPL

If the process exceeds buffered I/O or AST quotas, or if there is no memory available to allocate an AST control block, this routine transfers control to EXE$ABORTIO with error status.

If P1 is clear, the routine transfers control to COM$FLUSHATTNS to remove the identified AST control block.

This routine exits to its caller.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL (success) SS$_EXQUOTA SS$_INSFMEM
R1 - R2	Destroyed
R3	Address of the IRP
R5	Address of the UCB
R6 - R8	Destroyed

Fields	Contents
PCB$W_ASTCNT	Decreased by 1
Specified listhead	Updated

IPL at exit: caller's IPL

ERL$DEVICERR

module: ERRORLOG

Logs a controller and/or device error. This routine allocates an error message buffer and writes data from the I/O request packet and unit control block. It also calls the driver register dump routine for device registers.

INPUT TO ROUTINE

Registers	Contents
R5	Address of unit control block

ERL$DEVICERR sets the error type code to device error. This routine uses fields in the UCB, DDB, DDT, and I/O request packet. It also assumes that the driver contains a register dump routine. It uses the DDT to calculate the address of the register dump routine and then calls it.

If you do not specify a dump routine in the DDTAB macro invocation, DDTAB supplies the address of IOC$RETURN. IOC$RETURN simply returns; it is a NOP.

OUTPUT FROM ROUTINE

Registers	Contents
---	---

Fields	Contents
UCB$L_EMB	Address of the error message buffer
UCB$W_STS	Shows error log in progress

ERL$DEVICTMO

module: ERRORLOG

Logs a device timeout. This routine performs the same functions and uses the same input and output as ERL$DEVICERR with one exception: the error type code is device timeout.

ERL$RELEASEMB

module: ERRORLOG

Wakes the error log process to write the contents of an error message buffer into the error logging file.

INPUT TO ROUTINE

Registers	Contents
R2	Address of error message buffer

Fields	Contents
ERL$V_TIMER (in ERL$GB_BUFFLAG)	Determines whether a timer is running on the buffer

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed

Fields	Contents
Busy message count (in ERL$B_BUSY)	Decreased by 1
Complete message count (in error message buffer header)	Incremented by 1

If ERL$B_MSGCNT is greater than the maximum message count, this routine wakes the error logger.

IPL at exit: caller's IPL

EXE$ABORTIO

module: SYSQIOREQ

FDT routines jump to this routine to finish an I/O operation without returning final I/O status in the IOSB. This routine zeroes the IOSB field of the I/O request packet, clears a bit to prevent a user mode AST, and inserts the I/O request packet in the I/O postprocessing queue.

INPUT TO ROUTINE

Registers	Contents
R0	First longword of status for I/O status block
R3	Address of I/O request packet
R4	Address of current PCB
R5	Address of UCB

Fields	Contents
ACB$V_QUOTA (in IRP$B_RMOD)	Set to 1 (when an AST is specified)

IPL at execution: IPL$_ASTDEL

OUTPUT FROM ROUTINE

Registers	Contents
None written	---

Fields	Contents
ACB$V_QUOTA (in IRP$B_RMOD)	Cleared to zero (if field previously set)
IRP$L_IOSB	Zero
PCB$W_ASTCNT	Incremented if ACB$V_QUOTA was set

EXE$ABORTIO places the I/O request packet into the I/O postprocessing queue headed by IOC$GL_PSBL.

IPL at exit: 0 (normal process IPL)

EXE$ALLOCBUF

module: MEMORYALC

FDT routines call this routine to allocate a buffer for a buffered I/O operation from the nonpaged system pool. This routine can place the process in a resource wait state if sufficient memory is not available, and the process has resource wait mode enabled. The caller must adjust process quotas.

INPUT TO ROUTINE

Registers	Contents
R1	Size of requested buffer in bytes
R4	Address of current PCB

Fields	Contents
PCB$V_SSRWAIT	One or zero. Determines whether process should wait, if no memory available for requested buffer. If this field is set, resource wait mode is disabled.

IPL at execution: caller's IPL, IPL 11, and IPL$_SYNCH

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL (success) SS$_INSFMEM
R1	Size of allocated buffer (requested size is rounded up to next 16-byte multiple)
R2	Address of allocated buffer
R3	Destroyed

Fields	Contents
IRP$W_SIZE (in allocated buffer)	Buffer size in bytes
IRP$B_TYPE (in allocated buffer)	DYN$C_BUFIO

IPL at exit: IPL$_ASTDEL

EXE$ALLOCIRP

EXE$ALLOCIRP

module: MEMORYALC

This routine allocates an I/O request packet from nonpaged dynamic memory. It performs the same functions and has the same input and output as EXE$ALLOCBUF, with the following exceptions:

- The caller does not specify a buffer size
- The allocated buffer is IRP$C_LENGTH bytes long
- The buffer size is set to IRP$C_LENGTH
- The buffer type is set to DYN$C_IRP

EXE$ALONONPAGED

module: MEMORYALC

Driver fork processes use this routine to allocate a block of memory from the nonpaged system pool.

The block header is not initialized.

INPUT TO ROUTINE

Registers	**Contents**
R1	Requested block size in bytes
Fields	**Contents**
none	---

IPL at execution: caller's IPL (must not fall below IPL11) and IPL 11

OUTPUT FROM ROUTINE

Registers	**Contents**
R0	Status code (0 or 1)
R1	Size of allocated buffer (requested size rounded up to next 16-byte multiple)
R2	Address of allocated block
R3	Destroyed
Fields	**Contents**
---	---

IPL at exit: caller's IPL

EXE$ALTQUEPKT

module: SYSQIOREQ

Driver FDT routines and fork processes call this routine to send an I/O request packet to a driver's alternate start I/O routine so that it bypasses the I/O request queue for the device's unit control block. EXE$ALTQUEPKT passes the address of the I/O request packet to the driver without regard for the status of the device unit.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet
R5	Address of the unit control block
Fields	**Contents**
DDT$L_ALTSTART	Address of the alternate start I/O routine
UCB$B_FIPL	Driver fork IPL
UCB$L_DDB	Address of unit's DDB
DDB$L_DDT	Address of the driver dispatch table

IPL at execution: UCB$L_FIPL

EXE$ALTQUEPKT calls the alternate start I/O routine and returns to its caller.

OUTPUT FROM ROUTINE

Registers	Contents
R0 - R5	Destroyed
Fields	**Contents**
---	---

IPL at exit: caller's IPL

EXE$BUFFRQUOTA

EXE$BUFFRQUOTA

module: EXSUBROUT

FDT routines call this routine to determine whether a process's buffered byte count quota usage permits the process to be granted additional buffered I/O. This routine may place the process in a resource wait state if quota usage is too large, and the process has resource wait mode enabled.

INPUT TO ROUTINE

Registers	Contents
R1	Number of requested bytes
R4	Address of PCB

Fields	Contents
PCB$V_SSRWAIT	When process exceeds quota, determines whether process should wait. If this field is set, resource wait mode is disabled.
IOC$GW_MAXBUF	Maximum number of buffered I/O bytes that system allows to any process
JIB$L_BYTLM	Process's byte count limit
JIB$L_BYTCNT	Process's byte count usage quota

IPL at execution: caller's IPL and IPL$_SYNCH

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL (success) SS$_EXQUOTA
R2 - R3	Destroyed

Fields	Contents
---	---

IPL at exit: IPL$_ASTDEL

EXE$BUFQUOPRC

module: EXSUBROUT

EXE$BUFQUOPRC performs the same function and has the same input and output as EXE$BUFFRQUOTA with the following exception: EXE$BUFQUOPRC does not check the field IOC$GW_MAXBUF.

EXE$DEANONPAGED

module: MEMORYALC

Deallocates a block of memory to the nonpaged system pool.

This routine performs the same functions and has the same input and output as the routine COM$DRVDEALMEM, with the following exceptions:

- R3 is destroyed
- The caller's IPL must be at IPL$_QUEUEAST or lower

EXE$FINISHIO

module: SYSQIOREQ

FDT routines transfer control to this routine to finish an I/O operation and return a quadword of final I/O status to the requesting process. This routine writes final I/O status into the I/O request packet and inserts the I/O request packet in the I/O postprocessing queue.

INPUT TO ROUTINE

Registers	Contents
R0	First longword of status for the I/O status block
R1	Second longword of status for the I/O status block
R3	Address of the I/O request packet
R4	Address of the current process control block
R5	Address of the UCB

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL
Fields	**Contents**
IRP$L_MEDIA	First longword of I/O status (R0)
IRP$L_MEDIA+4	Second longword of I/O status (R1)
UCB$L_OPCNT	Incremented by 1

This routine places the I/O request packet into the I/O postprocessing queue headed by IOC$GL_PSBL.

EXE$FINISHIOC

module: SYSQIOREQ

This routine performs the same functions and has the same input and output as EXE$FINISHIO with the following exception: EXE$FINISHIOC clears the contents of R1 before storing R0 and R1 in the I/O request packet.

EXE$FORK

module: FORKCNTRL

This routine performs the same functions as EXE$IOFORK except that this routine does not disable timeouts by clearing UCB$V_TIM in the UCB$W_STS field of the unit control block.

EXE$FORKDSPTH

module: FORKCNTRL

The interrupt service routine that dispatches fork processes in a fork queue. This routine gains control when the processor grants a software interrupt at IPLs 6 and 8 through 11. When EXE$FORKDSPTH gains control the stack contains the following information:

- 0(SP) contains the PC at the time of the interrupt
- 4(SP) contains the PSL at the time of the interrupt

R0 through R5 at the time of the interrupt are also saved by EXE$FORKDSPTH.

SWI$GL_FQFL indexed by the current IPL contains the address of the head of the fork queue for this IPL. Each entry in the fork queue is the address of a fork block that contains R3, R4, a PC, and implicitly R5; R5 is the address of the fork block.

If the queue is empty when the interrupt occurs, EXE$FORKDSPTH dismisses the interrupt without error.

EXE$FORKDSPTH empties the fork queue corresponding to the IPL of the interrupt. For each queue entry, it restores R3 and R4 from the fork block, saves the dispatch address and IPL on the stack, and executes a JSB to the saved PC address. When the queue is empty, it dismisses the interrupt.

The IPL on return from each fork process must equal the IPL at which the process was called. If IPL does not match, EXE$FORKDSPTH signals the fatal bugcheck BADFORKIPL.

EXE$INSERTIRP

module: SYSQIOREQ

Inserts an I/O request packet according to the base priority of the I/O request packet's originating process into the I/O request packet wait queue of a unit control block.

INPUT TO ROUTINE

Registers	Contents
R2	Address of the I/O queue list head for the device
R3	Address of the I/O request packet
Fields	**Contents**
---	---

IPL at execution: caller's IPL (fork level or higher)

OUTPUT FROM ROUTINE

Registers	Contents
R1	Destroyed

This routine places the I/O request packet in the queue and sets the Z condition code in the PSL as follows:

1 indicates that the entry is first in the queue.

0 indicates that at least one entry was already in the queue.

IPL at exit: caller's IPL

EXE$INSIOQ

module: SYSQIOREQ

Examines the unit control block. If the device is idle, this routine calls IOC$INITIATE; if the device is busy, it calls EXE$INSERTIRP.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet
R5	Address of the UCB
Fields	**Contents**
UCB$B_FIPL	Driver fork IPL
UCB$V_BSY (in UCB$W_STS)	Determines whether device is busy
UCB$L_IOQFL	Address of device I/O queue listhead

IPL at execution: driver fork level

OUTPUT FROM ROUTINE

Registers	Contents
R0 - R2	Destroyed

Additional registers used by the driver start I/O routine will be destroyed if the start I/O routine is called.

Fields	Contents
UCB$V_BSY (in UCB$W_STS)	Set to 1

IPL at exit: original IPL

EXE$INSTIMQ

module: EXESUBROUT

Inserts a timer queue element into the timer queue. Elements are ordered according to expiration time with those elements closest to due time taking priority.

INPUT TO ROUTINE

Registers	Contents
R0, R1	Quadword expiration time for new element
R5	Address of timer element to be queued

IPL at execution: IPL$_TIMER

OUTPUT FROM ROUTINE

Registers	Contents
R2 - R3	Destroyed

IPL at exit: IPL$_TIMER

EXE$IOFORK

module: FORKCNTRL

Saves the contents of R3 and R4 in the fork block specified by R5. This routine pops the return PC off the top of stack and saves the PC value in the fork block. It inserts the fork block address into the fork queue corresponding to the IPL stored in the fork block.

INPUT TO ROUTINE

Registers	Contents
R5	Address of the fork block (usually the UCB)
0(SP)	Return address of caller
4(SP)	Return address of caller's caller

Fields	Contents
FKB$B_FIPL (in fork block)	Fork IPL

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R3	Destroyed
R4	FKB$B_FIPL

Fields	Contents
UCB$V_TIM (in UCB$W_STS)	Zero
FKB$L_FR3 (in UCB)	R3
FKB$L_FR4 (in UCB)	R4
FKB$L_FPC (in UCB)	0(SP)

The routine queues the UCB address to the list headed by SWI$GL_FQFL. If the queue is empty, requests a software interrupt at fork IPL.

IPL at exit: caller's IPL

EXE$MODIFY

module: SYSQIOFDT

FDT routines transfer control to this device-independent routine that validates and readies a user buffer for a DMA read/write operation. Use EXE$MODIFY instead of EXE$READ when you wish your driver to read and write to a buffer. EXE$MODIFY disables a paging mechansim used during write-only operations.

This routine performs the following functions:

- Translates read logical functions to read physical functions
- Transfers queue I/O parameters to the I/O request packet

- Verifies that the caller has access to the specified buffer
- Locks the buffer's pages into physical memory. If a page fault occurs during this step, the routine returns control to the Queue I/O Request system service, which repeats the request.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet
R4	Address of the current PCB
R5	Address of the UCB assigned to the device unit
R6	Address of the CCB for the channel assigned to the device unit
R7	Bit number of the I/O function code
R8	FDT entry address
AP	Address of the first function-dependent QIO parameter (P1)

Fields	Contents
0(AP)	Virtual address of buffer (P1)
4(AP)	Number of bytes in transfer (P2)
12(AP)	Carriage control byte (P4)
IRP$W_FUNC	I/O function code

IPL at execution: caller's IPL (IPL$_ASTDEL)

If this routine completes successfully, it transfers control to EXE$QIODRVPKT. If EXE$MODIFY fails, it transfers control to EXE$ABORTIO.

EXE$MODIFY does not check for zero length transfers and will queue an IRP that specifies a zero length buffer to the UCB. The driver start I/O routine should check for zero length buffers to avoid mapping them to UNIBUS or MASSBUS space, because the attempted mapping causes a system failure.

OUTPUT FROM ROUTINE

Registers	Contents
R0 - R2	Destroyed

Fields	Contents
IRP$B_CARCON	P4
IRP$V_FUNC (in IRP$W_STS)	Set to 1 (indicates a read function)
IRP$L_SVAPTE	Address of page table entry that maps the first page of the buffer

Fields	Contents
IRP$W_BCNT	Size of the transfer in bytes

IPL at exit: caller's IPL

EXE$MODIFYLOCK

module: SYSQIOFDT

FDT routines call EXE$MODIFYLOCK to perform buffer processing on a DMA transfer. This routine:

- Determines whether the caller has write access to the buffer
- Locks the buffer's pages into memory. If a page fault occurs during this process, the routine returns control to the Queue I/O Request system service, which resubmits the request.

Use EXE$MODIFYLOCK instead of EXE$READLOCK when you expect your driver to read and write to a buffer. EXE$MODIFYLOCK disables a paging mechanism used in write-only operations.

INPUT TO ROUTINE

Registers	Contents
R0	Starting address of buffer
R1	Size of transfer in bytes
R3	Address of the I/O request packet
R4	Address of current PCB
R6	Address of the CCB

Fields	Contents
---	---

IPL at execution: caller's IPL (IPL$_ASTDEL)

If EXE$MODIFYLOCK fails, it transfers control to EXE$ABORTIO. If the routine completes successfully, control is returned to its caller.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL
R1	Address of the PTE that maps the first page of the buffer
R2	Destroyed
R3	Address of the IRP

Fields	Contents
IRP$L_SVAPTE	Address of the PTE that maps the first page of the buffer
IRP$W_BCNT	Size of the transfer in bytes
IRP$V_FUNC (in IRP$W_STS)	A value of 1 (indicating a read function)

IPL at exit: caller's IPL

EXE$MODIFYLOCKR

module: SYSQIOFDT

This routine determines whether a process has write access to the buffer pages it requested, then, if access is permitted, it locks the pages into memory. If the access check or page locking procedure fails, the routine calls the driver to clean up QIO bookkeeping. Drivers typically use EXE$MODIFYLOCKR when they must lock multiple areas into memory for one I/O request, and then need to unlock previously locked areas after an I/O request is aborted.

INPUT TO ROUTINE

Registers	Contents
R0	Starting address of buffer
R1	Length of the buffer in bytes
R3	Address of the IRP
R4	Address of the current process's PCB
R6	Address of the channel control block

Fields	Contents
---	---

EXE$MODIFYLOCKR may fail for a number of reasons:

- The buffer access check fails. In this case, the routine returns SS$_ACCVIO to the driver in R0.
- The caller process has an insufficient working set limit to lock all the buffer pages into memory. The routine returns SS$_INSFWSL in R0.
- A page fault occurs while the routine is locking pages into memory. The status returned in R0 in this case is zero.

If any of the above errors occur, the routine calls back the driver as a coroutine with error status in R0 and all other registers preserved.

The driver performs necessary queue I/O cleanup, that is, it carries out any procedures that the system does not perform as part of the normal queue I/O request abort processing.

The driver must preserve all registers, including R0 and R1.

When the driver returns by executing an RSB instruction, EXE$MODIFYLOCKR aborts the I/O request if R0 contains an error status, then performs processing that results in the I/O request's being resubmitted to the driver. For example:

```
        JSB     G^EXE$MODIFYLOCKR
        BLBS    BUF_LOCK_OK

BUF_LOCK_FAIL:
                <clean up this QIO bookkeeping>
                RSB

BUF_LOCK_OK:
                <continue this QIO>
```

If the subroutine is successful, it returns control to its caller.

OUTPUT FROM ROUTINE

Registers	**Contents**
R0	SS$_NORMAL (1)
R1	Address of the PTE that maps the first page of the buffer
R2	Function indicator (set to 1)
R3	Address of the IRP
Fields	**Contents**
IRP$L_SVAPTE	Address of the PTE that maps the first page of the buffer
IRP$W_BCNT	Size of the transfer in bytes
IRP$M_FUNC (in IRP$W_FUNC)	Set to 1

IPL at exit: caller's IPL

EXE$ONEPARM

module: SYSQIOFDT

Device-independent FDT routine that copies a single QIO parameter into the I/O request packet and calls EXE$QIODRVPKT.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet for the current I/O request
R4	Address of the process control block of the current process
R5	Address of the unit control block of the device assigned to the user-specified process I/O channel
R6	Address of the channel control block that describes the user-specified process I/O channel
R7	Bit number of the user-specified I/O function code
R8	Address of FDT entry
AP	Address of the first function-dependent parameter specified in the user's request

Fields	Contents
---	---

OUTPUT FROM ROUTINE

Registers	Contents
---	---

Fields	Contents
IRP$L_MEDIA (of IRP)	P1

IPL at exit: caller's IPL

This routine exits to EXE$QIODRVPKT.

Chapter 8 provides more information about this routine.

EXE$QIODRVPKT

module: SYSQIOREQ

FDT routines call this routine to send an IRP to a driver start I/O routine. This routine calls EXE$INSIOQ and then transfers control to EXE$QIORETURN.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet
R4	Address of the process control block
R5	Address of the unit control block

Fields	Contents
UCB$B_FIPL	Driver fork IPL
UCB$V_BSY (in UCB$W_STS)	Unit busy flag
UCB$L_IOQFL	Address of unit I/O queue listhead

EXE$QIORETURN

module: SYSQIOREQ

Sets a success status code in R0, lowers IPL to 0, and returns to the system service dispatcher.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL

IPL at exit: 0

This routine returns by issuing a RET instruction.

EXE$READ

module: SYSQIOFDT

Device-independent FDT routine that validates and readies a user buffer for a DMA read operation. This routine performs the same functions and has the same input and output as EXE$MODIFY, with a single exception noted in the description of EXE$MODIFY.

EXE$READCHK

module: SYSQIOFDT

Checks pages for write accessibility by a process. This routine writes the total byte count of a transfer into the I/O request packet.

If pages do not allow write access, the routine transfers control to EXE$ABORTIO, which terminates the request with access violation status. If EXE$READCHK completes successfully, control returns to its caller.

INPUT TO ROUTINE

Registers	**Contents**
R0	Address of buffer
R1	Size of the transfer in bytes
R3	Address of the I/O request packet
Fields	**Contents**
---	---

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	**Contents**
R0	Address of buffer (success)
R1	Size of transfer in bytes
R2	Value of 1 (to indicate a read)
R3	Address of IRP
Fields	**Contents**
IRP$W_BCNT	Size of transfer in bytes
IRP$V_FUNC (in IRP$W_STS)	Value of 1 (indicates a read function)

IPL at exit: caller's IPL

EXE$READCHKR

module: SYSQIOFDT

This routine performs the same function as EXE$READCHK, except that, upon error, it calls the driver FDT routine back as a coroutine to clean up QIO bookkeeping. See the description of error procedures in EXE$MODIFYLOCKR for further information.

EXE$READLOCK

module: SYSQIOFDT

FDT routines call this routine to check buffer accessibility and lock the user buffer in memory for a DMA read transfer. This routine performs the same functions and has the same input and output as EXE$MODIFYLOCK, except that it is used when the driver performs only a read I/O function.

EXE$READLOCKR

module: SYSQIOFDT

This subroutine determines whether a process has write access to requested buffer pages and, if access is permitted, it locks those pages into memory. EXE$READLOCKR performs the same functions and has the same input and output as EXE$MODIFYLOCKR.

EXE$SENSEMODE

module: SYSQIOFDT

Device-independent FDT routine that copies device-dependent characteristics from the device's UCB into R1. This routine writes a success code into R0 and transfers control to EXE$FINISHIO.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet for the current I/O request
R4	Address of the PCB of the current process
R5	Address of the UCB of the device assigned to the user-specified process I/O channel
R6	Address of the CCB that describes the user-specified process I/O channel
R7	Bit number of the user-specified I/O function code
R8	Address of function decision table dispatch
AP	Address of the first function-dependent parameter specified in the user's request

Fields	Contents
UCB$L_DEVDEPEND	Device-dependent status

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL
R1	Device-dependent characteristics copied from UCB$L_DEVDEPEND

Fields	Contents
---	---

IPL at exit: caller's IPL

This routine exits to EXE$FINISHIO.

For additional information, refer to Chapter 8.

EXE$SETCHAR

module: SYSQIOFDT

Device-independent FDT routine that writes a quadword whose address is QIO parameter P1 into the device's unit control block. Writes a success code into R0 and transfers control to EXE$FINISHIO.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the IRP for the current I/O request
R4	Address of the current PCB
R5	Address of the UCB of the assigned device unit
R6	Address of the CCB that describes the specified process I/O channel
R7	Bit number of the I/O function code
R8	Address of the FDT dispatcher
AP	Address of the first function-dependent QIO parameter

Fields	Contents
0(AP)	Address of new device characteristics (P1)

IPL at execution: caller's IPL

If this routine fails because the user lacks read access to the characteristics quadword, control transfers to EXE$ABORTIO with access violation status.

If EXE$SETCHAR completes successfully, it transfers control to EXE$FINISHIO.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL (success) SS$_ACCVIO (failure)

Fields	Contents
UCB$B_DEVCLASS	Byte 0 of quadword
UCB$B_DEVTYPE	Byte 1 of quadword
UCB$W_DEVBUFSIZ	Bytes 2 and 3 of quadword
UCB$L_DEVDEPEND	Bytes 4 through 7 of quadword

IPL at exit: caller's IPL

Refer to Chapter 8 for additional information on this routine.

EXE$SETMODE

module: SYSQIOFDT

Device-independent FDT routine that writes a quadword whose address is a QIO parameter into the I/O request packet and calls EXE$QIODRVPKT.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet for the current I/O request
R4	Address of the PCB of the current process
R5	Address of the UCB of the device assigned to the user-specified process I/O channel
R6	Address of the CCB that describes the user-specified process I/O channel
R7	Bit number of the I/O function code
R8	Address of the FDT entry
AP	Address of the first function-dependent QIO parameter

Fields	Contents
P0(AP)	Address of a quadword of device characteristics

IPL at execution: caller's IPL

If the user lacks read access to the device characteristics quadword, the routine transfers control to EXE$ABORTIO with access violation status. If EXE$SETMODE completes successfully, it normally exits to EXE$QIODRVPKT.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL (success) SS$_ACCVIO

Fields	Contents
IRP$L_MEDIA	First longword of device characteristics quadword
IRP$L_MEDIA+4	Second longword of device characteristics quadword

IPL at exit: caller's IPL

For more information about this routine, refer to Chapter 8.

EXE$SNDEVMSG

module: MBDRIVER

Driver fork processes call this routine to send messages to system processes such as OPCOM. This routine constructs a message on the stack and calls EXE$WRTMAILBOX to send the message to a mailbox.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the mailbox UCB
R4	Message type
R5	Address of the UCB
Fields	**Contents**
UCB$W_INIT	Device unit number
UCB$L_DDB	Address of device DDB
DDB$T_NAME	Device controller name

Mailbox UCB fields

IPL at execution: caller's IPL (must be at or below IPL$_MAILBOX)

This routine can fail for one of the following reasons:

- The message is too large for the mailbox
- The message mailbox is full of messages
- The system is unable to allocate memory for the message
- The caller lacks privilege to write to the mailbox

If any of the above conditions occur, EXE$SNDEVMSG returns error status to the caller.

If EXE$SNDEVMSG completes successfully, it exits with an RSB instruction.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL (success) SS$_MBTOOSML (message too large for mailbox) SS$_MBFULL (mailbox full of messages) SS$_INSFMEM (memory allocation problem) SS$_NOPRIV (no owner write access)
R1-R4	Destroyed

Fields	Contents
---	---

IPL at exit: caller's IPL

EXE$WRITE

module: SYSQIOFDT

Device-independent FDT routine that validates and readies a user buffer for a DMA write operation. This routine performs the same steps as EXE$MODIFY, and has the same input and output.

EXE$WRITECHK

module: SYSQIOFDT

Checks pages for read accessibility by a process and writes the total byte count of a transfer into the I/O request packet. If pages do not allow read access, the routine transfers control to EXE$ABORTIO, which terminates the request with access violation status. If EXE$WRITECHK completes successfully, it returns to its caller.

INPUT TO ROUTINE

Registers	Contents
R0	Address of buffer
R1	Size of the transfer in bytes
R3	Address of the I/O request packet

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R0	Buffer address (success)
R1	Size of the transfer in bytes
R2	Cleared (indicates a write function)
R3	Address of the I/O request packet

Fields	Contents
IRP$W_BCNT	Contains transfer size in bytes

IPL at exit: caller's IPL

EXE$WRITECHKR

module: SYSQIOFDT

This routine performs the same functions as EXE$WRITECHK, except that, upon error, it calls the driver FDT routine back as a coroutine to clean up QIO bookkeeping.

See the description of error procedures in EXE$MODIFYLOCKR for more information about coroutine cleanup.

EXE$WRITELOCK

module: SYSQIOFDT

FDT routines call this routine to determine whether the caller has read access to the buffer and to lock the buffer in memory for a DMA write transfer.

INPUT TO ROUTINE

Registers	Contents
R0	Starting address of I/O buffer
R1	Length of transfer in bytes
R3	Address of the I/O request packet
R4	Address of the PCB
R6	Address of the CCB

Fields	Contents
---	---

IPL at execution: caller's IPL (IPL$_ASTDEL)

This routine calls EXE$WRITECHK and MMG$IOLOCK. MMG$IOLOCK locks pages in memory. If EXE$WRITELOCK fails because a page fault occurs during the locking procedure, it transfers control to the Queue I/O Request system service, which repeats the I/O request. It exits to EXE$ABORTIO if it cannot complete successfully. If the routine does complete without error, it returns control to its caller.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL
R1	Address of the PTE that maps the first page of the buffer
R2	Destroyed
R3	Address of the IRP

Fields	Contents
IRP$L_SVAPTE	Address of the PTE that maps the first page of the buffer
IRP$W_BCNT	Size of the transfer in bytes
IRP$V_FUNC (in IRP$W_STS)	A value of 0 (indicating a write function)

IPL at exit: caller's IPL

EXE$WRITELOCKR

module: SYSQIOFDT

This routine determines whether the process has read access to the requested buffer pages, and, if access is permitted, it locks those pages into memory. EXE$WRITELOCKR performs the same functions as EXE$MODIFYLOCKR, with the following exceptions:

- R2, on output, contains a zero to indicate a write function
- IRP$M_FUNC (in IRP$W_FUNC) is clear (zero) to indicate a write function)

EXE$WRTMAILBOX

module: MBDRIVER

Driver fork processes call this routine to send messages to mailboxes.

INPUT TO ROUTINE

Registers	Contents
R3	Size of message
R4	Message address
R5	Address of mailbox UCB

Fields

Mailbox UCB fields

IPL at execution: caller's IPL (must be at or below IPL$_MAILBOX)

This routine can fail for one of the following reasons:

- The message is too large for the mailbox
- The message mailbox is full of messages
- The system is unable to allocate memory for the message
- The caller lacks privilege to write to the mailbox

If any of the above conditions occur, EXE$WRTMAILBOX returns an error status to its caller.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL (success) SS$_MBTOOSML (message too large for mailbox) SS$_MBFULL (mailbox full of messages) SS$_INSFMEM (memory allocation problem) SS$_NOPRIV (no owner write access)
R1 - R2	Destroyed

EXE$ZEROPARM

module: SYSQIOFDT

Device-independent FDT routine that clears the parameter field of the IRP and calls EXE$QIODRVPKT.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet for the current I/O request
R4	Address of the process control block of the current process
R5	Address of the unit control block of the device assigned to the user-specified process I/O channel
R6	Address of the channel control block that describes the user-specified process I/O channel

Registers	Contents
R7	Bit number of the user-specified I/O function code
R8	Address of FDT entry
AP	Address of the first function-dependent parameter specified in the user's request

Fields	Contents
---	---

IPL at execution: caller's IPL

This routine exits by transferring control to EXE$QIODRVPKT.

OUTPUT FROM ROUTINE

Registers	Contents
---	---

Fields	Contents
IRP$L_MEDIA	Zero

IPL at exit: caller's IPL

For additional information, refer to Chapter 8.

IOC$ALOUBAMAP(N)

module: IOSUBNPAG

This routine searches the map register bit map in the adapter control block to allocate a set of contiguous map registers to a driver fork process.

INPUT TO ROUTINE

Registers	Contents
R3	Number of map registers to allocate (if entry is IOC$ALOUBAMAPN)
R5	Address of the UCB

Fields	Contents
UCB$W_BCNT	Transfer byte count (if entry is IOC$ALOUBAMAP)
UCB$W_BOFF	Byte offset in page (if entry is IOC$ALOUBAMAP)
UCB$L_CRB	Address of the CRB

Fields	Contents
CRB$L_INTD+ VEC$L_ADP	Address of the device's adapter control block
VEC$V_MAPLOCK (in CRB$L_INTD +VEC$W_MAPREG)	Bit that indicates whether map registers are permanently allocated to this controller
ADP$W_MRBITMAP	Determines which map registers are available

IPL at execution: caller's IPL

If map registers are already permanently allocated to the controller, this routine exits successfully without allocating any map registers. Otherwise, the routine searches the map register bit map for the required number of contiguous map registers, calls IOC$ALTUBAMAP, and exits by issuing an RSB instruction.

OUTPUT FROM ROUTINE

Registers	Contents
R0	1 (success) 0 (insufficient contiguous map registers)
R1 - R2	Destroyed

Fields	Contents
CRB$L_INTD+ VEC$B_NUMREG	Number of map registers allocated
CRB$L_INTD+ VEC$W_MAPREG	Starting map register number
ADP$W_MRBITMAP	Bits for allocated map registers set to zero.

IPL at exit: caller's IPL

IOC$ALTUBAMAP

module: IOSUBNPAG

Clears or sets a field of bits in the UNIBUS adapter map register allocation bit map.

Registers	Contents
R0	Alternation bit mask (zeros to clear bits, ones to set bits)
R1	Address of the channel request block
R2	Address of the adapter control block
R4	Number of the starting map register

Fields	Contents
CRB$L_INTD+ VEC$B_NUMREG	Number of map registers needed

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R3 - R4	Destroyed

Fields	Contents
ADP$W_MRBITMAP	Bits describing available map registers

IPL at exit: caller's IPL

IOC$APPLYECC

module: IOSUBRAMS

Disk drivers call this routine to apply an ECC correction to data transferred from a device into memory. This routine corrects the data by exclusive ORing a correction pattern from the unit control block. It also sets a UCB bit to indicate that an ECC correction has been made.

Input from Routine

Registers	Contents
R0	Number of bytes of data that have been transferred, not including the block to be corrected; this must be a multiple of 512 bytes
R5	Address of the unit control block

Fields	Contents
UCB$W_BCNT	Length of transfer in bytes
UCB$W_EC1	Starting bit number of the error burst
UCB$W_EC2	Exclusive OR correction pattern
UCB$L_SVPN	Address of the system page table entry of a page that is available for use by driver
UCB$L_SVAPTE	System virtual address of the page table entry that maps the transfer

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R0 - R2	Destroyed
Fields	**Contents**
UCB$V_ECC (in UCB$W_DEVSTS)	Set to 1 to show that an ECC correction was made

IPL at exit: caller's IPL

IOC$CANCELIO

module: IOSUBNPAG

Device-independent cancel I/O routine that sets a cancel I/O bit in the unit control block if the I/O request packet being currently processed on the device originates from the current process on the specified channel and the unit is busy.

INPUT TO ROUTINE

Registers	Contents
R2	channel index number
R3	Address of the I/O request packet
R4	Address of the current PCB
R5	Address of the unit control block
Fields	**Contents**
IRP$L_PID	Process identification of the process that queued the I/O request
IRP$W_CHAN	Channel index number
PCB$L_PID	Process identification of the process that requested cancellation
UCB$V_BSY (in UCB$W_STS)	Device busy flag

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
---	---

Fields	Contents
UCB$V_CANCEL (in UCB$W_STS)	Set if I/O request should be cancelled

IPL at exit: caller's IPL

IOC$DIAGBUFILL

module: IOSUBNPAG

Driver fork processes call this routine to fill a diagnostic buffer, if the QIO specifies such a buffer. This routine writes completion time and final error counters into buffer. It also calls the driver register dump routine to fill the remainder of buffer.

INPUT TO ROUTINE

Registers	Contents
R4	Address of the device's control/status register
R5	Address of the unit control block

Field	Contents
UCB$L_IRP	Address of the current IRP
IRP$V_DIAGBUF (in IRP$W_STS)	Determines whether diagnostic buffer is present. If set, one exists.
IRP$L_DIAGBUF	Address of the diagnostic buffer, if one is present
UCB$B_ERTCNT	Final error retry count
UCB$L_DDB	Address of the device data block
DDB$L_DDT	Address of the driver dispatch table
DDT$L_REGDUMP	Address of the driver register dump routine
EXE$GQ_SYSTIME	Current system time (time at I/O request completion)
DDT$L_REGDUMP	Address of the driver register dump routine

IPL at execution: caller's IPL

This routine saves the system time and final error count in the diagnostic buffer. It then calls the driver register dump routine, and exits with an RSB instruction.

OUTPUT FROM ROUTINE

Registers	Contents
R0 - R1	Destroyed
R2	Address of the DDT
R3	Address of the I/O request packet
R4	Device CSR register
R5	Address of the unit control block
Fields	**Contents**
---	---

IPL at exit: caller's IPL

IOC$INITIATE

module: IOSUBNPAG

Starts a driver fork process to process an I/O request packet. This routine writes the I/O request packet address and I/O request packet transfer parameters into the unit control block. It also clears device status bits. If the QIO specifies a diagnostic buffer, this routine writes system time into the buffer. It also executes a JMP instruction to transfer control to the driver start I/O routine.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet
R5	Address of the unit control block
Fields	**Contents**
IRP$L_SVAPTE	Address of system buffer (buffered I/O) or address of PTE that maps process buffer (direct I/O).
IRP$W_BOFF	Byte offset of start of buffer
IRP$W_SIZE	Size in bytes of transfer
IRP$V_DIAGBUF (in IRP$W_STS)	Determines whether a diagnostic buffer is present. This field is set if one exists.
IRP$L_DIAGBUF	Address of the diagnostic buffer, if one is present
EXE$GQ_SYSTIME	Current system time (when I/O processing began)

Fields	Contents
UCB$L_DDB	Address of DDB
UCB$L_DDT	Address of DDT
DDT$L_START	Address of driver start I/O routine

IPL at execution: caller's IPL

IOC$INITIATE exits by jumping to the driver start entry specified in the driver dispatch table.

OUTPUT FROM ROUTINE

Registers	Contents
R0 - R1	Destroyed

Fields	Contents
UCB$L_IRP	Address of the start of the I/O request packet
UCB$L_SVAPTE	IRP$L_SVAPTE
UCB$W_BOFF	IRP$W_BOFF
UCB$W_BCNT	IRP$W_BCNT
UCB$V_CANCEL (in UCB$W_STS)	Zero
UCB$V_TIMOUT (in UCB$W_STS)	Zero
diagnostic buffer	Current system time (first quadword)

IPL at exit: caller's IPL

IOC$IOPOST

module: IOCIOPOST

Interrupt service routine that processes I/O request packets in an I/O postprocessing queue. This routine gains control when the processor grants a software interrupt at IPL$_IOPOST. For each queue entry, it adjusts quota use and unlocks pages or deallocates write buffers. It queues a kernel mode AST to copy final I/O status to the IOSB, to copy buffered read data, and to deallocate read buffers. The AST kernel mode routine code is located in module IOCIOPOST. The kernel mode AST routine queues a user mode AST if specified in the QIO. When the postprocessing queue is empty, IOC$IOPOST dismisses the interrupt.

INPUT TO ROUTINE

Registers	Contents
---	---

Fields	Contents
IOC$GL_PSFL	Head of the I/O postprocessing queue. This routine uses this field to locate fields in the IRP.
IRP$L_PID	Process identification of the process that initiated the I/O request. This routine uses this field to locate the PCB.

IPL at execution: IPL$_IOPOST, IPL$_ASTDEL

IOC$IOPOST generates different results for direct and buffered I/O. For direct I/O, the routine unlocks the pages locked for the I/O request and sets the Queue I/O event flag. The pages unlocked include any pages defined in the IRP extension area descriptors (if an IRPE exists). For buffered I/O read functions, the routine copies the data from the system buffer to the process buffer, then releases the system buffer. It also sets a Queue I/O event flag, if one was requested.

For both direct and buffered I/O, IOC$IOPOST performs the following functions:

- Copies the diagnostic buffer from system to process space and releases the system buffer
- Copies I/O completion status (if requested) from the I/O request packet to the process's I/O status block
- Queues an AST to the process, if one was requested
- Deallocates the IRP and any IRP extensions

Note that kernel mode ASTs handle much of the processing described above.

IOC$LOADUBAMAP(A)

module: LOADMREG

Driver fork processes for DMA transfers call this routine to load the UNIBUS map registers required by the current transfer with a page frame number, the data path number, possibly the byte offset bit, and possibly the longword access enable bit. This routine confirms that enough map registers have been allocated and sets the last map register invalid to stop a wild transfer.

INPUT TO ROUTINE

Registers	Contents
R5	Address of unit control block

The data path and map registers are already allocated.

Field	Contents
UCB$W_BOFF	Offset to the first byte in the first page of the transfer
UCB$W_BCNT	Number of bytes in the transfer
UCB$L_CRB	Address of the controller's channel request block
CRB$L_INTD+ VEC$B_DATAPATH	Number of the data path to be allocated
VEC$V_LWAE (in CRB$L_INTD+ VEC$B_DATAPATH)	Determines length of buffering. Set if longword buffering used (instead of quadword buffering)
CRB$L_INTD+VEC$L_NUMREG	Number of map registers allocated
CRB$L_INTD+VEC$L_ADP	Address of the adapter control block
UBA$L_MAP	Address of the first UNIBUS map register
UCB$L_SVAPTE	Address of the page table entry for the first page of the transfer

OUTPUT FROM ROUTINE

Registers	Contents
R0 - R2	Destroyed

Fields	Contents
Allocated map registers	Byte offset is set for entry IOC$LOADUBAMAP (never set for IOC$LOADUBAMAPA)

IPL at exit: caller's IPL

IOC$PURGDATAP

module: LIOSUB

Device drivers using buffered data paths call this subroutine after a data transfer. IOC$PURGDATAP purges the UNIBUS adapter buffered data path as well as checking for and clearing purge errors.

INPUT TO ROUTINE

Registers	Contents
R5	Address of the UCB

Fields	Contents
---	---

IPL at execution: caller's IPL

This routine obtains the start of UNIBUS adapter register space using the following chain of pointers:

UCB$L_CRB ⟶ CRB$L_INTD+VEC$L_ADP ⟶ ADP$L_CSR

This routine extracts the caller's data path number (buffered or direct) from the channel request block. The routine then purges the data path and stores the contents of the data path register in R1. IOC$PURGDATAP clears any purge errors in the data path register. It also sets the appropriate status in R0, computes the base of UNIBUS map registers, and writes the base into R2.

A purge of data path 0 is legal and always results in success status.

IOC$PURGDATAP alters R0 through R3 but preserves all other registers.

OUTPUT FROM ROUTINE

Registers	**Contents**
R0	Low bit set (success) Low bit clear (failure)
R1	Contents of data path after purge (for register dump routine)
R2	Address of the start of UNIBUS map registers (for the register dump routine)
R3	Address of the CRB
Fields	**Contents**
---	---

IPL at exit: caller's IPL

IOC$RELCHAN

module: IOSUBNPAG

Driver fork processes call this routine to release controller data channels assigned to a device. If the channel wait queue contains waiting fork processes, this routine dequeues a process, assigns the channel to that process, restores R3 through R5, and reactivates the suspended process.

INPUT TO ROUTINE

Registers	**Contents**
R5	Address of the unit control block
Fields	**Contents**
UCB$L_CRB	Address of the channel request block
CRB$L_LINK	Address of the secondary CRB

Fields	Contents
CRB$V_BSY (in CRB$B_MASK)	Set if the channel is busy
CRB$L_INTD+VEC$L_IDB	Address of the interrupt data block
IDB$L_OWNER	Channel's owner UCB address
CRB$L_WQFL	Head of the queue of waiting UCBs

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R0 - R2	Destroyed

Fields	Contents
IDB$L_OWNER	Clear (if no driver is waiting for the channel)
CRB$V_BSY	Clear (if no driver is waiting for the channel)

IPL at exit: caller's IPL

IOC$RELDATAP

module: IOSUBNPAG

Driver fork processes call this routine to release a UNIBUS adapter buffered data path. This routine performs no operation if a data path is permanently allocated to the controller. If the data path wait queue contains waiting fork processes, it dequeues a process, allocates the data path to that process, restores R3 through R5, and reactivates the suspended process. This routine should not be called unless the driver owns a buffered data path.

INPUT TO ROUTINE

Registers	Contents
R5	Address of unit control block

Fields	Contents
UCB$L_CRB	Address of the channel request block
CRB$L_INTD+VEC$L_ADP	Address of the adapter control block
CRB$L_INTD+ VEC$B_DATAPATH	Data path specifier

Fields	Contents
VEC$V_PATHLOCK	Set to 1 to indicate that the data path is permanently allocated to the controller
ADP$L_DPQFL	Head of the adapter data path wait queue

IPL at execution: caller's IPL

If the bit map is corrupted, this routine signals a bugcheck with message code INCONSTATE. After IOC$RELDATAP completes successfully, it exits with an RSB instruction.

OUTPUT FROM ROUTINE

Registers	Contents
R0 - R2	Destroyed

Fields	Contents
ADP$W_DPBITMAP	Data path is set to free if not allocated to another driver fork process
bits 0 through 4 (in CRB$L_INTD+ VEC$B_DATAPATH)	Clear

IPL at exit: caller's IPL

IOC$RELMAPREG

module: IOSUBNPAG

Driver fork processes call this routine to release a set of UNIBUS adapter map registers. This routine performs no operation if map registers are permanently allocated to the controller. If the map register wait queue contains waiting fork processes, it dequeues a process and attempts to allocate the required set of map registers. If successful, it restores R3 through R5 and reactivates the suspended process. If not successful, it reinserts the fork process in the map register wait queue and dequeues the next process. This routine assumes that the caller is the current owner of the controller data channel.

INPUT TO ROUTINE

Registers	Contents
R5	Address of unit control block

Fields	Contents
UCB$L_CRB	Address of the CRB
VEC$V_MAPLOCK (in CRB$L_INTD+ VEC$W_MAPREG)	If set, indicates that map registers are permanently allocated to the controller

Fields	Contents
CRB$L_INTD+VEC$L_ADP	Address of the adapter control block
CRB$L_INTD+ VEC$W_MAPREG	Number of the starting map register
CRB$L_INTD+ VEC$B_NUMREG	Number of map registers to release
ADP$L_MRQFL	Head of the queue of waiting drivers

IPL at execution: caller's IPL

IOC$RELMAPREG calls IOC$ALTUBAMAP and IOC$ALOUBAMAP. It exits with an RSB instruction.

OUTPUT FROM ROUTINE

Registers	Contents
R0 - R2	Destroyed

Fields	Contents
ADP$W_MRBITMAP	Map registers set to free

IPL at exit: caller's IPL

IOC$RELSCHAN

module: IOSUBNPAG

This routine releases a secondary controller's data channel; that is, the MASSBUS adapter controller data channel. For more information, refer to Appendix F.

This routine has the same inputs and outputs as IOC$RELCHAN.

IOC$REQCOM

module: IOSUBNPAG

Driver fork processes call this routine after a device I/O operation and all device-dependent processing of an I/O request are complete. This routine writes R0 and R1 into the I/O request packet status field. It then inserts the I/O request packet into the I/O postprocessing queue. If error logging is occurring, it writes final status into the error message buffer and calls ERL$RELEASEMB. If the I/O request packet wait queue contains entries, it dequeues an I/O request packet and calls IOC$INITIATE. Otherwise, it clears a unit control block busy status bit to indicate that the device is idle.

INPUT TO ROUTINE

Registers	Contents
R0	First longword of I/O status
R1	Second longword of I/O status
R5	Address of unit control block

Fields	Contents
UCB$V_ERLOGIP (in UCB$W_STS)	Set or clear. Determines whether error logging should be performed
UCB$W_STS	Final device status
UCB$B_ERTCNT	Final error counters
UCB$L_EMB	Address of the error log message buffer
UCB$L_IRP	Address of the IRP

IPL at execution: caller's IPL

This routine places the I/O request packet in the queue headed by IOC$GL_PSBL. If UCB$L_IOQEL has a packet queued to it, IOC$REQCOM sends the packet to IOC$INITIATE. This routine exits by branching to IOC$RELCHAN.

OUTPUT FROM ROUTINE

Registers	Contents
R2 - R3	Destroyed

If IOC$INITIATE is called, other registers will be destroyed.

Fields	Contents
IRP$L_MEDIA	I/O status (R0)
IRP$L_MEDIA+4	I/O status (R1)
EMB$Q_IOSB	I/O status (R0 and R1)
UCB$L_OPCNT	Incremented by 1
EMB$B_ERTCNT	UCB$B_ERTCNT
EMB$B_ERTCNT+1	UCB$B_ERRCNT
EMB$W_DV_STS	UCB$W_STS
UCB$V_BSY (in UCB$W_STS)	Clear (if no more packets in queue)

IPL at exit: caller's IPL

IOC$REQDATAP(NW)

module: IOSUBNPAG

Driver fork processes call this routine to request a UNIBUS adapter buffered data path for a DMA transfer. This routine performs no operation if a data path is permanently allocated to the controller. This routine locates a free data path and writes the data path number in the CRB. If no data paths are free, it saves R3 and R4 in the UCB fork block, inserts the fork block address in a data path wait queue, and suspends the driver fork process.

INPUT TO ROUTINE

Registers	**Contents**
R5	Address of unit control block
0(SP)	Caller's return address
4(SP)	Return address of the caller's caller
Fields	**Contents**
UCB$L_CRB	Address of the channel request block
VEC$V_PATHLOCK (in CRB$L_INTD+ VEC$B_DATAPATH)	If set, indicates that the data path already is allocated
CRB$L_INTD+VEC$L_ADP	Address of the adapter control block
ADP$W_DPBITMAP	Indicates what data paths are available

IPL at execution: caller's IPL

If IOC$REQDATAP cannot allocate a data path, and NW is not specified, the routine saves process context by placing the contents of R3, R4 and the PC in the UCB fork block and placing R5 in the data path wait queue (ADP$L_DPQBL). If, however, NW is specified, the routine does not suspend the process to wait for the data path.

OUTPUT FROM ROUTINE

Registers	**Contents**
R0	SS$_NORMAL (success) 0 (failure)
Fields	**Contents**
CRB$L_INTD+ VEC$B_DATAPATH	Data path number
ADP$W_DPBITMAP	Bit for allocated data path clear

IPL at exit: caller's IPL

If the channel is busy, this routine saves driver context by storing the contents of R3 and R4 in UCB$L_FR3 and UCB$L_FR4, respectively, storing 0(SP) in UCB$L_FPC and placing the contents of R5 in the CRB wait queue (CRB$W_WQFL).

IOC$REQPCHANH exits by issuing an RSB instruction.

OUTPUT FROM ROUTINE

Registers	**Contents**
R0 - R2	Destroyed
R4	IDB$L_CSR
Fields	**Contents**
IDB$L_OWNER	R5

IPL at exit: caller's IPL

IOC$REQPCHANL

module: IOSUBNPAG

Driver fork processes call this routine to request a channel on the primary controller with low priority. This routine performs in the same manner as IOC$REQPCHANH, except that, should driver have to wait for the channel, IOC$REQPCHANL places the UCB at the end of the channel wait queue.

IOC$REQSCHANH

module: IOSUBNPAG

Driver fork processes call this routine to request a channel on the secondary controller with high priority.

The input to and output from this routine are the same as for IOC$REQPCHANH, except that the secondary controller data channel is assigned.

IOC$REQSCHANL

module: IOSUBNPAG

Driver fork processes call this routine to request a channel on the secondary controller with low priority.

The input to and output from this routine are the same as for IOC$REQPCHANH, except that the secondary controller data channel is assigned.

IOC$RETURN

module: IOSUBNPAG

This routine merely returns by issuing an RSB instruction. It has no input requirements and produces no output.

IOC$VERIFYCHAN

module: IOSUBPAGD

Drivers call this routine to validate a user-supplied channel number, contruct a channel index, and obtain the address of the channel control block to which the channel number points.

INPUT TO ROUTINE

Registers	**Contents**
R0	Channel number
Fields	**Contents**
CTL$GL_CCBASE	Base address of the process channel control block table

IPL at execution: IPL$_ASTDEL or below

Since this routine gains access to information stored in user process virtual address space, it should only be called when the user process is mapped.

IOC$REQMAPREG

module: IOSUBNPAG

Driver fork processes call this routine to request a set of UNIBUS adapter map registers for a DMA transfer. This routine performs no operation if map registers are permanently allocated to the controller. This routine locates the required number of map registers and writes the number of registers and the number of the first register into the CRB. If sufficient map registers are not available, it saves R3 and R4 in the UCB fork block, inserts the fork block address in a map register wait queue, and suspends the driver fork process.

INPUT TO ROUTINE

Registers	**Contents**
R5	Address of unit control block
0(SP)	Return address of caller
4(SP)	Return address of the caller's caller
Fields	**Contents**
UCB$W_BCNT	Transfer byte count
UCB$W_BOFF	Byte offset into page of start of buffer
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of the adapter control block
VEC$V_MAPLOCK (in CRB$L_INTD+ VEC$W_MAPREG)	Determines status of map lock bit
ADP$W_MRBITMAP	Adapter map register allocation bit map

IPL at execution: caller's IPL

If registers are not available, this routine suspends the process by saving the following context:

- R3 and R4 are saved in UCB$L_FR3 and UCB$L_FR4, respectively.
- PC is saved in UCB$L_FPC.
- R5 is saved in ADP$L_MRQBL, which is the adapter's map register wait queue.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL (success)
R1 - R2	Destroyed

Fields	Contents
CRB$L_INTD+ VEC$W_MAPREG	Starting map register number of those allocated
CRB$L_INTD+ VEC$B_NUMREG	Number of map registers allocated
ADP$W_MRBITMAP	Allocated map registers

IPL after execution: caller's IPL

IOC$REQPCHANH

module: IOSUBNPAG

Driver fork processes call this routine to request a channel on the primary controller with high priority. If the controller data channel is idle, this routine writes the UCB address in the interrupt data block and returns the CSR address in R4. Otherwise, it saves R3 in the UCB fork block, inserts the fork block address at the front of the channel wait queue, and suspends the driver fork process.

INPUT TO ROUTINE

Registers	Contents
R5	Address of unit control block
0(SP)	Return address of the caller
4(SP)	Return address of the caller's caller

Fields	Contents
UCB$L_CRB	Address of the channel request block
CRB$L_LINK	Address of the secondary channel request block
CRB$L_INTD+VEC$L_IDB	Interrupt data block address
CRB$V_BSY in CRB$B_MASK	Set or clear. If set, indicates that the channel is busy
IDB$L_CSR	Address of device CSR

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS$_NORMAL (success) SS$_IVCHAN (invalid channel number) SS$_NOPRIV (no privilege to access specified channel)
R1	Address of channel control block
R2	Channel index number

Fields	Contents
---	---

IPL at exit: caller's IPL

IOC$WFIKPCH

module: IOSUBNPAG

Driver fork processes call this routine to suspend driver processing to wait for an interrupt or device timeout and still retain the controller data channel. This routine saves R3, R4, and the driver's return PC from top of stack in the UCB fork block. It sets UCB bits to indicate that an interrupt or a timeout is expected and sets the timeout time in the unit control block. It clears the UCB bit that indicates that the unit is timed out and lowers IPL back to the IPL saved on top of stack. Then, it returns to the caller of the driver fork process.

The two bytes following the JSB to IOC$WFIKPCH contain the relative offset to the timeout routine.

INPUT TO ROUTINE

Registers	Contents
R5	Address of unit control block
0(SP)	Address following the JSB to IOC$WFIKPCH
4(SP)	Timeout value in seconds
8(SP)	IPL to which to lower before returning to the caller's caller
12(SP)	Return address of the caller's caller

Field	Contents
EXE$GL_ABSTIM	Absolute time. Used to compute time at which device times out

IPL at execution: Fork a device IPL (caller's IPL)

This routine removes 0(SP) through 11(SP) from the stack explicitly and 12(SP) through 15(SP) implicitly by exiting with an RSB instruction, which returns to the caller's caller.

OUTPUT FROM ROUTINE

Registers	Contents
---	---
Fields	**Contents**
UCB$L_DUETIM	Sum of timeout value and EXE$GL_ABSTIM
UCB$V_INT	Set to indicate that interrupts are expected on the device
UCB$V_TIM	Set to indicate that timeouts are expected on the device
UCB$V_TIMOUT	Cleared to indicate that unit is not timed out
UCB$L_FR3	R3
UCB$L_FR4	R4
UCB$FPC	0(SP)+2

IPL at exit: IPL specified in 8(SP)

IOC$WFIRLCH

module: IOSUBNPAG

Driver fork processes call this routine to suspend driver processing to wait for an interrupt or device timeout first releasing the controller data channel.

The input to and output from this routine are the same as IOC$WFIKPCH except that IOC$WFIRLCH exits to IOC$RELCHAN, which releases the controller dates channel.

APPENDIX D

SAMPLE DRIVER FOR AN ANALOG TO DIGITAL CONVERTER

This appendix contains the source listing of a driver for an analog-to-digital converter.

A machine-readable copy of this driver is also available. Its file specification is:

SYS$EXAMPLES:ADDRIVER.MAR

```
        .TITLE  ADDRIVER - VAX/VMS AD11-K DRIVER
        .IDENT  'V03-002'
;
;*****************************************************************************
;*
;*  COPYRIGHT (c) 1978, 1979, 1980, 1982
;*  BY DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS
;*
;*  THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND  COPIED
;*  ONLY  IN  ACCORDANCE  WITH  THE  TERMS  OF  SUCH  LICENSE AND WITH THE
;*  INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY  OTHER
;*  COPIES  THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
;*  OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS  HEREBY
;*  TRANSFERRED.
;*
;*  THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT  NOTICE
;*  AND  SHOULD  NOT  BE  CONSTRUED  AS  A COMMITMENT BY DIGITAL EQUIPMENT
;*  CORPORATION.
;*
;*  DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF  ITS
;*  SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
;*
;*****************************************************************************
;
;
;++
;
; FACILITY:
;
;       VAX/VMS AD11-K I/O DRIVER
;
; ABSTRACT:
;
; DEVICE TABLES AND DRIVER CODE FOR THE AD11-K ANALOGUE
; TO DIGITAL CONVERTER WITH OPTIONAL AM11-K MULTIPLEXER.
;
; AUTHOR:
;
;       S. PROGRAMMER, SEPTEMBER 1978.
;
; MODIFIED BY:
;
;--
```

```
        .SBTTL  FUNCTIONAL DESCRIPTION OF DRIVER
;+
; THE DRIVER SUPPORTS A/D SAMPLING ON GROUPS OF CHANNELS VIA QIO
; READ REQUESTS.  NO EXTERNALLY TRIGGERED SAMPLING (I.E., CLOCK
; OVERFLOW OR SCHMITT TRIGGER) IS SUPPORTED.  THE AM11-K MULTIPLEXER
; MAY BE PRESENT, BUT NO AUTOMATIC RANGING AMPLIFICATION IS
; DONE AT DRIVER LEVEL.  THE BUILT-IN DAC MAY BE USED FOR TESTING VIA
; A LOOPBACK QIO FUNCTION DEFINED ESPECIALLY FOR THIS DEVICE.
;
; THE QIO FUNCTIONS AVAILABLE ARE:
;
; IO$_READVBLK          -READ VIRTUAL BLOCK
; IO$_READLBLK          -READ LOGICAL BLOCK
; IO$_READPBLK          -READ PHYSICAL BLOCK=IO$_LOOPBACK
; IO$_LOOPBACK          -WRITE DAC, READ RESULTS; REQUIRES
;                        PHYSICAL I/O PRIVILEGE
;
; THE STANDARD QIO PARAMETERS ARE:
;
; P1=BUFFER ADDRESS
; P2=BUFFER BYTE COUNT
; P3=SPECIFIER OF CHANNELS TO SAMPLE:
;       BIT 0-7/INITIAL CHANNEL # (0-63)
;       BIT 8-15/TOTAL # OF CHANNELS TO SAMPLE (1-64)
;       BIT 16-23/CHANNEL INCREMENT (0-63)
;       BIT 24-31/IGNORED
; P4=DAC VALUE, USED FOR LOOPBACK ONLY:
;       BIT 0-7/8 BIT DAC VALUE
;       BIT 8-31/IGNORED
; P5,P6 ARE NOT USED
;
; IN ADDITION TO THE STANDARD STATUS CODES THAT CAN BE RETURNED FOR
; A QIO, THE FOLLOWING DEVICE-SPECIFIC I/O STATUS VALUES ARE DEFINED:
;
; SS$_DATAOVERUN        -ERROR BIT SET IN CSR; SAMPLING ABORTED
;                        WITH LAST GOOD SAMPLE IN BUFFER
; SS$_BADPARAM          -INVALID CHANNEL SPECIFIER; NO SAMPLES TAKEN
; SS$_BUFFEROVF         -USER BUFFER OVERRUN; AS MANY CHANNELS AS WILL
;                        FIT ARE SAMPLED
;
; THE SAMPLES ARE RETURNED IN THE CALLER'S BUFFER PACKED ONE SAMPLE
; PER WORD, BITS 0-11. THE BYTE COUNT RETURNED IN THE SECOND WORD OF
; THE I/O STATUS BLOCK ALWAYS REFLECTS THE # OF BYTES ACTUALLY FILLED
; WITH SAMPLE DATA.  THE NUMBER OF SAMPLES IS ONE HALF THE RETURNED
; BYTE COUNT.
;
; EXAMPLE: SWEEP THROUGH 32 INPUTS CONNECTED IN DIFFERENTIAL MODE
;          (AD11-K AND AM11-K):
;
; SWEEPBUF:     .BLKW   32
; NUMINPUT:     .LONG   32
; CHANSPEC:     .BYTE   0,32,2          ;START WITH CHANNEL 0;
;                                       ; SAMPLE CHANNELS 0,2,4,...,62
;
;               $QIO_S  CHAN=X,FUNC=IO$_READVBLK,-
;                       P1=SWEEPBUF,P2=NUMINPUT,P3=CHANSPEC
;-
```

```
        .SBTTL  MACRO LIBRARY CALLS

;
; EXTERNAL SYMBOLS (LIB/LIB):
;

        $CRBDEF                         ;CHANNEL REQUEST BLOCK
        $DDBDEF                         ;DEVICE DATA BLOCK
        $IDBDEF                         ;INTERRUPT DATA BLOCK
        $IODEF                          ;I/O FUNCTION CODES
        $IPLDEF                         ;HARDWARE IP DEFINITIONS
        $IRPDEF                         ;I/O REQUEST PACKET
        $UCBDEF                         ;UNIT CONTROL BLOCK
        $VECDEF                         ;INTERRUPT VECTOR BLOCK
        $JIBDEF                         ;JOB INFORMATION BLOCK

;
; USER DEFINED EXTERNAL SYMBOLS ARE CONTAINED IN A USER LIBRARY .
; THE CONTENTS OF THIS LIBRARY CAN BE MERGED WITH THE SYSTEM LIBRARY
; TO ALLOW USER PROGRAMS TO USE EXTENDED FUNCTION CODES WITHOUT HAVING
; TO DEFINE THEM LOCALLY.
; THIS DRIVER MUST BE ASSEMBLED WITH A USER LIBRARY TO DEFINE $XIODEF.
;
        $XIODEF                         ;EXTENDED QIO FUNCTIONS.THIS MACRO
                                        ;CONTAINS THE DEFINITIONS FOR
                                        ;IO$_LOOPBACK
```

```
        .SBTTL  LOCAL DEFINITIONS

;
; LOCAL DEFINITIONS:
;
; QIO ARGUMENT LIST OFFSETS:
;

P1=0                                    ;FIRST,
P2=4                                    ; SECOND,
P3=8                                    ; THIRD,
P4=12                                   ; FOURTH,
P5=16                                   ; FIFTH,
P6=20                                   ; AND SIXTH PARAMETERS

;
; DEVICE PARAMETERS:
;

DAC_TIMER=20                            ;20 USEC TIMER FOR DAC SETTLE
MAX_INLCHN=63                           ;MAXIMUM INITIAL CHANNEL #,
MAX_NUMCHN=64                           ; NUMBER OF CHANNELS,
MAX_INCCHN=63                           ; AND CHANNEL INCREMENT
ADC_TIMER=2                             ;A/D CONVERSION TIMEOUT=2 SEC

;
; DEVICE REGISTER DEFINITIONS:
;

        $DEFINI AD

$DEF    AD_CSR  .BLKW   1               ;CONTROL/STATUS REGISTER

        _VIELD  AD_CSR,0,<-             ;DEFINE CSR FIELDS: AD_CSR_M_XXX
        <GO,,M>,-                       ; START A/D CONVERSION
        <,3>,-                          ; 3 UNUSED BITS
        <EXT,,M>,-                      ; EXTERNAL START ENABLE
        <COV,,M>,-                      ; CLOCK OVERFLOW ENABLE
        <IE,,M>,-                       ; INTERRUPT ENABLE
        <DON,,M>,-                      ; CONVERSION DONE FLAG
        <MUX,6,M>,-                     ; 6 BIT MUX CHANNEL #
        <,1>,-                          ; BIT 14 IS UNUSED
        <ERR,,M>,-                      ; ERROR FLAG
        >                               ;END OF CSR FIELDS
$DEF    AD_DBR  .BLKW   1               ;A/D DATA BUFFER REGISTER
.=.-2                                   ;DATA BUFF REG=DAC BUFF REG
$DEF    AD_DAC  .BLKW   1               ;DAC DATA BUFFER REF

        $DEFEND AD                      ;END OF A/D REGISTER DEFNS
```

```
;
; DEVICE DEPENDENT UCB EXTENSIONS:
;

        $DEFINI UCB

.=UCB$K_LENGTH                              ;STEP TO END OF STANDARD UCB
                                            ;NOTE:  NEXT 4 BYTES ASSUMED
                                            ; ADJACENT
$DEF    UCB$B_AD_CURCHN .BLKB   1           ;CURRENT MUX CHANNEL #
$DEF    UCB$B_AD_NUMCHN .BLKB   1           ;# CHANNELS LEFT TO SAMPLE
$DEF    UCB$B_AD_INCCHN .BLKB   1           ;CHANNEL INCREMENT
                        .BLKB   1           ;SPARE BYTE
$DEF    UCB$W_AD_CSR    .BLKW   1           ;SAVED CSR
        $FIELD  UCB$W_CSR,1,<-              ;BORROW UNUSED CSR BIT
        <BFO,,M>,-                          ; FOR USER BUFFER OVERRUN
        >
UCB$K_ADLENGTH=.                            ;LENGTH OF A/D UCB

        $DEFEND UCB                         ;END OF UCB EXTENSIONS
;
; A/D DRIVER USE OF TEMPORARY IRP STORAGE:
;
IRP$L_CHSPEC=IRP$L_MEDIA                    ;CHANNEL SPECIFIER(P3)
IRP$L_DACVAL=IRP$L_MEDIA+4                  ;OPTIONAL DAC VALUE(P4)
```

```
        .SBTTL  DRIVER PROLOGUE AND DISPATCH TABLES

;
; DRIVER PROLOGUE TABLE:
;

        DPTAB   -                       ;DEFINE DRIVER PROLOGUE TABLE:
                END=AD_END,-            ; END OF DRIVER,
                ADAPTER=UBA,-           ; UNIBUS ADAPTER,
                UCBSIZE=UCB$K_ADLENGTH,-; SIZE OF A/D UCB,
                NAME=ADDRIVER           ; DRIVER NAME
                                        ;
        DPT_STORE       INIT            ;VALUES TO BE SET ON LOAD
        DPT_STORE       UCB,UCB$B_FIPL,B,8      ;DEVICE FORK IPL
        DPT_STORE       UCB,UCB$B_DIPL,B,22     ;AD11 HARDWARE IPL
        DPT_STORE       UCB,UCB$L_DEVCHAR,L,-   ;AD11 DEVICE CHARACTERISTICS
                        <DEV$M_AVL-             ; AVAILABLE,
                        !DEV$M_IDV-             ; INPUT DEVICE,
                        !DEV$M_RTM>             ; REALTIME DEVICE
                                                ;
        DPT_STORE       REINIT                  ;VALUES TO SET ON RELOAD
        DPT_STORE       CRB,CRB$L_INTD+4,D,-    ;INTERRUPT SERVICE ADDR
                        AD_INTERRUPT
        DPT_STORE       CRB,-                   ;ADDR OF CONTROLLER
                        CRB$L_INTD+VEC$L_INITIAL,- ; INITIALIZATION
                        D,AD_CTLINIT
        DPT_STORE       CRB,-                   ;ADDR OF UNIT
                        CRB$L_INTD+VEC$L_UNITINIT,- ; INITIALIZATION
                        D,AD_UNITINIT
        DPT_STORE       DDB,DDB$L_DDT,D,-       ;ADDR OF DRIVER
                        AD$DDT                  ; DISPATCH TABLE
                                                ;
        DPT_STORE       END                     ;END DRIVER PROLOGUE

;
; DRIVER DISPATCH TABLE:
;

        DDTAB   -                       ;DDT CREATION MACRO
                DEVNAM=AD,-             ;NAME OF DEVICE
                START=AD_STARTIO,-      ;ADDR OF START I/O ROUTINE
                FUNCTB=AD_FUNCTABLE     ;ADDR OF FDT
```

```
        .SBTTL  AD11-K FUNCTION DECISION TABLE

;
; AD11 FUNCTION DECISION TABLE:
;

AD_FUNCTABLE:                           ;FUNCTION DECISION TABLE START
        FUNCTAB ,-                      ;LEGAL FUNCTIONS:
                <LOOPBACK,-             ; LOOPBACK READ FROM DAC
                READPBLK,-              ; READ PHYSICAL BLOCK
                READLBLK,-              ; READ LOGICAL BLOCK
                READVBLK>               ; READ VIRTUAL BLOCK
        FUNCTAB ,-                      ;BUFFERED I/O FUNCTIONS:
                <LOOPBACK,-             ; LOOPBACK READ FROM DAC
                READPBLK,-              ; READ PHYSICAL BLOCK
                READLBLK,-              ; READ LOGICAL BLOCK
                READVBLK>               ; READ VIRTUAL BLOCK
        FUNCTAB -                       ;PREPROCESSING ROUTINES:
                AD_READ,-               ;CALL SINGLE PREPROCESSOR FOR:
                <LOOPBACK,-             ; LOOPBACK READ FROM DAC
                READPBLK,-              ; READ PHYSICAL BLOCK
                READLBLK,-              ; READ LOGICAL BLOCK
                READVBLK>               ; AND READ VIRTUAL BLOCK
```

```
        .SBTTL  AD_READ:        READ FUNCTION PROCESSING
;+
; AD_READ - READ FUNCTION PREPROCESSING
;
; THIS ROUTINE IS CALLED FROM THE FUNCTION DECISION TABLE DISPATCHER
; TO PROCESS A READ PHYSICAL, READ LOGICAL, READ VIRTUAL, OR LOOPBACK
; I/O FUNCTION.
;
; AD_READ FIRST VERIFIES THE CALLER'S PARAMETERS, TERMINATING THE
; REQUEST WITH IMMEDIATE SUCCESS OR ERROR IF NECESSARY.  P3 AND
; P4 ARE STORED IN THE IRP.  A SYSTEM BUFFER IS ALLOCATED AND
; ITS ADDRESS IS SAVED IN THE IRP.  THE CALLER'S QUOTA IS UPDATED,
; AND THE READ REQUEST IS QUEUED TO THE DRIVER FOR STARTUP.
;
; INPUTS:
;
;       R0,R1,R2 = SCRATCH
;       R3 = IRP ADDRESS
;       R4 = ADDR OF PCB FOR CURRENT PROCESS
;       R5 = DEVICE UCB ADDRESS
;       R6 = ADDRESS OF CCB
;       R7 = I/O FUNCTION CODE
;       R8 = FDT DISPATCH ADDR
;       R9-R11 = SCRATCH
;       AP = ADDR OF FUNCTION PARAMETER LIST
;
; OUTPUTS:
;
;       R0,R1,R2 = DESTROYED
;       R3-R11,AP = PRESERVED
;       IRP$L_CHSPEC(R3) = CHANNEL SPECIFIER (P3)
;       IRP$L_DACVAL(R3) = OPTIONAL DAC VALUE (P4)
;       IRP$L_SVAPTE(R3) = ADDR OF ALLOCATED SYSTEM BUFFER
;       IRP$W_BOFF(R3) = REQUESTED BYTE COUNT
;
;       SYSTEM BUFFER:
;               LONGWD 0/ADDR OF START OF DATA=BUFF ADDR+12
;               LONGWD 1/ADDR OF USER BUFFER
;               LONGWD 2/DATA STRUCTURE BOOKKEEPING
;-

        .ENABL  LSB

AD_READ:                                ;READ FUNCTION PREPROCESSING
        MOVZWL  P2(AP),R1               ;GET USER BYTE COUNT
        BEQL    10$                     ;BRANCH IF READ OF 0 BYTES
                                        ; (=INSTANT SUCCESS)
        MOVZWL  #SS$_BADPARAM,R0        ;ASSUME CHANNEL SPEC ERROR
        MOVAL   P3(AP),R2               ;GET ADDR OF CHANNEL SPEC
        CMPB    (R2)+,#MAX_INLCHN       ;INITIAL CHAN # TOO LARGE?
        BGTRU   20$                     ;BRANCH IF SO
        TSTB    (R2)                    ;# CHANNELS = 0?
        BEQL    10$                     ;BRANCH IF SO (SUCCESS)
        CMPB    (R2)+,#MAX_NUMCHN       ;# CHANNELS TO SAMPLE TOO LARGE?
        BGTRU   20$                     ;BRANCH IF SO
        CMPB    (R2),#MAX_INCCHN        ;CHANNEL INCREMENT TOO LARGE?
        BGTRU   20$                     ;BRANCH IF SO
        MOVQ    P3(AP),IRP$L_CHSPEC(R3) ;STORE P3 AND P4 (OPTIONAL DAC)
                                        ; IN IRP UNTIL REQUEST EXECUTION
```

```
        MOVL    P1(AP),R0               ;GET ADDR OF USER BUFFER
        JSB     G^EXE$READCHK           ;VERIFY THAT CALLER HAS
                                        ; WRITE ACCESS TO BUFFER
        PUSHR   #^M<R0,R3>              ;SAVE USER BUFF ADDR, IRP ADDR
        ADDL    #12,R1                  ;ADD 12 BYTES TO REQUESTED BUFF
                                        ; SIZE FOR BUFF HEADER
        JSB     G^EXE$BUFFRQUOTA        ;VERIFY BUFFER SPACE LEFT
                                        ; IN CALLER'S QUOTA
        BLBC    R0,30$                  ;BRANCH IF INSUFFICIENT QUOTA
        JSB     G^EXE$ALLOCBUF          ;ALLOCATE A SYSTEM BUFFER
        BLBC    R0,30$                  ;BRANCH IF NONE AVAILABLE
        POPR    #^M<R0,R3>              ;RESTORE USER BUFFER, IRP ADDR
        MOVL    R2,IRP$L_SVAPTE(R3)     ;SAVE ADDR OF SYSTEM BUFFER
        MOVW    R1,IRP$W_BOFF(R3)       ; AND REQUESTED BYTE COUNT
        MOVZWL  R1,R1                   ;CONVERT TO LONGWORD
        MOVL    PCB$L_JIB(R4),R4        ;GET JOB INFORMATION BLOCK ADDRESS
        SUBL    R1,JIB$L_BYTCNT(R4)     ;DEDUCT REQUESTED BYTE COUNT
                                        ; FROM PROCESS' QUOTA
        MOVAB   12(R2),(R2)+            ;SAVE ADDR OF START OF USER DATA
                                        ; IN 1ST LONGWD OF SYSTEM BUFFER
        MOVL    R0,(R2)                 ;SAVE USER BUFFER ADDR IN
                                        ; 2ND LONGWD
        JMP     G^EXE$QIODRVPKT         ;QUEUE I/O PKT TO DRIVER

;
; COME HERE IF USER REQUESTED READ OF 0 BYTES OR 0 CHANNELS.
; THIS IS ALWAYS SUCCESSFUL AND DOES NO DEVICE I/O:
;

10$:    MOVZWL  #SS$_NORMAL,R0          ;SET NORMAL COMPLETION STATUS
20$:    JMP     G^EXE$FINISHIOC         ;COMPLETE I/O REQUEST

;
; COME HERE TO ABORT I/O REQUEST WITH EXCEPTION STATUS IN R0:
;

30$:    POPR    #^M<R2,R3>              ;CLEAR BUFFER ADDR; RESTORE IRP
                                        ; ADDR
        JMP     G^EXE$ABORTIO           ;COMPLETE I/O REQUEST

        .DSABL  LSB
```

```
        .SBTTL  AD_STARTIO:     PERFORM A/D CONVERSIONS
;+
; AD_STARTIO - START I/O OPERATION ON AD11-K A/D CONVERTER.
;
; THIS ROUTINE IS ENTERED WHEN THE ASSOCIATED UNIT IS IDLE AND A
; PACKET IS AVAILABLE FOR PROCESSING.
;
; TO PREPARE FOR SAMPLING, AD_STARTIO PERFORMS THESE STEPS:
;
; 1. SET UP UCB WITH CHANNEL SPECIFIER AND ADDRESS IN SYSTEM
;    BUFFER TO HOLD FIRST SAMPLE.
; 2. IF LOOPBACK WAS SPECIFIED, THE DAC IS SET WITH THE CALLER-
;    SPECIFIED VALUE.
;
; THE DRIVER THEN LOOPS FROM AD_NXTSAMPLE TO AD_ENDSAMPLE
; COLLECTING SAMPLES UNTIL ALL SAMPLES HAVE BEEN COLLECTED,
; OR AN ERROR OCCURS.  AN INTERRUPT IS RECEIVED FOR EACH SAMPLE,
; BUT, TO SAVE TIME, THE DRIVER NEVER FORKS UNTIL TIME TO
; COMPLETE THE I/O REQUEST.
;
; INPUTS:
;
;       R3 = ADDR OF IRP
;       R5 = ADDR OF DEVICE UNIT UCB
;
; OUTPUTS:
;
;       R0,R1,R2 = DESTROYED
;       OTHER REGISTERS ARE PRESERVED
;-

        .ENABL  LSB

AD_STARTIO:                                     ;START NEXT QIO
        MOVL    IRP$L_CHSPEC(R3),-              ;COPY CHANNEL SPEC FROM
                UCB$B_AD_CURCHN(R5)             ; IRP TO UCB
        MOVL    @IRP$L_SVAPTE(R3),-             ;SET ADDR OF START DATA
                UCB$L_SVAPTE(R5)                ; IN UCB
        MOVL    UCB$L_CRB(R5),R4                ;GET CRB ADDRESS,
        MOVL    @CRB$L_INTD+VEC$L_IDB(R4),R4 ; THEN CSR ADDRESS
        BICB3   #^C<IO$M_FCODE>,-               ;GET THE I/O
                IRP$W_FUNC(R3),R0               ; FUNCTION CODE
        CMPB    R0,#IO$_LOOPBACK                ;LOOPBACK?
        BNEQ    AD_NXTSAMPLE                    ;BRANCH IF NOT
        MOVZBW  IRP$L_DACVAL(R3),-              ;SET DAC VALUE IN
                AD_DAC(R4)                      ; DAC BUFFER REGISTER
        MFPR    S^#PR$_ICR,R1                   ;GET CURRENT INTERVAL COUNTER (USEC)
        ADDL    #DAC_TIMER,R1                   ; +DAC SETTLE TIME IN USEC
        BLSS    10$                             ;BRANCH IF COUNTER DOESN'T
                                                ; OVERFLOW
        MOVAW   -10000(R1),R1                   ;ELSE CALCULATE COUNTER
                                                ; FOR NEXT INTERVAL
10$:    MFPR    S^#PR$_ICR,R0                   ;READ INTERVAL COUNTER NOW
        CMPL    R0,R1                           ;REACHED SETTLE TIME YET?
        BLSS    10$                             ;BRANCH IF NOT
```

```
AD_NXTSAMPLE:                                   ;START NEXT SAMPLE
        MOVZBW  #AD_CSR_M_IE!AD_CSR_M_GO,R0 ;SET INTERRUPT ENABLE AND
                                                ; START A/D CONVERSION
        INSV    UCB$B_AD_CURCHN(R5),-           ;SET MUX CHAN #
                #8,#6,R0                        ; FOR CSR
        DSBINT                                  ;DISABLE INTERRUPTS (IPL=IPL$POWER)
        BBSC    #UCB$V_POWER,-                  ;BRANCH IF POWER FAILURE
                UCB$W_STS(R5),AD_POWERFAIL ; AND CLEAR POWER FAIL SIGNAL
        MOVW    R0,AD_CSR(R4)                   ;SET CSR
        WFIKPCH AD_TIMEOUT,#ADC_TIMER   ;WAIT FOR INTERRUPT, OR TIMEOUT
        MOVW    AD_CSR(R4),UCB$W_AD_CSR(R5) ;SAVE CSR IN UCB
        BLSS    AD_CSRERROR                     ;BRANCH IF ERROR
        MOVW    AD_DBR(R4),@UCB$L_SVAPTE(R5) ;COPY A/D VALUE INTO
                                                ; SYSTEM BUFFER
        ADDL    #2,UCB$L_SVAPTE(R5)             ;STEP BUFFER POINTER
        SUBL    #2,UCB$W_BCNT(R5)               ;DECREASE # BYTES LEFT IN REQUEST
        DECB    UCB$B_AD_NUMCHN(R5)             ;DECR # CHANNELS LEFT TO SAMPLE
        BEQL    AD_DONE                         ;BRANCH IF NONE
        CMPW    UCB$W_BCNT(R5),#2               ;AT LEAST 2 BYTES LEFT IN BUFFER?
        BLSSU   AD_BUFFEROVF                    ;BRANCH IF NOT
        BICW    #UCB$W_CSR_M_BFO,-              ;ELSE CLEAR BUFFER OVERRUN
                UCB$W_AD_CSR(R5)                ; BIT IN CSR COPY
        ADDB    UCB$B_AD_INCCHN(R5),-           ;NEXT CHANNEL # =
                UCB$B_AD_CURCHN(R5)             ; CURRENT CHANNEL+INCREMENT
        BICB    #^C<MAX_NUMCHN-1>,-             ; MODULO MAXIMUM
                UCB$B_AD_CURCHN(R5)             ; CHANNEL #

AD_ENDSAMPLE:                                   ;THIS SAMPLE COMPLETE
        BRB     AD_NXTSAMPLE                    ;GO START NEXT SAMPLE

        .DSABL  LSB
```

```
        .SBTTL  -               I/O REQUEST COMPLETION

;
; COME HERE TO COMPLETE I/O REQUEST WITH NORMAL OR ERROR STATUS.
;
; USER BUFFER OVERRUN, I.E., NO MORE SAMPLES CAN BE COLLECTED:
;

        .ENABL  LSB

AD_BUFFEROVF:                                   ;
        BISW    #UCB$W_CSR_M_BFO,-              ;SET BUFFER OVERRUN BIT
                UCB$W_AD_CSR(R5)                ; IN CSR COPY

;
; CSR ERROR BIT WAS SET:
;

AD_CSRERROR:                                    ;
        TSTW    AD_DBR(R4)                      ;CLEAR ERROR
        BRB     AD_DONE                         ;JOIN COMMON I/O COMPLETION

;
; DEVICE TIMED OUT DUE TO EITHER A REAL TIMEOUT OR TO A
; POWER FAILURE.  BOTH CAUSES ARE HANDLED THE SAME.
;

AD_TIMEOUT:                                     ;
        CLRW    AD_CSR(R4)                      ;CLEAR INTERRUPT ENABLE,
        TSTW    AD_DBR(R4)                      ; PENDING CONVERSION, INT, OR ERROR
        SETIPL  UCB$B_FIPL(R5)                  ;LOWER PRIORITY TO DEVICE LEVEL
        BRB     10$                             ;JOIN COMMON CODE TO
                                                ; TERMINATE REQUEST

;
; POWER FAILURE DETECTED WHILE ATTEMPTING TO INITIATE A READ OR
; LOOPBACK REQUEST.  TERMINATE REQUEST THE SAME AS IF IT OCCURRED
; DURING THE QIO.
;

AD_POWERFAIL:                                   ;
        ENBINT                                  ;LOWER IPL BACK TO FORK IPL
10$:    MOVZWL  #SS$_TIMEOUT,R0                 ;SET STATUS TO TIMED OUT
        BRB     20$                             ;JOIN COMMON CODE TO TERMINATE
                                                ; REQUEST
```

```
;
; NORMAL STATUS, CANCEL I/O, AND GENERAL I/O REQUEST COMPLETION:
;

AD_DONE:                                        ;
        CLRW    AD_CSR(R4)                      ;CLEAR INTERRUPT ENABLE
        IOFORK                                  ;REQUEST RESUMPTION AS FORK PROCESS
        MOVZWL  #SS$_DATAOVERUN,R0              ;ASSUME CSR ERROR
        BBS     #AD_CSR_V_ERR,-                 ;BRANCH IF SO
                UCB$W_AD_CSR(R5),20$            ;
        MOVZWL  #SS$_BUFFEROVF,R0               ;ASSUME BUFFER OVERRUN
        BBS     #UCB$W_CSR_V_BFO,-              ;BRANCH IF SO
                UCB$W_AD_CSR(R5),20$            ;
        MOVZWL  #SS$_NORMAL,R0                  ;ELSE, STATUS IS NORMAL
                                                ;
20$:    SUBW3   UCB$W_BCNT(R5),-                ;GET # BYTES REQUESTED
                IRP$W_BCNT(R3),R1               ; -# BYTES NOT XFERRED
        INSV    R1,#16,#16,R0                   ; =# BYTES XFERRED
        CLRL    R1                              ;CLEAR SECOND I/O STATUS LONGWD
        REQCOM                                  ;REQUEST I/O COMPLETION

        .DSABL  LSB
```

```
        .SBTTL  AD_INTERRUPT:   AD11-K A/D CONVERTER INTERRUPT SERVICE
;+
; AD_INTERRUPT - A/D CONVERTER INTERRUPT SERVICE
;
; THIS ROUTINE IS ENTERED VIA A JSB INSTRUCTION WHEN AN
; INTERRUPT OCCURS ON AN AD11 A/D CONVERTER.  INTERRUPT SERVICE
; GETS THE ADDRESS OF THE UCB OF THE INTERRUPTING DEVICE, RESTORES
; THE REMAINING CONTEXT OF THE DRIVER FORK PROCESS WHICH INITIATED
; THE DEVICE ACTIVITY, AND CALLS THE DRIVER FORK PROCESS.
;
; INPUTS:
;
;       ALL GENERAL REGISTERS = RANDOM
;       SP/ INTERRUPT STACK
;       0(SP) = ADDR OF IDB ADDR
;       4(SP) = SAVED R0
;       8(SP) = SAVED R1
;       12(SP) = SAVED R2
;       16(SP) = SAVED R3
;       20(SP) = SAVED R4
;       24(SP) = SAVED R5
;       28(SP) = SAVED PC
;       32(SP) = SAVED PSL
;       IPL/ HARDWARE DEVICE LEVEL
;
; OUTPUTS AT CALL TO DRIVER FORK:
;
;       R3 = RESTORED FROM DRIVER FORK PROCESS (IRP ADDR)
;       R4 = RESTORED FROM DRIVER FORK PROCESS (CSR ADDR)
;       R5 = UCB ADDR
;       STACK IS SAME AS ABOVE, BUT IDB POINTER POPPED
;       IPL/ HARDWARE DEVICE LEVEL
;-

        .ENABL  LSB

AD_INTERRUPT:                           ;A/D CONVERTER INTERRUPT SERVICE
        MOVL    @(SP)+,R3               ;GET IDB ADDR
        MOVQ    IDB$L_CSR(R3),R4        ;GET DEVICE CSR AND UCB ADDR
        BBCC    #UCB$V_INT,-            ;BRANCH IF INT UNEXPECTED,
                UCB$W_STS(R5),AD_UNSOL  ; AND CLEAR EXPECTED BIT
        MOVL    UCB$L_FR3(R5),R3        ;RESTORE REMAINING DRIVER
                                        ; CONTEXT: R3; (R4 ALREADY SET)
        JSB     @UCB$L_FPC(R5)          ;CALL DRIVER FORK PROCESS
                                        ;
10$:    MOVQ    (SP)+,R0                ;RESTORE REGISTERS
        MOVQ    (SP)+,R2                ;
        MOVQ    (SP)+,R4                ;
        REI                             ;
AD_UNSOL:                               ;HANDLE UNSOLICITED INTERRUPT
        CLRW    AD_CSR(R4)              ;DISMISS SPURIOUS INTERRUPT
        TSTW    AD_DBR(R4)              ;READ DATA BUFFER TO CLEAR ERROR
        BRB     10$                     ;JOIN INTERRUPT RESTORE

        .DSABL  LSB
```

```
        .SBTTL  AD_CTLINIT:     AD11-K CONTROLLER INITIALIZATION
;+
; AD_CTLINIT - AD11-K CONTROLLER INITIALIZATION
;
; THIS ROUTINE IS CALLED AT SYSTEM STARTUP AND AFTER A POWER
; FAILURE.
;
; THE CSR IS CLEARED TO DISABLE INTERRUPTS.  THIS WILL FORCE THE
; LAST SAMPLE (IF ONE IS IN PROGRESS) TO TIME OUT IN CASE INITIALIZATION
; IS THE RESULT OF A POWER FAILURE.  THE TIMEOUT WILL OCCUR IN 0-1
; SECONDS.
;
; THE DATA BUFFER REGISTER IS READ TO CLEAR A PENDING CONVERSION,
; INTERRUPT, OR ERROR FOR DEVICE INITIALIZATION.
;
; INPUTS:
;
;       R4 = AD11 CSR ADDRESS
;       R5 = IDB ADDRESS OF DEVICE UNIT
;       R6 = ADDR OF DDB
;       R8 = ADDR OF CRB
;
; OUTPUTS:
;
;       ALL REGISTERS PRESERVED
;-

AD_CTLINIT:                             ;
        CLRW    AD_CSR(R4)              ;CLEAR CSR (IE IN PARTICULAR)
        TSTW    AD_DBR(R4)              ;CLEAR ANY PENDING CONVERSION,
                                        ; INTERRUPT, OR ERROR
        RSB                             ;
```

```
        .SBTTL  AD_UNITINIT:    AD11-K UNIT INITIALIZATION
;+
; AD_UNITINIT - AD11-K UNIT INITIALIZATION
;
; THIS ROUTINE IS CALLED AT SYSTEM STARTUP AND AFTER A POWER
; FAILURE.  THE UCB AND IDB ARE INITIALIZED.
;
; INPUTS:
;
;       R5 = ADDRESS OF DEVICE UCB
;
; OUTPUTS:
;
;       R0 = IDB ADDRESS
;       OTHER REGISTERS ARE PRESERVED
;       UCB$W_STS(R5), ONLINE BIT IS SET
;       IDB$L_OWNER(R0) = ADDRESS OF OWNING UCB
;-

AD_UNITINIT:                                    ;
        BISW    #UCB$M_ONLINE,-                 ;SET UNIT ONLINE
                UCB$W_STS(R5)                   ;
        MOVL    UCB$L_CRB(R5),R0                ;GET CRB ADDRESS
        MOVL    CRB$L_INTD+VEC$L_IDB(R0),R0 ;GET IDB ADDR
        MOVL    R5,IDB$L_OWNER(R0)              ;SET UCB ADDR OF OWNING UNIT
        RSB                                     ;
                                                ;
AD_END:                                         ;END OF DRIVER LABEL


        .END
```

APPENDIX E

SAMPLE DRIVER FOR DR11-W DEVICES

This appendix contains the source listing of a driver for two connected DR11-W devices.

A machine-readable copy is also available. Its file specification is:

SYS$EXAMPLES:XADRIVER.MAR

```
        .TITLE  XADRIVER - VAX/VMS DR11-W DRIVER
        .IDENT  'V03-002'

;
;*****************************************************************************
;*                                                                           *
;*  COPYRIGHT (c) 1978, 1980, 1982 BY                                        *
;*  DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.                   *
;*  ALL RIGHTS RESERVED.                                                     *
;*                                                                           *
;*  THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED    *
;*  ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE           *
;*  INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER      *
;*  COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY    *
;*  OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY       *
;*  TRANSFERRED.                                                             *
;*                                                                           *
;*  THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE     *
;*  AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT         *
;*  CORPORATION.                                                             *
;*                                                                           *
;*  DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS      *
;*  SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.                  *
;*                                                                           *
;*                                                                           *
;*****************************************************************************
;
;++
;
; FACILITY:
;
;       VAX/VMS Executive, I/O Drivers
;
; ABSTRACT:
;
;       This module contains the DR11-W driver:
;
;               Tables for loading and dispatching
;               Controller initialization routine
;               FDT routine
;               The start I/O routine
;               The interrupt service routine
;               Device specific Cancel I/O
;               Error logging register dump routine
;
; ENVIRONMENT:
;
;       Kernel Mode, Nonpaged
;
; AUTHOR:
;
;       J. R. Programmer 10-JAN-79
;
```

```
;
; MODIFIED BY:
;
;       V03-002 JRP0195         J. Programmer           17-MAR-1982
;               Correct definition of XA$K_FNCT3.
;
;       V03-001 SSP0070         S. Programmer           15-MAR-1982
;               Correct use of DSBINT and SETIPL in WORD MODE READ section so
;               that IPL saved on stack when WFIKPCH is executed is fork IPL
;               not device IPL.  If there were several IRPs queued to the
;               device, the starting of a waiting IRP would cause this code
;               segment to be entered under the supervision of the fork
;               dispatcher.  The IPL upon return to the dispatcher will be
;               that saved on the stack before the WFIKPCH.  If that IPL is
;               not fork IPL, the fork dispatcher will bugcheck.  Lower IPL to
;               fork IPL in cancel I/O routine before purging UBA data path.
;               The UBA routines, particularly the release UBA resources
;               routines, must be called at fork IPL to synchronize access to
;               the UBA I/O data base and to ensure that any I/O threads
;               resumed as a result of the newly available resources are
;               resumed at fork IPL (presumably the IPL in effect when they
;               were suspended).  Discontinued clearing of the UCB$V_BSY bit
;               before executing REQCOM in the cancel I/O routine since
;               clearing the bit can only serve to confuse REQCOM.
;
;       V02-006 JRP0034         J. R. Programmer                27-July-1981
;               Fixed bug loading CSR values in START IO.
;
;
;
;--
```

```
        .SBTTL  External and local symbol definitions

; External symbols

        $ACBDEF                         ; AST control block
        $CRBDEF                         ; Channel request block
        $DDBDEF                         ; Device data block
        $DPTDEF                         ; Driver prolog table
        $EMBDEF                         ; EMB offsets
        $IDBDEF                         ; interrupt dispatch block
        $IODEF                          ; I/O function codes
        $IPLDEF                         ; Hardware IPL definitions
        $IRPDEF                         ; I/O request packet
        $PRDEF                          ; Internal processor registers
        $PRIDEF                         ; Scheduler priority increments
        $UCBDEF                         ; Unit control block
        $VECDEF                         ; Interrupt vector block
        $XADEF                          ; Define device specific characteristics

; Local symbols

; Argument list (AP) offsets for device-dependent QIO parameters

P1      = 0                             ; First QIO parameter
P2      = 4                             ; Second QIO parameter
P3      = 8                             ; Third QIO parameter
P4      = 12                            ; Fourth QIO parameter
P5      = 16                            ; Fifth QIO parameter
P6      = 20                            ; Sixth QIO parameter

; Other constants

XA_DEF_TIMEOUT  = 10                    ; 10 second default device timeout
XA_DEF_BUFSIZ   = 65535                 ; Default buffer size
XA_RESET_DELAY  = 2                     ; Delay N microseconds after RESET

; DR11-W definitions that follow the standard UCB fields
; *** N O T E ***  ORDER OF THESE UCB FIELDS IS ASSUMED

        $DEFINI UCB
        .=UCB$L_DPC+4
$DEF    UCB$L_XA_ATTN                   ; Attention AST listhead
                .BLKL   1
$DEF    UCB$W_XA_CSRTMP                 ; Temporary storage of CSR image
                .BLKW   1
$DEF    UCB$W_XA_BARTMP                 ; Temporary storage of BAR image
                .BLKW   1
$DEF    UCB$W_XA_CSR                    ; Saved CSR on interrupt
                .BLKW   1
$DEF    UCB$W_XA_EIR                    ; Saved EIR on interrupt
                .BLKW   1
$DEF    UCB$W_XA_IDR                    ; Saved IDR on interrupt
                .BLKW   1
$DEF    UCB$W_XA_BAR                    ; Saved BAR register on interrupt
                .BLKW   1
$DEF    UCB$W_XA_WCR                    ; Saved WCR register on interrupt
                .BLKW   1
$DEF    UCB$W_XA_ERROR                  ; Saved device status flag
                .BLKW   1
```

```
$DEF    UCB$L_XA_DPR                    ; Data Path Register contents
                .BLKL   1
$DEF    UCB$L_XA_FMPR                   ; Final Map Register contents
                .BLKL   1
$DEF    UCB$L_XA_PMPR                   ; Previous Map Register contents
                .BLKL   1
$DEF    UCB$W_XA_DPRN                   ; Saved Datapath Register Number
                .BLKW   1               ; And Datapath Parity error flag

; Bit positions for device-dependent status field in UCB

        $VIELD  UCB,0,<-                ; UCB device specific bit definitions
                <ATTNAST,,M>,-          ; ATTN AST requested
                <UNEXPT,,M>,-           ; Unexpected interrupt received
                >
UCB$K_SIZE=.
        $DEFEND UCB

; Device register offsets from CSR address

        $DEFINI XA                      ; Start of DR11-W definitions
$DEF    XA_WCR                          ; Word count
                        .BLKW   1
$DEF    XA_BAR                          ; Buffer address
                        .BLKW   1
$DEF    XA_CSR                          ; Control/status


; Bit positions for device control/status register

        $EQULST XA$K_,,0,1,<-           ; Define CSR FNCT bit values
                <FNCT1,2>-
                <FNCT2,4>-
                <FNCT3,8>-
                <STATUSA,2048>-         ; Define CSR STATUS bit values
                <STATUSB,1024>-
                <STATUSC,512>-
        >

        $VIELD  XA_CSR,0,<-             ; Control/status register
                <GO,,M>,-               ; Start device
                <FNCT,3,M>,-            ; CSR FNCT bits
                <XBA,2,M>,-             ; Extended address bits
                <IE,,M>,-               ; Enable interrupts
                <RDY,,M>,-              ; Device ready for command
                <CYCLE,,M>,-            ; Starts slave transmit
                <STATUS,3,M>,-          ; CSR STATUS bits
                <MAINT,,M>,-            ; Maintenance bit
                <ATTN,,M>,-             ; Status from other processor
                <NEX,,M>,-              ; Nonexistent memory flag
                <ERROR,,M>,-            ; Error or external interrupt
        >
```

```
$DEF    XA_EIR                          ; Error information register

; Bit positions for error information register

        $VIELD  XA_EIR,0,<-             ; Error information register
                <REGFLG,,M>,-           ; Flags whether EIR or CSR is accessed
                <SPARE,7,M>,-           ; Unused - spare
                <BURST,,M>,-            ; Burst mode transfer occured
                <DLT,,M>,-              ; timeout for successive burst xfer
                <PAR,,M>,-              ; Parity error during DATI/P
                <ACLO,,M>,-             ; Power fail on this processor
                <MULTI,,M>,-            ; Multi-cycle request error
                <ATTN,,M>,-             ; ATTN - same as in CSR
                <NEX,,M>,-              ; NEX - same as in CSR
                <ERROR,,M>,-            ; ERROR - same as in CSR
        >
                .BLKW   1

$DEF    XA_IDR                          ; Input Data Buffer register
$DEF    XA_ODR                          ; Output Data Buffer register
                .BLKW   1

        $DEFEND XA                      ; End of DR11-W definitions
```

```
        .SBTTL  Device Driver Tables

; Driver prologue table

        DPTAB   -                                       ; DPT-creation macro
                END=XA_END,-                            ; End of driver label
                ADAPTER=UBA,-                           ; Adapter type
                FLAGS=DPT$M_SVP,-                       ; Allocate system page table
                UCBSIZE=UCB$K_SIZE,-                    ; UCB size
                NAME=XADRIVER                           ; Driver name
        DPT_STORE INIT                                  ; Start of load
                                                        ; initialization table
        DPT_STORE UCB,UCB$B_FIPL,B,8                    ; Device fork IPL
        DPT_STORE UCB,UCB$B_DIPL,B,22                   ; Device interrupt IPL
        DPT_STORE UCB,UCB$L_DEVCHAR,L,<-                ; Device characteristics
                DEV$M_RTM!-                             ; Real Time device
                DEV$M_ELG!-                             ; Error Logging enabled
                DEV$M_IDV!-                             ;   input device
                DEV$M_ODV>                              ;   output device
        DPT_STORE UCB,UCB$B_DEVCLASS,B,DC$_REALTIME     ; Device class
        DPT_STORE UCB,UCB$B_DEVTYPE,B,DT$_DR11W ; Device Type
        DPT_STORE UCB,UCB$W_DEVBUFSIZ,W,-               ; Default buffer size
                XA_DEF_BUFSIZ
        DPT_STORE REINIT                                ; Start of reload
                                                        ; initialization table
        DPT_STORE DDB,DDB$L_DDT,D,XA$DDT                ; Address of DDT
        DPT_STORE CRB,CRB$L_INTD+4,D,-                  ; Address of interrupt
                XA_INTERRUPT                            ; service routine
        DPT_STORE CRB,CRB$L_INTD+VEC$L_INITIAL,-; Address of controller
                D,XA_CONTROL_INIT                       ; initialization routine
        DPT_STORE END                                   ; End of initialization
                                                        ; tables

; Driver dispatch table

        DDTAB   -                                       ; DDT-creation macro
                DEVNAM=XA,-                             ; Name of device
                START=XA_START,-                        ; Start I/O routine
                FUNCTB=XA_FUNCTABLE,-                   ; FDT address
                CANCEL=XA_CANCEL,-                      ; Cancel I/O routine
                REGDMP=XA_REGDUMP,-                     ; Register dump routine
                DIAGBF=<<13*4>+<<3+5+1>*4>>,-           ; Diagnostic buffer size
                ERLGBF=<<13*4>+<1*4>+<EMB$L_DV_REGSAV>> ; Error log buffer size

;
; Function decision table
;
XA_FUNCTABLE:                                           ; FDT for driver
        FUNCTAB ,-                                      ; Valid I/O functions
                <READPBLK,READLBLK,READVBLK,WRITEPBLK,WRITELBLK,WRITEVBLK,-
                SETMODE,SETCHAR,SENSEMODE,SENSECHAR>
        FUNCTAB ,                                       ; No buffered functions
        FUNCTAB XA_READ_WRITE,-                         ; Device-specific FDT
                <READPBLK,READLBLK,READVBLK,WRITEPBLK,WRITELBLK,WRITEVBLK>
        FUNCTAB +EXE$READ,<READPBLK,READLBLK,READVBLK>
        FUNCTAB +EXE$WRITE,<WRITEPBLK,WRITELBLK,WRITEVBLK>
        FUNCTAB XA_SETMODE,<SETMODE,SETCHAR>
        FUNCTAB +EXE$SENSEMODE,<SENSEMODE,SENSECHAR>
```

```
        .SBTTL  XA_CONTROL_INIT, Controller initialization

;++
; XA_CONTROL_INIT, Called when driver is loaded, system is booted, or
; power failure recovery.
;
; Functional Description:
;
;       1) Allocates the direct data path permanently
;       2) Assigns the controller data channel permanently
;       3) Clears the Control and Status Register
;       4) If power recovery, requests device timeout
;
; Inputs:
;
;       R4 = address of CSR
;       R5 = address of IDB
;       R6 = address of DDB
;       R8 = address of CRB
;
; Outputs:
;
;       VEC$V_PATHLOCK bit set in CRB$L_INTD+VEC$B_DATAPATH
;       UCB address placed into IDB$L_OWNER
;
;
;--

XA_CONTROL_INIT:

        MOVL    IDB$L_UCBLST(R5),R0     ; Address of UCB
        MOVL    R0,IDB$L_OWNER(R5)      ; Make permanent controller owner
        BISW    #UCB$M_ONLINE,UCB$W_STS(R0)
                                        ; Set device status "on-line"

; If powerfail has occured and device was active, force device timeout.
; The user can set his own timeout interval for each request.  Time-
; out is forced so a very long timeout period will be short circuited.

        BBS     #UCB$V_POWER,UCB$W_STS(R0),10$
                                        ; Branch if powerfail
        BISB    #VEC$M_PATHLOCK,CRB$L_INTD+VEC$B_DATAPATH(R8)
                                        ; Permanently allocate direct datapath
10$:
        BSBW    XA_DEV_RESET            ; Reset DR11W
        RSB                             ; Done
```

```
        .SBTTL  XA_READ_WRITE, FDT for device data transfers

;++
; XA_READ_WRITE, FDT for READLBLK,READVBLK,READPBLK,WRITELBLK,WRITEVBLK,
;                        WRITEPBLK
;
; Functional description:
;
;       1) Rejects QUEUE I/O's with odd transfer count
;       2) Rejects QUEUE I/O's for BLOCK MODE request to UBA Direct Data
;          PATH on odd byte boundary
;       3) Stores request timeout count specified in P3 into IRP
;       4) Stores FNCT bits specified in P4 into IRP
;       5) Stores word to write into ODR from P5 into IRP
;       6) Checks block mode transfers for memory modify access
;
; Inputs:
;
;       R3 = Address of IRP
;       R4 = Address of PCB
;       R5 = Address of UCB
;       R6 = Address of CCB
;       R8 = Address of FDT routine
;       AP = Address of P1
;               P1 = Buffer Address
;               P2 = Buffer size in bytes
;               P3 = Request timeout period (conditional on IO$M_TIMED)
;               P4 = Value for CSR FNCT bits (conditional on IO$M_SETFNCT)
;               P5 = Value for ODR (conditional on IO$M_SETFNCT)
;               P6 = Address of Diagnostic Buffer
;
; Outputs:
;
;       R0 = Error status if odd transfer count
;       IRP$L_MEDIA = timeout count for this request
;       IRP$L_SEGVBN = FNCT bits for DR11-W CSR and ODR image
;
;--

XA_READ_WRITE:
        BLBC    P2(AP),10$              ; Branch if transfer count even
2$:     MOVZWL  #SS$_BADPARAM,R0        ; Set error status code
5$:     JMP     G^EXE$ABORTIO           ; Abort request
10$:    MOVZWL  IRP$W_FUNC(R3),R1       ; Fetch I/O Function code
        MOVL    P3(AP),IRP$L_MEDIA(R3)  ; Set request specific timeout count
        BBS     #IO$V_TIMED,R1,15$      ; Branch if timeout specified
        MOVL    #XA_DEF_TIMEOUT,IRP$L_MEDIA(R3)
                                        ; Else set default timeout value
15$:    BBC     #IO$V_DIAGNOSTIC,R1,20$ ; Branch if not maintenance request
        EXTZV   #IO$V_FCODE,#IO$S_FCODE,R1,R1 ; AND out all function modifiers
        CMPB    #IO$_READPBLK,R1        ; If maintenance function, must be
                                        ; physical I/O read or write
        BEQL    20$
        CMPB    #IO$_WRITEPBLK,R1
        BEQL    20$
        MOVZWL  #SS$_NOPRIV,R0          ; No privilege for operation
        BRB     5$                      ; Abort request
20$:    EXTZV   #0,#3,P4(AP),R0         ; Get value for FNCT bits
        ASHL    #XA_CSR$V_FNCT,R0,IRP$L_SEGVBN(R3) ; Shift into position for CSR
        MOVW    P5(AP),IRP$L_SEGVBN+2(R3) ; Store ODR value for later
```

```
; If this is a block mode transfer, check buffer for modify access
; whether or not the function is read or write.  The DR11-W does
; not decide whether to read or write, the user's device does.
; For word mode requests, return to read check or write check.
;
; If this is a BLOCK MODE request and the UBA Direct Data Path is
; in use, check the data buffer address for word alignment.  If buffer
; is not word aligned, reject the request.

        BBS     #IO$V_WORD,IRP$W_FUNC(R3),30$
                                        ; Branch if word mode transfer
        BBS     #XA$V_DATAPATH,UCB$L_DEVDEPEND(R5),25$
                                        ; Branch if Buffered Data Path in use
        BLBS    P1(AP),2$               ; DDP, branch on bad alignment
25$:    JMP     G^EXE$MODIFY            ; Checke buffer for modify access
30$:    RSB                             ; Return
```

```
        .SBTTL  XA_SETMODE, Set Mode, Set characteristics FDT

;++
; XA_SETMODE, FDT routine to process SET MODE and SET CHARACTERISTICS
;
; Functional description:
;
;       If IO$M_ATTNAST modifier is set, queue attention AST for device
;       If IO$M_DATAPATH modifier is set, queue packet.
;       Else, finish I/O.
;
; Inputs:
;
;       R3 = I/O packet address
;       R4 = PCB address
;       R5 = UCB address
;       R6 = CCB address
;       R7 = Function code
;       AP = QIO Paramater list address
;
; Outputs:
;
;       If IO$M_ATTNAST is specified, queue AST on UCB attention AST list.
;       If IO$M_DATAPATH is specified, queue packet to driver.
;       Else, use exec routine to update device characteristics
;
;--

XA_SETMODE:
        MOVZWL  IRP$W_FUNC(R3),R0       ; Get entire function code
        BBC     #IO$V_ATTNAST,R0,20$    ; Branch if not an ATTN AST

; Attention AST request

        PUSHR   #^M<R4,R7>
        MOVAB   UCB$L_XA_ATTN(R5),R7    ; Address of ATTN AST control block list
        JSB     G^COM$SETATTNAST        ; Set up attention AST
        POPR    #^M<R4,R7>
        BLBC    R0,50$                  ; Branch if error
        BISW    #UCB$M_ATTNAST,UCB$W_DEVSTS(R5)
                                        ; Flag ATTN AST expected.
        BBC     #UCB$V_UNEXPT,UCB$W_DEVSTS(R5),10$
                                        ; Deliver AST if unsolicited interrupt
        BSBW    DEL_ATTNAST
10$:    JMP     G^EXE$FINISHIO          ; That's all for now

; If modifier IO$M_DATAPATH is set,
; queue packet.  The data path is changed at driver level to preserve
; order with other requests.

20$:    BBS     S^#IO$V_DATAPATH,R0,30$ ; If BDP modifier set, queue packet

        JMP     G^EXE$SETCHAR           ; Set device characteristics

; This is a request to change data path usage, queue packet

30$:    CMPL    #IO$_SETCHAR,R7         ; Set characteristics?
        BNEQ    45$                     ; No, must have the privelege
        JMP     G^EXE$SETMODE           ; Queue packet to start I/O

; Error, abort I/O

45$:    MOVZWL  #SS$_NOPRIV,R0          ; No priv for operation
50$:    CLRL    R1
        JMP     G^EXE$ABORTIO           ; Abort I/O on error
```

```
        .SBTTL  XA_START, Start I/O routines
;++
; XA_START - Start a data transfer, set characteristics, enable ATTN AST.
;
; Functional Description:
;
;       This routine has two major functions:
;
;       1) Start an I/O transfer.  This transfer can be in either word
;          or block mode.  The FNCTN bits in the DR11-W CSR are set.  If
;          the transfer count is zero, the STATUS bits in the DR11-W CSR
;          are read and the request completed.
;       2) Set Characteristics.  If the function is change data path, the
;          new data path flag is set in the UCB.
;
; Inputs:
;
;       R3 = Address of the I/O request packet
;       R5 = Address of the UCB
;
; Outputs:
;
;       R0 = final status and number of bytes transferred
;       R1 = value of CSR STATUS bits and value of input data buffer register
;       Device errors are logged
;       Diagnostic buffer is filled
;
;--
        .ENABL  LSB

XA_START:

; Retrieve the address of the device CSR

        ASSUME  IDB$L_CSR EQ 0
        MOVL    UCB$L_CRB(R5),R4        ; Address of CRB
        MOVL    @CRB$L_INTD+VEC$L_IDB(R4),R4
                                        ; Address of CSR

; Fetch the I/O function code

        MOVZWL  IRP$W_FUNC(R3),R1       ; Get entire function code
        MOVW    R1,UCB$W_FUNC(R5)       ; Save FUNC in UCB for Error Logging
        EXTZV   #IO$V_FCODE,#IO$S_FCODE,R1,R2 ; Extract function field

; Dispatch on function code.  If this is SET CHARACTERISTICS, we will
; select a data path for future use.
; If this is a transfer function, it will either be processed in word
; or block mode.

        CMPB    #IO$_SETCHAR,R2         ; Set characteristics?
        BNEQ    3$
```

```
;++
; SET CHARACTERISTICS - Process Set Characteristics QIO function
;
; INPUTS:
;
;       XA_DATAPATH bit in Device Characteristics specifies which data path
;       to use.  If bit is a one, use buffered data path.  If zero, use
;       direct datapath.
;
; OUTPUTS:
;
;       CRB is flagged as to which datapath to use.
;       DEVDEPEND bits in device characteristics is updated
;               XA_DATAPATH = 1 -> buffered data path in use
;               XA_DATAPATH = 0 -> direct data path in use
;--

        MOVL    UCB$L_CRB(R5),R0                ; Get CRB address
        MOVQ    IRP$L_MEDIA(R3),UCB$B_DEVCLASS(R5) ; Set device characteristics
        BISB    #VEC$M_PATHLOCK,CRB$L_INTD+VEC$B_DATAPATH(R0)
                                                ; Assume direct datapath
        BBC     #XA$V_DATAPATH,UCB$L_DEVDEPEND(R5),2$ ;  Were we right?
        BICB    #VEC$M_PATHLOCK,CRB$L_INTD+VEC$B_DATAPATH(R0) ; Set buffered datapath
2$:
        CLRL    R1                              ; Return Success
        MOVZWL  #SS$_NORMAL,R0
        REQCOM

; If subfunction modifier for device reset is set, do one here

3$:     BBC     S^#IO$V_RESET,R1,4$     ; Branch if not device reset
        BSBW    XA_DEV_RESET            ; Reset DR11-W

; This must be a data transfer function - i.e. READ OR WRITE
; Check to see if this is a zero length transfer.
; If so, only set CSR FNCT bits and return STATUS from CSR

4$:     TSTW    UCB$W_BCNT(R5)          ; Is transfer count zero?
        BNEQ    10$                     ; No, continue with data transfer
        BBC     S^#IO$V_SETFNCT,R1,6$   ; Set CSR FNCT specified?
        DSBINT
        MOVW    IRP$L_SEGVBN+2(R3),XA_ODR(R4)
                                        ; Store word in ODR
        MOVZWL  XA_CSR(R4),R0
        BICW    #<XA_CSR$M_FNCT!XA_CSR$M_ERROR>,R0
        BISW    IRP$L_SEGVBN(R3),R0
        MOVW    R0,XA_CSR(R4)
        BBC     #XA$V_LINK,UCB$L_DEVDEPEND(R5),5$ ; Link mode?
        BICW3   #XA$K_FNCT2,R0,XA_CSR(R4)       ; Make FNCT bit 2 a pulse
5$:
        ENBINT
6$:
        BSBW    XA_REGISTER             ; Fetch DR11-W registers
        BLBS    R0,7$                   ; If error, then log it
        JSB     G^ERL$DEVICERR          ; Log a device error
7$:     JSB     G^IOC$DIAGBUFILL        ; Fill diagnostic buffer if specified
        MOVL    UCB$W_XA_CSR(R5),R1     ; Return CSR and EIR in R1
        MOVZWL  UCB$W_XA_ERROR(R5),R0   ; Return status in R0
        BISB    #XA_CSR$M_IE,XA_CSR(R4) ; Enable device interrupts
        REQCOM                          ; Request done
```

```
; Build CSR image in R0 for later use in starting transfers

10$:
        DIVW3   #2,UCB$W_BCNT(R5),UCB$L_XA_DPR(R5)
                                        ; Make byte count into word count
        MOVZWL  XA_CSR(R4),R0
        BICW    #^C<XA_CSR$M_FNCT>,R0
        BISW    #XA_CSR$M_IE,R0         ; Set Interrupt Enable
        BBC     S^#IO$V_SETFNCT,R1,20$  ; Set FNCT bits in CSR?
        BICW    #<XA_CSR$M_FNCT>,R0     ; Yes, Clear previous FNCT bits
        BISB    IRP$L_SEGVBN(R3),R0     ; OR in new value
20$:    BBC     S^#IO$V_DIAGNOSTIC,R1,23$       ; Check for maintenance function
        BISW    #XA_CSR$M_MAINT,R0      ; Set maintenance bit in CSR image

; Is this a word mode or block mode request?

23$:    MOVW    R0,UCB$W_XA_CSRTMP(R5)  ; Save CSR image in UCB
        BBC     S^#IO$V_WORD,R1,BLOCK_MODE ; Check if word or block mode
        BRW     WORD_MODE               ; Branch to handle word mode
```

```
;++
; BLOCK MODE -- Process a Block Mode (DMA) transfer request
;
; FUNCTIONAL DESCRIPTION:
;
;       This routine takes the buffer address, buffer size, fucntion code,
;       and function modifier fields from the IRP.  It calculates the UNIBUS
;       address, allocates the UBA map registers, loads the DR11-W device
;       registers and starts the request.
;--
; Set up UBA
; Start transfer

BLOCK_MODE:

; If IO$M_CYCLE subfunction is specified, set CYCLE bit in CSR image

        BBC     #IO$V_CYCLE,R1,25$      ; Set CYCLE bit in CSR?
        BISW    #XA_CSR$M_CYCLE,UCB$W_XA_CSRTMP(R5) ; If yes, or into CSR image

; Allocate UBA data path and map registers

25$:
        REQDPR                          ; Request UBA data path
        REQMPR                          ; Request UBA map registers
        LOADUBA                         ; Load UBA map registers

; Calculate the UNIBUS transfer address for the DR11-W from the UBA
; map register address and byte offset.

        MOVZWL  UCB$W_BOFF(R5),R1       ; Byte offset in first page of xfer
        MOVL    UCB$L_CRB(R5),R2        ; Address of CRB
        INSV    CRB$L_INTD+VEC$W_MAPREG(R2),#9,#9,R1
                                        ; Insert page number
        EXTZV   #16,#2,R1,R2            ; Extract bits 17:16 of bus address
        ASHL    #XA_CSR$V_XBA,R2,R2     ; Shift extended memeroy bits for CSR
        BISW    #XA_CSR$M_GO,R2         ; Set "GO" bit into CSR image
        BISW    R2,UCB$W_XA_CSRTMP(R5)  ; Set into CSR image we are building
        BICW3   #<XA_CSR$M_GO!XA_CSR$M_CYCLE>,UCB$W_XA_CSRTMP(R5),R0
                                        ; CSR image less "GO" and "CYCLE"
        BICW3   #XA$K_FNCT2,UCB$W_XA_CSRTMP(R5),R2 ; CSR image less FNCT bit 2
        MOVW    R1,UCB$W_XA_BARTMP(R5)  ; Save BAR for error logging

; At this juncture:
;       R0 = CSR image less "GO" and "CYCLE"
;       R1 = low 16 bits of transfer bus address
;       R2 = CSR image less FNCT bit 2
;       UCB$L_XA_DPR(R5) = transfer count in words
;       UCB$W_XA_CSRTMP(R5) = CSR image to start transfer with

; Set DR11-W registers and start transfer
; Note that read-modify-write cycles are NOT performed to the DR11-W CSR.
; The CSR is always written directly into.  This prevents inadvertently setting
; the EIR select flag (writing bit 15) if error happens to become true.

        DSBINT                          ; Disable interrupts (powerfail)
        MNEGW   UCB$L_XA_DPR(R5),XA_WCR(R4)
                                        ; Load negative of transfer count
        MOVW    R1,XA_BAR(R4)           ; Load low 16 bits of bus address
        MOVW    R0,XA_CSR(R4)           ; Load CSR image less "GO" and "CYCLE"
        BBC     #XA$V_LINK,UCB$L_DEVDEPEND(R5),26$ ; Link mode?
        MOVW    R2,XA_CSR(R4)           ; Yes, load CSR image less "FNCT" bit 2
        BRB     126$            ; Only if link mode in dev characteristics
```

```
26$:
        MOVW    UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ; Move all bits to CSR

; Wait for transfer complete interrupt, powerfail, or device timeout

126$:
        WFIKPCH XA_TIME_OUT,IRP$L_MEDIA(R3) ; Wait for interrupt

; Device has interrupted, FORK

        IOFORK                          ; FORK to lower IPL

; Handle request completion, release UBA resources, check for errors

        MOVZWL  #SS$_NORMAL,-(SP)       ; Assume success, store code on stack
        CLRW    UCB$W_XA_DPRN(R5)       ; Clear DPR number and DPR error flag
        PURDPR                          ; Purge UBA buffered data path
        BLBS    R0,27$                  ; Branch if no datapath error
        MOVZWL  #SS$_PARITY,(SP)        ; Flag parity error on device
        INCB    UCB$W_XA_DPRN+1(R5)     ; Flag PDR error for log
27$:    MOVL    R1,UCB$L_XA_DPR(R5)     ; Save data path register in UCB
        EXTZV   #VEC$V_DATAPATH,-       ; Get Datapath register no.
                #VEC$S_DATAPATH,-       ; For Error Log
                CRB$L_INTD+VEC$B_DATAPATH(R3),R0
        MOVB    R0,UCB$W_XA_DPRN(R5)    ; Save for later in UCB
        EXTZV   #9,#7,UCB$W_XA_BAR(R5),R0 ; Low bits, final map register no.
        EXTZV   #4,#2,UCB$W_XA_CSR(R5),R1 ; Hi bits of map register no.
        INSV    R1,#7,#2,R0             ; Entire map register number
        CMPW    R0,#496                 ; Is map register number in range?
        BGTR    28$                     ; No, forget it - compound error
        MOVL    (R2)[R0],UCB$L_XA_FMPR(R5) ; Save map register contents
        CLRL    UCB$L_XA_PMPR(R5)       ; Assume no previous map register
        DECL    R0                      ; Was there a previous map register?
        CMPV    #VEC$V_MAPREG,#VEC$S_MAPREG,-
                CRB$L_INTD+VEC$W_MAPREG(R3),R0
        BGTR    28$                     ; No if gtr
        MOVL    (R2)[R0],UCB$L_XA_FMPR(R5) ; Save previous map register contents
28$:    RELMPR                          ; Release UBA resources
        RELDPR

; Check for errors and return status

        TSTW    UCB$W_XA_WCR(R5)        ; All words transferred?
        BEQL    30$                     ; Yes
        MOVZWL  #SS$_OPINCOMPL,(SP)     ; No, flag operation not complete
30$:    BBC     #XA_CSR$V_ERROR,UCB$W_XA_CSR(R5),35$ ; Branch on CSR error bit
        MOVZWL  UCB$W_XA_ERROR(R5),(SP) ; Flag for controller/drive error status
        BSBW    XA_DEV_RESET            ; Reset DR11-W
35$:    BLBS    (SP),40$                ; Any errors after all this?
        JSB     G^ERL$DEVICERR          ; Yes, log them
40$:    BSBW    DEL_ATTNAST             ; Deliver outstanding ATTN AST's
        JSB     G^IOC$DIAGBUFILL        ; Fill diagnostic buffer
        MOVL    (SP)+,R0                ; Get final device status
        MULW3   #2,UCB$W_XA_WCR(R5),R1  ; Calculate final transfer count
        ADDW    UCB$W_BCNT(R5),R1
        INSV    R1,#16,#16,R0           ; Insert into high byte of IOSB
        MOVL    UCB$W_XA_CSR(R5),R1     ; Return CSR and EIR in IOSB
        BISB    #XA_CSR$M_IE,XA_CSR(R4) ; Enable interrupts
        REQCOM                          ; Finish request in exec
        .DSABL  LSB
```

```
;++
; WORD MODE -- Process word mode (interrupt per word) transfer
;
; FUNCTIONAL DESCRIPTION:
;
;       Data is transferred one word at a time with an interrupt for each word.
;       The request is handled separately for a write (from memory to DR11-W
;       and a read (from DR11-W to memory).
;       For a write, data is fetched from memory, loaded into the ODR of the
;       DR11-W and the system waits for an interrupt.  For a read, the system
;       waits for a DR11-W interrupt and the IDR is transferred into memory.
;       If the unsolicited interrupt flag is set, the first word is transferred
;       directly into memory withou waiting for an interrupt.
;--

        .ENABL  LSB
WORD_MODE:


; Dispatch to separate loops on READ or WRITE

        CMPB    #IO$_READPBLK,R2        ; Check for read function
        BEQL    30$

;++
; WORD MODE WRITE -- Write (output) in word mode
;
; FUNCTIONAL DESCRIPTION:
;
;       Transfer the requested number of words from user memory to
;       the DR11-W ODR one word at a time, wait for interrupt for each
;       word.
;--

10$:
        BSBW    MOVFRUSER               ; Get two bytes from user buffer
        DSBINT                          ; Lock out interrupts
                                        ; Flag interrupt expected
        MOVW    R1,XA_ODR(R4)           ; Move data to DR11-W
        MOVW    UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ; Set DR11-W CSR
        BBC     #XA$V_LINK,UCB$L_DEVDEPEND(R5),15$ ; Link mode?
        BICW3   #XA$K_FNCT2,UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ; Clear interrupt FNCT bit 2
                                        ; Only if link mode specified
15$:

; Wait for interrupt, powerfail, or device timeout

        WFIKPCH XA_TIME_OUTW,IRP$L_MEDIA(R3)

; Check for errors, decrement transfer count, and loop until complete

        IOFORK                          ; Fork to lower IPL
        BITW    #XA_EIR$M_NEX!-
                XA_EIR$M_MULTI!-
                XA_EIR$M_ACLO!-
                XA_EIR$M_PAR!-
                XA_EIR$M_DLT,UCB$W_XA_EIR(R5) ; Any errors?
        BEQL    20$                     ; No, continue
        BRW     40$                     ; Yes, abort transfer.
20$:    DECW    UCB$L_XA_DPR(R5)        ; All words transferred?
        BNEQ    10$                     ; No, loop until finished.

; Transfer is done, clear interrupt expected flag and FORK
; All words read or written in WORD MODE.  Finish I/O.
```

```
RETURN_STATUS:

        JSB     G^IOC$DIAGBUFILL        ; Fill diagnostic buffer if present
        BSBW    DEL_ATTNAST             ; Deliver outstanding ATTN AST's
        MOVZWL  #SS$_NORMAL,R0          ; Complete success status
22$:    MULW3   #2,UCB$L_XA_DPR(R5),R1  ; Calculate actual bytes xfered
        SUBW3   R1,UCB$W_BCNT(R5),R1    ; From requested number of bytes
        INSV    R1,#16,#16,R0           ; And place in high word of R0
        MOVL    UCB$W_XA_CSR(R5),R1     ; Return CSR and EIR status
        BISB    #XA_CSR$M_IE,XA_CSR(R4) ; Enable device interrupts
        REQCOM                          ; Finish request in exec

;++
; WORD MODE READ -- Read (input) in word mode
;
; FUNCTIONAL DESCRIPTION:
;
;       Transfer the requested number of wrods from the DR11-W IDR into
;       user memory one word at a time, wait for interrupt for each word.
;       If the unexpected (unsolicited) interrupt bit is set, transfer the
;       first (last received) word to memory without waiting for an
;       interrupt.
;--

30$:
        DSBINT  UCB$B_DIPL(R5)          ; Lock out interrupts

; If an unexpected (unsolicited) interrupt has occured, assume it
; is for this READ request and return value to user buffer without
; waiting for an interrupt.

        BBSC    #UCB$V_UNEXPT,UCB$W_DEVSTS(R5),37$
                                        ; Branch if unexpected interrupt
        SETIPL  #IPL$_POWER
35$:

; Wait for interrupt, powerfail, or device timeout

        WFIKPCH XA_TIME_OUTW,IRP$L_MEDIA(R3)

; Check for errors, decrement transfer count and loop until done

        IOFORK                          ; Fork to lower IPL
37$:
        BITW    #XA_EIR$M_NEX!-
                XA_EIR$M_MULTI!-
                XA_EIR$M_ACLO!-
                XA_EIR$M_PAR!-
                XA_EIR$M_DLT,UCB$W_XA_EIR(R5) ; Any errors?
        BNEQ    40$                     ; Yes, abort transfer.
        BSBW    MOVTOUSER               ; Store two bytes into user buffer

; Send interrupt back to sender.  Acknowledge we got last word.

        DSBINT
        MOVW    UCB$W_XA_CSRTMP(R5),XA_CSR(R4)
        BBC     #XA$V_LINK,UCB$L_DEVDEPEND(R5),38$ ; Link mode?
        BICW3   #XA$K_FNCT2,UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ; Yes, clear FNCT 2
38$:
        DECW    UCB$L_XA_DPR(R5)                ; Decrement transfer count
        BNEQ    35$                     ; Loop until all words transferred
        ENBINT
        BRB     RETURN_STATUS           ; Finish request in common code
```

```
; Error detected in word mode transfer

40$:
        BSBW    DEL_ATTNAST             ; Deliver ATTN AST's
        BSBW    XA_DEV_RESET            ; Error, reset DR11-W
        JSB     G^IOC$DIAGBUFILL        ; Fill diagnostic buffer if present
        JSB     G^ERL$DEVICERR          ; Log device error
        MOVZWL  UCB$W_XA_ERROR(R5),R0   ; Set controller/drive status in R0
        BRW     22$

        .DSABL  LSB
;
; MOVFRUSER - Routine to fetch two bytes from user buffer.
;
; INPUTS:
;
;       R5 = UCB address
;
; OUTPUTS:
;
;       R1 = Two bytes of data from user's buffer
;       Buffer descriptor in UCB is updated.
;
        .ENABL  LSB
MOVFRUSER:
        MOVAL   -(SP),R1                ; Address of temporary stack loc
        MOVZBL  #2,R2                   ; Fetch two bytes
        JSB     G^IOC$MOVFRUSER         ; Call exec routine to do the deed
        MOVL    (SP)+,R1                ; Retrieve the bytes
        BRB     20$                     ; Update UCB buffer pointers
;
; MOVTOUSER - Routine to store two bytes into user's buffer.
;
; INPUTS:
;
;       R5 = UCB address
;       UCB$W_XA_IDR(R5) = Location where two bytes are saved
;
; OUTPUTS:
;
;       Two bytes are stored in user buffer and buffer descriptor in
;       UCB is updated.
;
MOVTOUSER:
        MOVAB   UCB$W_XA_IDR(R5),R1     ; Address of internal buffer
        MOVZBL  #2,R2
        JSB     G^IOC$MOVTOUSER         ; Call exec
20$:                                    ; Update buffer pointers in UCB
        ADDW    #2,UCB$W_BOFF(R5)       ; Add two to buffer descriptor
        BICW    #^C<^X01FF>,UCB$W_BOFF(R5) ; Modulo the page size
        BNEQ    30$                     ; If NEQ, no page boundary crossed
        ADDL    #4,UCB$L_SVAPTE(R5)     ; Point to next page
30$:
        RSB
;
        .DSABL  LSB
        .PAGE
```

```
        .SBTTL  DR11-W DEVICE timeout
;++
; DR11-W device timeout
; If a DMA transfer was in progress, release UBA resources.
; For DMA or WORD mode, deliver ATTN AST's, log a device timeout error,
; and do a hard reset on the controller.
;
; Clear DR11-W CSR
; Return error status
;
; Power failure will appear as a device timeout
;--
        .ENABL  LSB
XA_TIME_OUT:                            ; timeout for DMA transfer

        SETIPL  UCB$B_FIPL(R5)          ; Lower to FORK IPL
        PURDPR                          ; Purge buffered data path in UBA
        RELMPR                          ; Release UBA map registers
        RELDPR                          ; Release UBA data path

XA_TIME_OUTW:                           ; timeout for WORD mode transfer

        MOVL    UCB$L_CRB(R5),R4        ; Fetch address of CSR
        MOVL    @CRB$L_INTD+VEC$L_IDB(R4),R4
        BSBW    XA_REGISTER             ; Read DR11-W registers
        JSB     G^IOC$DIAGBUFILL        ; Fill diagnostic buffer
        JSB     G^ERL$DEVICTMO          ; Log device timeout
        BSBW    DEL_ATTNAST             ; And deliver the AST's
        BSBW    XA_DEV_RESET            ; Reset controller
        MOVZWL  #SS$_TIMEOUT,R0         ; Error status
        CLRL    R1
        CLRW    UCB$W_DEVSTS(R5)        ; Clear ATTN AST flags
        BICW    #<UCB$M_TIM!UCB$M_INT!UCB$M_TIMOUT!UCB$M_CANCEL!UCB$M_POWER>,-
                UCB$W_STS(R5)           ; Clear unit status flags
        REQCOM                          ; Complete I/O in exec
        .DSABL  LSB
        .PAGE
```

```
        .SBTTL  XA_INTERRUPT, Interrupt service routine for DR11-W
;++
; XA_INTERRUPT, Handles interrupts generated by DR11-W
;
; Functional description:
;
;       This routine is entered whenever an interrupt is generated
;       by the DR11-W.  It checks that an interrupt was expected.
;       If not, it sets the unexpected (unsolicited) interrupt flag.
;       All device registers are read and stored into the UCB.
;       If an interrupt was expected, it calls the driver back at its Wait
;       For Interrupt point.
;       Deliver ATTN AST's if unexpected interrupt.
;
; Inputs:
;
;       00(SP) = Pointer to address of the device IDB
;       04(SP) = saved R0
;       08(SP) = saved R1
;       12(SP) = saved R2
;       16(SP) = saved R3
;       20(SP) = saved R4
;       24(SP) = saved R5
;       28(SP) = saved PSL
;       32(SP) = saved PC
;
; Outputs:
;
;       The driver is called at its Wait For Interrupt point if an
;       interrupt was expected.
;       The current value of the DR11-W CSR's are stored in the UCB.
;
;--
XA_INTERRUPT:                           ; Interrupt service for DR11-W
        MOVL    @(SP)+,R4               ; Address of IDB and pop SP
        MOVQ    (R4),R4                 ; CSR and UCB address from IDB

; Read the DR11-W device registers (WCR, BAR, CSR, EIR, IDR) and store
; into UCB.

        BSBW    XA_REGISTER             ; Read device registers

; Check to see if device transfer request active or not
; If so, call driver back at Wait for Interrupt point and
; Clear unexpected interrupt flag.

20$:    BBCC    #UCB$V_INT,UCB$W_STS(R5),25$
                                        ; If clear, no interrupt expected

; Interrupt expected, clear unexpected interrupt flag and call driver
; back.

        BICW    #UCB$M_UNEXPT,UCB$W_DEVSTS(R5)
                                        ; Clear unexpected interrupt flag
        MOVL    UCB$L_FR3(R5),R3        ; Restore drivers R3
        JSB     @UCB$L_FPC(R5)          ; Call driver back
        BRB     30$
```

```
; Deliver ATTN AST's if no interrupt expected and set unexpected
; interrupt flag.

25$:
        BISW    #UCB$M_UNEXPT,UCB$W_DEVSTS(R5) ; Set unexpected interrupt flag
        BSBW    DEL_ATTNAST             ; Deliver ATTN AST's
        BISB    #XA_CSR$M_IE,XA_CSR(R4) ; Enable device interrupts

; Restore registers and return from interrupt

30$:
        POPR    #^M<R0,R1,R2,R3,R4,R5>  ; Restore registers
        REI                             ; Return from interrupt
        .PAGE
```

```
        .SBTTL  XA_REGISTER - Handle DR11-W CSR transfers
;++
; XA_REGISTER - Routine to handle DR11-W register transfers
;
; INPUTS:
;
;       R4 - DR11-W CSR address
;       R5 - UCB address of unit
;
; OUTPUTS:
;
;       CSR, EIR, WCR, BAR, IDR, and status are read and stored into UCB.
;       The DR11-W is placed in its initial state with interrupts enabled.
;       R0 - .true. if no hard error
;            .false. if hard error (cannot clear ATTN)
;
; If the CSR ERROR bit is set and the associated condition can be cleared, then
; the error is transient and recoverable.  The status returned is SS$_DRVERR.
; If the CSR ERROR bit is set and cannot be cleared by clearing the CSR, then
; this is a hard error and cannot be recovered.  The returned status is
; SS$_CTRLERR.
;
;       R0,R1 - destroyed, all other registers preserved.
;--

XA_REGISTER:

        MOVZWL  #SS$_NORMAL,R0          ; Assume success
        MOVZWL  XA_CSR(R4),R1           ; Read CSR
        MOVW    R1,UCB$W_XA_CSR(R5)     ; Save CSR in UCB
        BBC     #XA_CSR$V_ERROR,R1,55$  ; Branch if no error
        MOVZWL  #SS$_DRVERR,R0          ; Assume "drive" error
55$:    BICW    #^C<XA_CSR$M_FNCT>,R1   ; Clear all uninteresting bits for later
        BISB    #<XA_CSR$M_ERROR/256>,XA_CSR+1(R4) ; Set EIR flag
        MOVW    XA_EIR(R4),UCB$W_XA_EIR(R5) ; Save EIR in UCB
        MOVW    R1,XA_CSR(R4)           ; Clear EIR flag and errors
        MOVW    XA_CSR(R4),R1           ; Read CSR back
        BBC     #XA_CSR$V_ATTN,R1,60$   ; If attention still set, hard error
        MOVZWL  #SS$_CTRLERR,R0         ; Flag hard controller error
60$:    MOVW    XA_IDR(R4),UCB$W_XA_IDR(R5) ; Save IDR in UCB
        MOVW    XA_BAR(R4),UCB$W_XA_BAR(R5)
        MOVW    XA_WCR(R4),UCB$W_XA_WCR(R5)
        MOVW    R0,UCB$W_XA_ERROR(R5)   ; Save status in UCB
        RSB
```

```
        .SBTTL  XA_CANCEL, Cancel I/O routine
;++
; XA_CANCEL, Cancels an I/O operation in progress
;
; Functional description:
;
;       Flushes Attention AST queue for the user.
;       If transfer in progress, do a device reset to DR11-W and finish the
;       request.
;       Clear interrupt expected flag.
;
; Inputs:
;
;       R2 = channel index number
;       R3 = address of current IRP
;       R4 = address of the PCB requesting the cancel
;       R5 = address of the device's UCB
;
; Outputs:
;
;--

XA_CANCEL:                                          ; Cancel I/O

        BBCC    #UCB$V_ATTNAST,UCB$W_DEVSTS(R5),20$
                                                    ; ATTN AST enabled?

; Finish all ATTN AST's for this process.

        PUSHR   #^M<R2,R6,R7>
        MOVL    R2,R6                               ; Set up channel number
        MOVAB   UCB$L_XA_ATTN(R5),R7                ; Address of listhead
        JSB     G^COM$FLUSHATTNS                    ; Flush ATTN AST's for process
        POPR    #^M<R2,R6,R7>

; Check to see if a data transfer request is in progress
; for this process on this channel

20$:
        SETIPL  UCB$B_DIPL(R5)                      ; Lock out device interrupts
        JSB     G^IOC$CANCELIO                      ; Check if transfer going
        BBC     #UCB$V_CANCEL,UCB$W_STS(R5),30$ ; Branch if not for this guy

; If BLOCK mode DMA request in progress, release UBA resources
; If transfer is in progress, do a device reset to DR11-W

        BBC     #UCB$V_INT,UCB$W_STS(R5),25$    ; Branch if transfer not in progress
        BBS     #IO$V_WORD,IRP$W_FUNC(R3),25$   ; Branch if not BLOCK mode xfer
        PUSHR   #^M<R2,R3,R4>                       ; Save some registers
        MOVL    UCB$L_CRB(R5),R4                    ; Get CRB address
        MOVL    @CRB$L_INTD+8(R4),R4                ; Get pointer to CSR in IDB
        BSBW    XA_DEV_RESET                        ; Reset DR11-W
        SETIPL  UCB$B_FIPL(R5)                      ; Lower IPL to release UBA
                                                    ; resources.
        PURDPR                                      ; Purge UBA buffered data path
        RELMPR                                      ; Release UBA map registers
        RELDPR                                      ; Release UBA data path register
        POPR    #^M<R2,R3,R4>
```

```
25$:
        MOVZWL  #SS$_CANCEL,R0                  ; Status is request canceled
        CLRL    R1
        CLRW    UCB$W_DEVSTS(R5)        ; Clear unexpected interrupt flag
        BICW    #<UCB$M_TIM!UCB$M_CANCEL!UCB$M_INT!UCB$M_TIMOUT>,-
                UCB$W_STS(R5)           ; Clear unit status flags
        REQCOM                                  ; Jump to exec to finish I/O

30$:
        SETIPL  UCB$B_FIPL(R5)                  ; Lower to FORK IPL
        RSB                                     ; Return

        .PAGE
```

```
        .SBTTL  DEL_ATTNAST, Deliver ATTN AST's
;++
; DEL_ATTNAST, Deliver all outstanding ATTN AST's
;
; Functional description:
;
;       This routine is used by the DR11-W driver to deliver all of the
;       outstanding attention AST's.  It is copied from COM$DELATTNAST in
;       the exec.  In addition, it places the saved value of the DR11-W CSR
;       and Input Data Buffer Register in the AST paramater.
;
; Inputs:
;
;       R5 = UCB of DR11-W unit
;
; Outputs:
;
;       R0,R1,R2 Destroyed
;       R3,R4,R5 Preserved
;--
DEL_ATTNAST:
        BBCC    #UCB$V_ATTNAST,UCB$W_DEVSTS(R5),30$
                                        ; Any ATTN AST's expected?
        PUSHR   #^M<R3,R4,R5>           ; Save R3,R4,R5
10$:    MOVL    8(SP),R1                ; Get address of UCB
        MOVAB   UCB$L_XA_ATTN(R1),R2    ; Address of ATTN AST listhead
        MOVL    (R2),R5                 ; Address of next entry on list
        BEQL    20$                     ; No next entry, end of loop
        BICW    #UCB$M_UNEXPT,UCB$W_DEVSTS(R1) ; Clear unexpected interrupt flag
        MOVL    (R5),(R2)               ; Close list
        MOVW    UCB$W_XA_IDR(R1),ACB$L_KAST+6(R5)
                                        ; Store IDR in AST paramater
        MOVW    UCB$W_XA_CSR(R1),ACB$L_KAST+4(R5)
                                        ; Store CSR in AST paramater
        PUSHAB  B^10$                   ; Set return address for FORK
        FORK                            ; FORK for this AST

; AST fork procedure

        MOVQ    ACB$L_KAST(R5),ACB$L_AST(R5)
                                        ; Re-arrange entries
        MOVB    ACB$L_KAST+8(R5),ACB$B_RMOD(R5)
        MOVL    ACB$L_KAST+12(R5),ACB$L_PID(R5)
        CLRL    ACB$L_KAST(R5)
        MOVZBL  #PRI$_IOCOM,R2          ; Set up priority increment
        JMP     G^SCH$QAST              ; Queue the AST

20$:    POPR    #^M<R3,R4,R5>           ; Restore registers
30$:    RSB                             ; Return

        .PAGE
```

```
        .SBTTL  XA_REGDUMP - DR11-W register dump routine
;++
; XA_REGDUMP - DR11-W Register dump routine.
;
; This routine is called to save the controller registers in a specified
; buffer.  It is called from the device error-logging routine and from the
; diagnostic buffer fill routine.
;
; Inputs:
;
;       R0 - Address of register save buffer
;       R4 - Address of Control and Status Register
;       R5 - Address of UCB
;
; Outputs:
;
;       The controller registers are saved in the specified buffer.
;
;               CSRTMP - The last command written to the DR11-W CSR by
;                        by the driver.
;               BARTMP - The last value written into the DR11-W BAR by
;                        the driver during a block mode transfer.
;               CSR - The CSR image at the last interrupt
;               EIR - The EIR image at the last interrupt
;               IDR - The IDR image at the last interrupt
;               BAR - The BAR image at the last interrupt
;               WCR - Word count register
;               ERROR - The system status at request completion
;               PDRN - UBA Datapath Register number
;               DPR - The contents of the UBA Data Path register
;               FMPR - The contents of the last UBA Map register
;               PMRP - The contents of the previous UBA Map register
;               DPRF - Flag for purge datapath error
;                       0 = no purger datapath error
;                       1 = parity error when datapath was purged
;
;       Note that the values stored are from the last completed transfer
;       operation. If a zero transfer count is specified, then the
;       values are from the last operation with a nonzero transfer count.
;--

XA_REGDUMP:

        MOVZBL  #11,(R0)+               ; Eleven registers are stored.
        MOVAB   UCB$W_XA_CSRTMP(R5),R1  ; Get address of saved register images
        MOVZBL  #8,R2                   ; Return 8 registers here
10$:    MOVZWL  (R1)+,(R0)+
        SOBGTR  R2,10$                  ; Move them all
        MOVZBL  UCB$W_XA_DPRN(R5),(R0)+ ; Save Datapath Register number
        MOVZBL  #3,R2                   ; And 3 more here
20$:    MOVL    (R1)+,(R0)+             ; Move UBA register contents
        SOBGTR  R2,20$
        MOVZBL  UCB$W_XA_DPRN+1(R5),(R0)+ ; Save Datapath Parity Error Flag
        RSB

        .PAGE
```

```
        .SBTTL  XA_DEV_RESET - Device reset DR11-W
;++
; XA_DEV_RESET - DR11-W Device reset routine
;
; This routine raises IPL to device IPL, performs a device reset to
; the required controler, and re-enables device interrupts.
;
; Inputs:
;
;       R4 - Address of Control and Status Register
;       R5 - Address of UCB
;
; Outputs:
;
;       Controller is reset, controller interrupts are enabled
;
;--

XA_DEV_RESET:

        PUSHR   #^M<R0,R1,R2>           ; Save some registers
        DSBINT                          ; Raise IPL to lock all interrupts
        MOVB    #<XA_CSR$M_MAINT/256>,XA_CSR+1(R4)
        CLRB    XA_CSR+1(R4)

; ***  Must delay here depending on reset interval

        MOVZBL  #XA_RESET_DELAY,R2      ; No. of microsecs to wait
5$:     MFPR    #PR$_ICR,R0             ; Read interval clock
10$:    MFPR    #PR$_ICR,R1             ; Read it again
        CMPL    R0,R1                   ; Compare both clock readings
        BEQL    10$                     ; Repeate until they differ
        SOBGTR  R2,5$                   ; Do this the specified no. of times

        MOVB    #XA_CSR$M_IE,XA_CSR(R4) ; Re-enable device interrupts
        ENBINT                          ; Restore IPL
        POPR    #^M<R0,R1,R2>           ; Restore registers

        RSB

XA_END:                                         ; End of driver label
        .END
```

APPENDIX F

MASSBUS ADAPTER

This appendix describes the data structures and macros used by DIGITAL for its standard magnetic tape and disk products. Customers using the DR32 should use equivalent techniques.

The MASSBUS adapter (MBA) is the hardware interface between the backplane interconnect and the high speed MASSBUS storage devices. The MASSBUS is the communication path linking the MASSBUS adapter to the mass storage device drives.

The MASSBUS adapter performs the following functions that allow communication between devices and memory:

- Mapping of virtual address to physical page frame numbers
- Buffering of data for transfers from main memory to the MASSBUS and vice versa
- Transfer of interrupts from MASSBUS devices to the backplane interconnect

A MASSBUS adapter supports any combination of up to eight device controllers. Typical MASSBUS controllers include the TM03 tape controller, the RP06, RM03, and RM80 disk controllers, and the DR32, which is a general purpose interface that acts as a controller for one or more nonstandard devices. Only one controller can transfer data over the MASSBUS at a time.

The TM03 tape controller supports up to eight tape drives. In contrast to tape controllers, there is a one-to-one relationship between a disk controller and its device; each controller supports only one disk drive. The VAX/VMS system interprets and maintains the I/O data base differently depending upon whether the controller is single or multiunit.

Each MASSBUS controller connected to a MASSBUS adapter is assigned a unit number in the range 0 to 7. The method of unit number assignment is controller-specific, but you can obtain the number from either unit plugs or switch packs. In the case of multiunit controllers, the unit number is distinct from the subunit numbers assigned to the individual drives connected to the controller.

Figure F-1 illustrates a possible MASSBUS configuration.

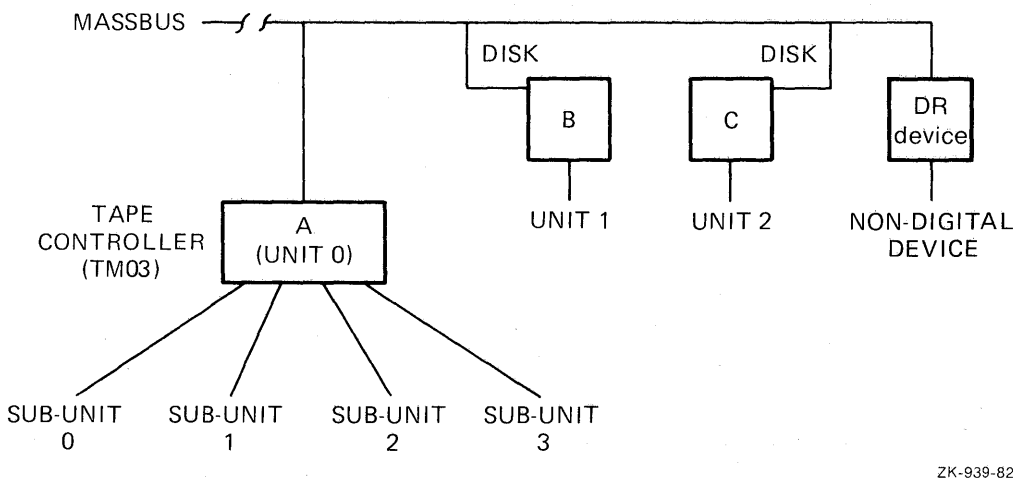


Figure F-1: MASSBUS Configuration

F.1 MASSBUS ADAPTER REGISTERS

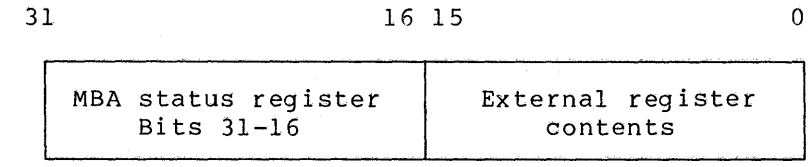
The MASSBUS adapter has three sets of registers:

- Internal registers for the MASSBUS adapter; that is, MBA registers
- External registers for each device (controller) on the MASSBUS; that is, device registers
- 256 map registers

To allow competing devices to share these resources, access to and modification of all MASSBUS adapter registers (internal, external, and map registers) are governed by certain rules and conventions. In particular, access to registers may, at times, require ownership of either the device controller or the MASSBUS adapter itself, or both. Subsequent sections in this appendix discuss the methods of obtaining such ownership of these shared resources.

MASSBUS adapter external registers are device-dependent and accessible whether or not the driver owns the MASSBUS adapter. However, in the case of multiunit MASSBUS adapter controllers, the driver may need to own the controller before it can gain access to a register.

MASSBUS adapter external registers are each 16 bits wide, but they must be accessed as longwords. When a driver reads an external register, the MASSBUS adapter concatenates the high order 16 bits of the MBA status register (one of the MBA internal registers) to the contents of the specified external register. A diagram of the resulting longword is shown below.



On a write to an external register, the MASSBUS adapter uses the low order 16 bits of the longword source operand to update the external register.

MASSBUS adapter internal and map registers are 32 bits in length. They must be accessed as longwords or the processor will signal a machine check exception. The driver for a MASSBUS device must obtain exclusive ownership of the MASSBUS adapter before modifying any of the MBA internal or map registers.

Bits 21 through 30 of each MBA map register are reserved; they cannot be written. Use of MBA map registers is analogous to use of UNIBUS adapter map registers with the following exceptions:

- Since the MASSBUS can handle only one transfer at a time, ownership of the MASSBUS adapter implies ownership of all its map registers. Thus, the driver need not independently request map registers.

- MBA map registers do not contain a byte offset field. The driver loads the full MASSBUS adapter virtual address, including the byte alignment, into the MASSBUS adapter virtual address register (VAR; one of the MBA internal registers) at the start of a data transfer. Use of the VAR register is described below.

- MBA map registers do not contain a data path field; the MASSBUS adapter has a single data path, and ownership of the adapter implies ownership of the path. Thus, the driver need not independently allocate the data path.

F.1.1 Loading MASSBUS Adapter Registers

To prepare for a data transfer over the MASSBUS, the driver that owns the MASSBUS adapter uses the LOADMBA macro to load the MBA map registers and associated MBA internal registers. The LOADMBA macro invokes the subroutine IOC$LOADMBAMAP, which performs the following steps:

- Determines the number of map registers needed to map the data area by adding the contents of UCB$W_BCNT to UCB$W_BOFF, adjusting the sum to the next even multiple of 512, and dividing the result by 512.

- Loads the specified number of map registers, beginning with MBA map register 0, with the contents of the page table entries pointed to by UCB$L_SVAPTE. This step maps the data area for the transfer into the low portion of MBA virtual address space. The routine also loads the next map register beyond the number used to map the data area with zeros (an invalid map entry). This procedure stops the transfer should a hardware failure occur.

- Loads the VAR register with the zero extended contents of UCB$W_BOFF. Since the first byte of the data area is located at offset UCB$W_BOFF within the page of memory mapped by MBA map register 0, then UCB$W_BOFF contains the virtual address of the start of the data area in MASSBUS adapter virtual address space.

- Loads the complement (negative) of UCB$W_BCNT into the MBA byte count register (BCR).

Note that if a driver wishes to perform a data transfer in the reverse direction (for example, read reverse on a tape) it must modify the contents of the VAR, as established by IOC$LOADMBAMAP, so that it points to the last byte of the data area. This is done by adding one less than the contents of UCB$W_BCNT to the contents of the VAR register.

During the progress of a data transfer over the MASSBUS, the VAR register is continuously updated so that it points to the current position in the data area. The VAX Hardware Handbook illustrates the mapping of the contents of the VAR register into physical memory.

F.1.2 MASSBUS Adapter Registers and Offsets

During system initialization, VAX/VMS builds an adapter control block (ADP) a channel request block (CRB), and an interrupt dispatch block (IDB) for each MASSBUS adapter. The system also allocates 4K bytes of system virtual address space for the adapter's register I/O space. The base of this I/O register virtual address space is placed in IDB$L_CSR. Thus, you can access MASSBUS adapter registers using the base register virtual address plus some offset. The $MBADEF macro defines the offsets for MASSBUS adapter registers. The major symbols defined by this macro are shown in Table F-1.

Table F-1: Major Offsets Defined by $MBADEF

Symbol	MBA Register Name	Hex Offset
MBA$L_CSR	Configuration Reg.	0
MBA$L_CR	Control Reg.	4
MBA$L_SR	Status Reg.	8
MBA$L_VAR	Virtual Address Reg.	C
MBA$L_BCR	Byte Count Reg.	10
MBA$L_DR	Diagnostic Reg.	14
MBA$L_SMR	Selected Map Reg.	18
MBA$L_CAR	Command Address Reg.	1C
MBA$L_ERB	External Register Base	400
MBA$L_AS	Attention Summary Reg.	414
MBA$L_MAP	Base of MAP Registers	800

The MASSBUS adapter internal registers occupy the low order 1024 bytes of addressable space even though there are only eight MBA internal registers. Beyond the internal registers, there are eight blocks of 32 longwords (128 bytes), one block for each of the eight device controllers that can be connected to a single MASSBUS adapter. Each of these blocks provides space for the device registers of each device controller. Beyond the device register space is tthe area reserved for the 256 MASSBUS adapter map registers.

Figure F-2 illustrates the relative positions of MASSBUS adapter registers and the values device drivers use to gain access to them. The base address of MASSBUS adapter address space, stored in IDB$L_CSR, is the address of the first MASSBUS adapter internal register. IDB$L_CSR represents the internal register's virtual location, while the MBA$L_ symbols represent register values as defined by $MBADEF. Note that the MASSBUS adapter register space occupies only the first 3K bytes out of the 8K bytes allotted to I/O physical addresses but that by convention, VAX/VMS allocates 4K bytes of virtual addresses to each MASSBUS adapter.

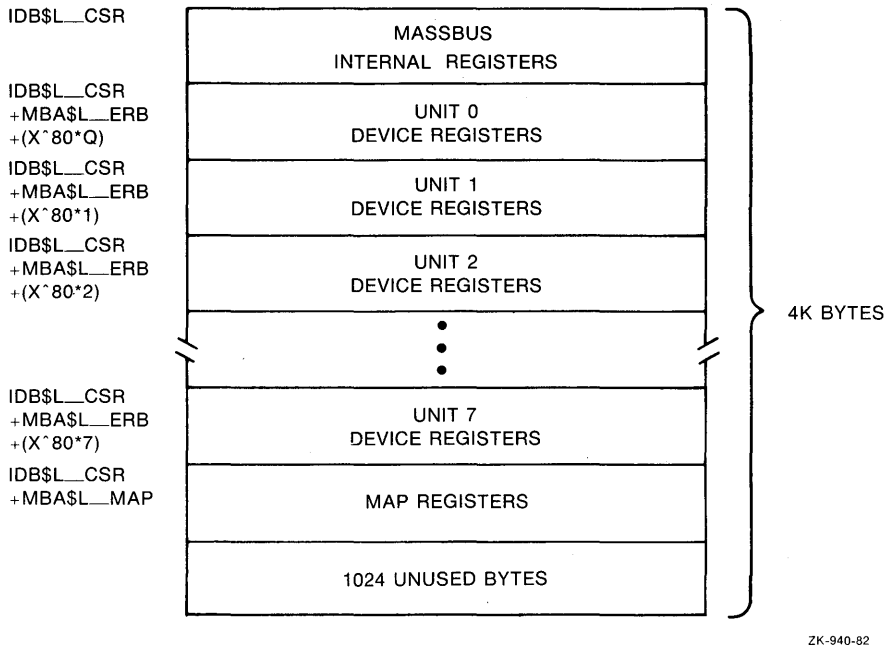


Figure F-2: Location of MASSBUS Registers in Physical Address Space

To address a map register in the MASSBUS adapter, the driver constructs the following address:

IDB$L_CSR + MAP$L_MAP + map register index

To address a device register, the driver constructs the following address:

IDB$L_CSR + MAP$L_ERB + (unit number * X^80) + register displacement

An individual driver should define offsets for the registers of its device. During execution, the driver computes a register address by summing the MBA starting virtual address (the contents of IDB$L_CSR), MBA$L_ERB, the unit number of the device controller multiplied by 80 (hex), and the offset of the specified register.

The Attention Summary register, as shown in Table F-1, appears to reside within the external register space reserved for MASSBUS adapter controller 0. Actually, the Attention Summary register is a composite register. Each MASSBUS adapter controller connected to the MASSBUS adapter contributes one bit of information to the register. This composite register appears in each of the eight device register spaces at offset 10 (hex) from the base of the device registers for that device. Thus, MBA$L_AS can be defined as either 410, 490, 510, 590 and so on. For convience, it has been defined as 410 (hex).

F.1.3 Modification of MASSBUS Adapter Registers

The driver for a MASSBUS device must obtain ownership of the MBA before modifying any of the MBA internal registers or the MBA map registers. A driver obtains ownership of the MBA by invoking either the REQPCHAN macro or the REQSCHAN macro, depending on whether the device is connected to a single unit MASSBUS controller or a multiunit

MASSBUS controller. For single unit controllers, invoke the REQPCHAN macro. Since the controller is dedicated to its single device, there is never any contention for the controller.

For multiunit devices, however, invoke the REQSCHAN macro to obtain MBA ownership because several devices may share the controller, and so must contend for its use. The multiunit controller relegates the MASSBUS adapter to a secondary position. Thus, for multiunit controllers, invoke REQPCHAN to gain ownership of the controller, and invoke REQSCHAN to obtain the MASSBUS adapter.

F.2 I/O DATA BASE FOR MASSBUS DEVICES

During initialization, the system creates an adapter control block, a channel request block, and an interrupt data block for each MASSBUS adapter included in the configuration. The driver loading procedure subsequently builds additional data structures for each device controller connected to a MASSBUS adapter. The type of structure created depends upon whether the device controller is a single or multiunit controller.

The system builds a unit control block for each single unit controller. Figure F-3 illustrates the I/O data base for a MASSBUS adapter with one single unit controller attached to it. Note that the ADP, CRB, and IDB all correspond to the MASSBUS adapter and can logically be considered a single extended data block. The UCB corresponds to the device/controller pair. Because of the one-to-one correspondence between a single unit controller and its device, the system does not need to distinguish between the two and thus does not maintain separate data blocks for each piece of hardware.

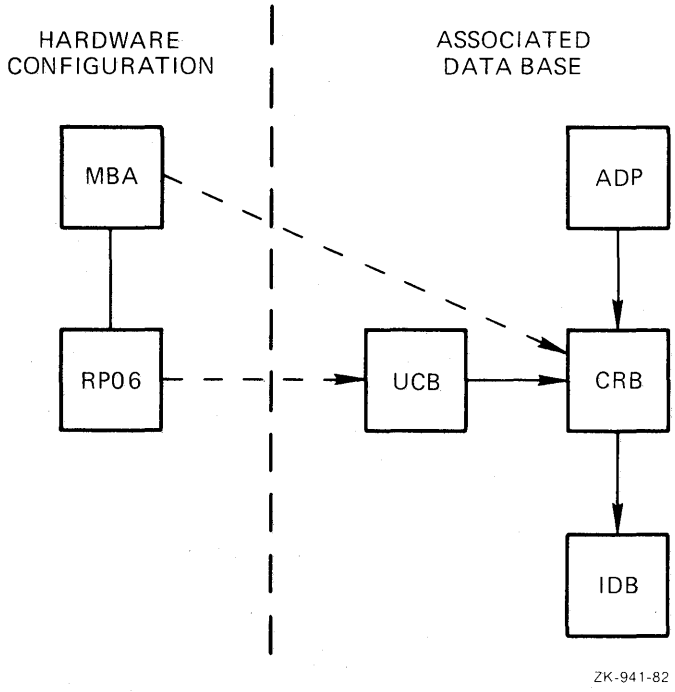


Figure F-3: I/O Data Base for MASSBUS Disk Unit

Multiunit controllers, however, require separate data structures for the controller and each of its subunits (devices). The driver loading procedure builds a CRB/IDB pair for the controller, as well as a UCB for each subunit. Figure F-4 shows the I/O data base created for a MASSBUS adapter with one disk unit and two tape units.

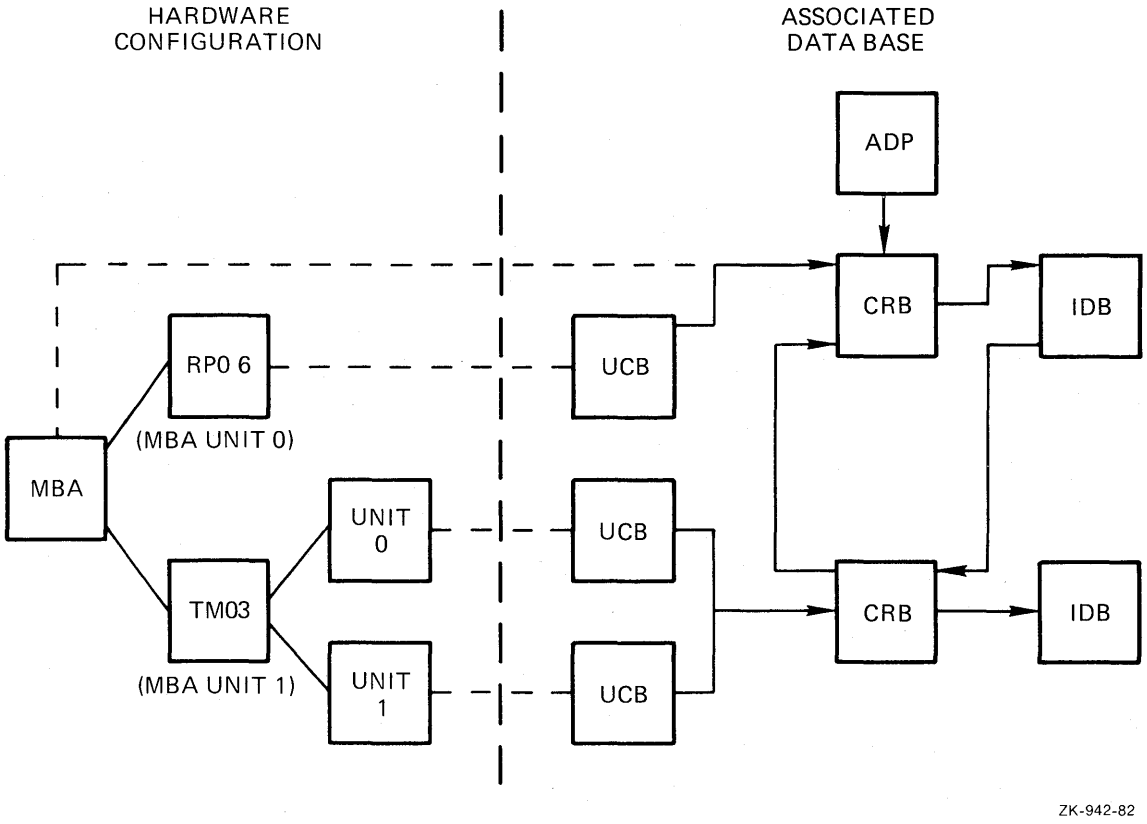


Figure F-4: I/O Data Base for MASSBUS Disk and Tape Units

Figure F-4 does not include several pointers used in interrupt dispatching. In particular, the IDB associated with the MASSBUS adapter maintains an array of up to eight longwords that point to the data structures associated with the eight possible MASSBUS controllers attached to the MASSBUS.

For single unit controllers, the IDB longword points to the device's UCB, whereas for multiunit controllers, the longword (or longwords) points to a field within the CRB associated with the multiunit controller. The low bit of this longword, when set, indicates a multiunit vector. The software checks this bit to determine whether the longword points to a single UCB or a multiunit CRB.

Also not pictured in Figure F-4 is the fact that multiunit interrupt data blocks also maintain an array of longwords. Each longword points to the individual unit control blocks of the subunits attached to the controller. Figure F-5 illustrates in more detail the set of I/O data structures for the MASSBUS adapter and its devices.

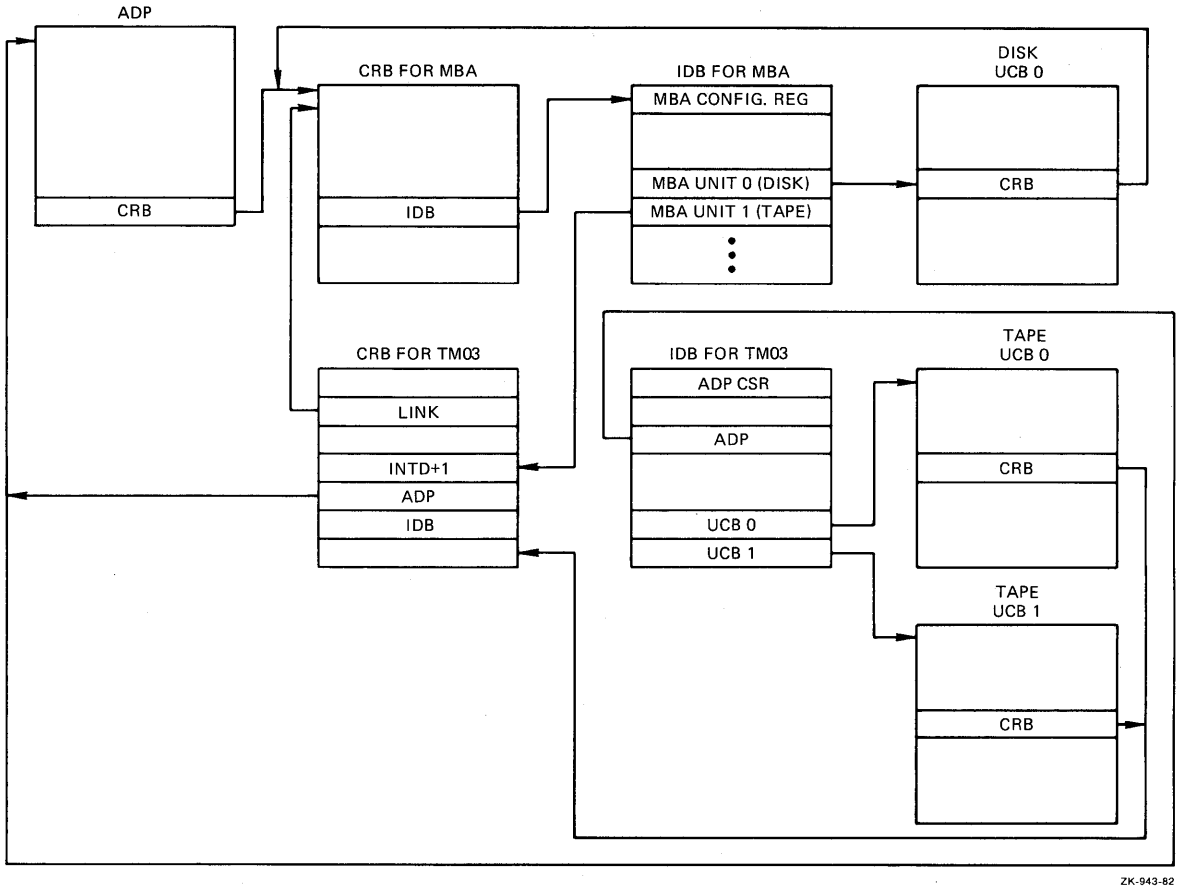


Figure F-5: I/O Data Structures Used in Dispatching an Interrupt

F.3 MASSBUS ADAPTER OPERATIONS

The MASSBUS accepts two kinds of operations: data transfer operations and nondata transfer operations. Data transfer operations require the use of MASSBUS adapter shared resources, while nondata transfers do not.

Before a driver can activate a data transfer operation on the MASSBUS, the driver must request and receive ownership of the MASSBUS adapter on behalf of the device unit. However, drivers must not initiate nondata transfer operations while they have control of the MASSBUS adapter. Section F.4.1 explains this statement further.

The MASSBUS adapter generates interrupts when data transfers terminate and when attention conditions arise on devices. When an interrupt occurs on the MASSBUS adapter, the MASSBUS adapter interrupt dispatcher determines whether the interrupt is a data transfer or an attention interrupt.

Data transfer interrupts occur when a data transfer either completes or aborts. When the interrupt occurs, the MBA Status Register (SR) contains information about the condition that caused the interrupt.

Attention interrupts occur when nondata transfers on MASSBUS devices terminate, or when the device undergoes an exceptional condition, such as becoming on-line.

The MASSBUS adapter's Attention Summary register controls attention interrupt handling. This register contains eight bits of data, one for each of the eight possible controllers that may be connected to the MASSBUS adapter. When a device incurs an attention condition, the hardware sets the corresponding bit in the Attention Summary register and generates a MASSBUS adapter interrupt.

If the attention condition occurs while a data transfer operation for another device is in progress, the hardware sets the bit in the summary register but suppresses the attention interrupt. The interrupt generated when the data transfer completes allows the MASSBUS adapter interrupt dispatcher to gain control, handle the data transfer interrupt, check the attention summary bit and then invoke the proper driver to handle the interrupt.

F.4 MASSBUS ADAPTER INTERRUPT DISPATCHING

When interrupts occur on the MASSBUS adapter, the MASSBUS adapter interrupt dispatcher gains control. This routine first determines whether the interrupt is the result of a data transfer or an attention condition. The routine checks to see if the MASSBUS adapter is owned, and if so, by whom.

F.4.1 Checking for MASSBUS Adapter Ownership

There are two conditions by which the interrupt dispatcher can determine that the interrupt is an attention interrupt:

- If the MASSBUS adapter is not owned
- If the MASSBUS adapter is owned, but the owner is not expecting an interrupt (UCB$M_INT in UCB$W_STS is clear)

When the MASSBUS adapter is owned and the owner expects an interrupt, the interrupt is assumed to be the result of a data transfer operation.

As mentioned earlier, a driver must not initiate nondata transfers on the MASSBUS adapter while it owns the adapter. For example, consider a MASSBUS adapter attached to two disk units, A and B. Disk A is performing an IO$_SEEK (a nondata transfer operation that completes fairly quickly), while at the same time, disk B is performing an IO$_RECAL operation (a nondata transfer operation that takes about .5 seconds to complete).

The driver for disk A correctly initiates its operation without obtaining possession of the MASSBUS adapter channel, but the disk B driver initiates its operation while it owns the MASSBUS adapter. Both of these operations, upon completion, set the bit in the Attention Summary register that corresponds to their respective drive units and interrupt. We will assume that disk A's IO$_SEEK completes first. The operation sets disk A's bit in the Attention Summary register and generates the MASSBUS adapter interrupt.

The MASSBUS adapter interrupt dispatcher finds that the adapter is owned, and that the owner is expecting an interrupt. Therefore, the interrupt dispatcher incorrectly assumes that it is handling a data transfer interrupt, and, moreover, that this interrupt is the one for which the owner of the MBA is waiting.

So, the MASSBUS adapter interrupt dispatcher returns control, through the fork block in the MASSBUS adapter owner's UCB, to the driver for

disk B, even though disk B's operation has not completed. The disk B driver will now incorrectly assume that the device has completed its operation, which can cause serious problems.

F.4.2 Dispatching the Interrupt

Once the MASSBUS adapter interrupt dispatcher determines the type of interrupt, it dispatches the interrupt to the driver. The interrupt dispatcher handles attention interrupts and data transfer interrupts in the same way, with one exception: on an attention interrupt, the interrupt dispatcher clears the MASSBUS adapter Status Register before dispatching the interrupt to the driver. The status register contains information used only in data transfer interrupt dispatching.

Interrupt dispatching to the driver differs depending on the type of controller.

The MASSBUS adapter interrupt dispatcher handles a solicited interrupt on a single unit controller by transferring control to the driver through the fork block in the UCB. On unsolicited interrupts on single-unit controllers, the interrupt dispatcher calls the driver unsolicited interrupt service routine.

On single unit controllers, the MASSBUS adapter interrupt dispatcher always clears the attention bit in the Attention Summary register before it calls back the driver after an interrupt.

Interrupt dispatching to the driver on multiunit controllers differs from single unit dispatching in two ways.

First, the interrupt dispatcher never clears the attention bit. This task is left to the driver because some multiunit controllers synchronize on this bit and guarantee the integrity of device registers only while the bit is set. If the interrupt dispatcher clears the bit before return to the driver, the driver can no longer rely on the contents of the device registers.

Second, a multiunit controller needs another interrupt dispatcher to handle simultaneous requests from its several subunits. This second level interrupt dispatcher resides in the driver for a multiunit controller. After an interrupt on a multiunit controller, the MASSBUS adapter interrupt dispatcher indirectly calls this driver interrupt dispatcher using code in the multiunit controller's channel request block. The driver loading procedure installs this code when it establishes the I/O data base.

F.5 SPECIAL MBA CONSIDERATIONS FOR DRIVERS

MASSBUS adapter considerations affect a driver's device unit initialization routine, start I/O routines and, for multiunit controllers only, the driver's DPTAB macro. MBA considerations also affect interrupt handling, as described in Section F.4.2. The next sections in this appendix discuss programming details for writing a MASSBUS device driver.

F.5.1 Considerations for Unit Initialization Routines

All drivers for MASSBUS adapter devices initialize two fields in the UCB (as well as initializing device-specific fields): UCB$B_SLAVE and UCB$B_SLAVE+1. The first of these fields should contain the

controller's MASSBUS adapter unit number. This unit number marks the controller's position on the MASSBUS adapter. The second of these contains the offset, in longwords, from the start of the MASSBUS adapter external registers of this controller's device registers. The value of this longword offset is always 32 times the MASSBUS adapter unit number of the controller.

Initialization of a device attached to a single unit controller is simple because the device unit number and the controller position number on the MASSBUS adapter are always equal. To initialize the field UCB$B_SLAVE, copy to it the contents of UCB$W_UNIT. To initialize UCB$B_SLAVE+1, multiply its contents by 32. The driver later uses this information to compute a pointer to this device's registers. By convention, R4 points to the MASSBUS adapter configuration register, and R5 points to the UCB of this device.

Thus, the following two instructions cause R3 to point to the device registers during normal system operation:

```
MOVZBL    UCB$B_SLAVE+1(R5),R3
MOVAL     MBA$L_ERB(R4)[R3],R3
```

For devices connected to a multiunit controller, determination of the controller's MBA position is more complex. When the unit initialization routine is invoked, the following values are in the following registers:

R3 -- Address of multiunit controller device registers
R4 -- Address of the MBA configuration register
R5 -- Address of device UCB

The driver computes the MBA position of the controller by using R3 and R4 to determine the number of bytes from the start of the MBA external registers to the start of the device's device registers. The difference, when divided by 128, is the controller's MBA position number.

F.5.2 The MASSBUS Adapter and the I/O Data Base

The unit control block of a device connected to a single unit controller, at offset UCB$L_CRB, contains the address of the MASSBUS adapter's channel request block. This CRB in turn contains, at offset CRB$L_INTD+VEC$L_IDB, the address of the MASSBUS interrupt dispatch block. This IDB points to the base address of the MASSBUS adapter registers at offset IDB$L_CSR.

Multiunit controllers maintain a more complicated I/O data base. The device UCB, at offset UCB$L_CRB, points to the multiunit controller's channel request block, and this structure points to the CRB for the MASSBUS adapter at offset CRB$L_LINK. Also, the multiunit controller's CRB points to its own IDB at offset CRB$L_INTD+VEC$L_IDB. This IDB points to the multiunit controller's device registers at offset IDB$L_CSR.

Thus, the unit control block for a device always points to that device's primary channel request block, whether it is the MASSBUS adapter's CRB or the multiunit controller's CRB. The primary channel request block points to the secondary CRB, if one exists for the device.

Figure F-5 shows these relationships among I/O data structures.

F.5.3 Considerations for the Start I/O Routine

Depending on the function being executed, the start I/O routine for a MASSBUS device performs all or some of the following tasks:

- Requests controller data channel(s) as described in Section F.5.3.1
- Clears errors on the MASSBUS adapter by setting the value -1 into the MBA Status Register; this is a write-ones-to-clear register (MASSBUS device registers and MBA registers are all longwords)
- Invokes the LOADMBA macro to load MBA map registers as described in Section F.5.3.2
- Loads device registers to start the function
- Waits for a device interrupt or timeout
- Releases controller data channel(s) as described in Section F.5.3.3
- Finishes the request like other drivers

F.5.3.1 Requesting Controller Data Channels - Device drivers for MASSBUS devices must request and receive ownership of the MASSBUS adapter channel before loading MBA internal registers or MBA map registers. In addition, drivers for devices connected to multiunit controllers must obtain ownership of the controller channel before modifying the contents of controller registers that may be shared among the units connected to the controller.

Drivers for single unit controllers must request ownership of the MASSBUS adapter channel by invoking the macro REQPCHAN.

Device drivers for multiunit controllers invoke the REQPCHAN macro when the operation requires ownership of only the primary channel (the multiunit controller channel). However, if the operation requires ownership of both primary and secondary channels (a data transfer operation), the driver must first obtain the controller channel and then request the MASSBUS adapter channel by invoking the REQSCHAN macro.

Again, the driver needs ownership of both channels only when performing a data transfer, and must release the channels before initiating a nondata transfer. Thus, a driver must obtain ownership of the MASSBUS adapter channel sometime before initiating a data transfer and must either not own the channel or release such ownership before it invokes the WFIKPCH macro following the start of a nondata transfer operation.

F.5.3.2 Loading Map Registers - MASSBUS device drivers invoke the LOADMBA macro before they initiate a data transfer to load the MBA map registers, the MBA Virtual Address Register, and the MBA Byte Count Register. Drivers cannot modify these registers during a transfer. The LOADMBA macro expects the following register contents:

- The address of the MBA Configuration Register in R4
- The address of the device UCB in R5

LOADMBA preserves the contents of R3 but modifies R0 through R2. The macro performs the following steps:

- Uses the contents of UCB$W_BCNT and UCB$W_BOFF to determine the number of pages that contain pieces of the I/O buffer
- Beginning with the page table entry pointed to by UCB$L_SVAPTE and continuing for the number of page table entries determined in the step above, copies the page frame numbers from the page table entries to the corresponding map registers, starting at Map Register 0
- Deposits an invalid value into the map register that immediately follows the last map register loaded with a PFN so that a hardware fault does not modify memory
- Moves the negative value of the transfer byte count (UCB$W_BCNT) into the MBA Byte Count Register
- Moves the byte offset in the first page of the transfer (UCB$W_BOFF) into the MBA Virtual Address Register
- Returns to the start I/O routine that invoked it

If the I/O operation about to be initiated by the driver is a reverse (that is, a read reverse on tape) operation, the driver must modify the contents of the MBA Virtual Address Register set up by LOADMBA. Since reverse operations access the I/O buffer from its highest address through its lowest address, the value to be loaded into the MBA Virtual Address Register must be the virtual address, in MBA virtual memory, of the last byte of the buffer. This value is numerically equal to one less than the sum of the contents of UCB$W_BOFF and UCB$W_BCNT.

F.5.3.3 Releasing Controller Data Channels - The driver releases the controller data channels by invoking the RELCHAN macro. RELCHAN releases all controller channels (both primary and secondary) currently owned by the device. To release only the secondary channel and retain ownership of the primary channel, a driver can invoke the RELSCHAN macro.

F.5.4 Considerations for the DPTAB Macro

The device driver for a MASSBUS device attached to a multiunit controller must set the DPT$M_SUBCNTRL bit in the FLAGS argument of the DPTAB macro. Setting this bit causes the driver loading procedure to create a second channel request block and an interrupt data block for the multiunit controller.

F.6 INTERRUPT SERVICE ROUTINES FOR MASSBUS DEVICES

The VAX/VMS MASSBUS interrupt dispatcher (MBA$INT) gains control when it receives an interrupt from the MASSBUS adapter. Because data transfers in progress suppress attention interrupts on the MASSBUS adapter, and because several devices may request attention simultaneously, certain device drivers may need to be informed of the interrupt. MBA$INT determines which drivers should be invoked as a result of the interrupt and then passes control to these drivers. For data transfer interrupts, MBA$INT preserves the value contained in the

MBA Status Register at the time of the interrupt so that the driver can have access to this value. For nondata transfers, MBA$INT clears this register before invoking the driver following receipt of an interrupt. MBA$INT only preserves the contents of registers R2 through R5. Drivers wishing to use other registers must save and restore them themselves.

F.6.1 Transferring Control to the Interrupt Service Routine

The method by which MBA$INT invokes a driver depends upon whether the driver services a device connected to a single unit controller or a device connected to a multiunit controller. Furthermore, if the device is connected to a single unit controller, the method of transfer from MBA$INT to the driver depends upon whether or not the interrupt is expected.

For single unit controller devices expecting interrupts, MBA$INT restores the driver context saved in the UCB fork block and transfers control (using a JSB instruction) to the instruction that follows the wait for interrupt.

For single unit controller devices not expecting interrupts, MBA$INT obtains the address of the driver's unsolicited interrupt routine from the driver dispatch table and calls the routine at the specified address.

For multiunit controller devices, MBA$INT transfers control to the driver's interrupt service routine by simulating a direct transfer, through an interrupt vector, to the multiunit controller's CRB. The CRB contains code that transfers control to the interrupt service routine. MBA$INT first pushes the processor status longword (PSL) onto the stack. The routine then calls (with a JSB instruction that leaves a PC within MBA$INT on the stack) the code within the CRB. This code contains the following sequence of instructions:

```
PUSHR     #M<R2,R3,R4,R5>
JSB       XX$INT
.LONG     XX$IDB
```

where XX$INT is the address of the interrupt service routine and XX$IDB is the address of the multiunit controller's IDB.

The execution of the above sequence of instructions, plus the instructions executed by MBA$INT (the pushing of the PSL onto the stack and the JSB) places a simulated interrupt frame onto the stack, including a saved PSL, a saved PC, saved registers and pointer to a pointer to the IDB.

F.6.2 Returning Control to MBA$INT

The way in which a driver returns control to MBA$INT depends on the way in which MBA$INT invoked it. Drivers for single unit controller devices return to MBA$INT through an RSB instruction, although the RSB may execute as a result of the driver's invoking the IOFORK macro.

Drivers for multiunit controller devices return control to MBA$INT by removing the indirect pointer to the IDB from the top of the stack, restoring registers R2 through R5, and executing an REI instruction. This sequence, executed within the driver's interrupt service routine, eliminates the simulated interrupt frame from the stack before returning to MBA$INT.

F.6.3 Considerations for Interrupt Service Routines

Drivers for single unit controller devices attached to the MASSBUS do not have interrupt services routines. Instead, MBA$INT handles all the functions that a driver interrupt service routine normally provides.

Drivers for multiunit controller devices on the MASSBUS must have their own interrupt services routines. In general, these routines perform the same functions as the interrupt service routines for UNIBUS devices (discussed in Chapter 11). However, UNIBUS and MASSBUS drivers diverge in two areas.

One difference between UNIBUS and MASSBUS drivers concerns the number of registers saved by the interrupt service routine. When the interrupt dispatcher transfers control to a MASSBUS driver interrupt service routine, registers R2 through R5 are pushed onto the stack. UNIBUS drivers save R0 through R5.

After handling an interrupt, both MASSBUS and UNIBUS driver interrupt service routines execute an REI instruction. For UNIBUS devices, the REI dismisses a real interrupt, whereas the MASSBUS driver's REI returns control to MBA$INT.

APPENDIX G

UNIBUS ADDRESSES FOR VAX-11 PROCESSORS

This appendix lists the starting physical addresses for VAX-11/780, VAX-11/750, and VAX-11/730 UNIBUS memory address space. The values in the table below are given in hexadecimal format.

		VAX-11/730	VAX-11/750	VAX-11/780
UNIBUS adapter number	0	00FC0000	00FC0000	20100000
	1	--	00F80000	20140000
	2	--	--	20180000
	3	--	--	201C0000

NOTE

The macros $IO730DEF, $IO750DEF and $IO780DEF define symbolic constants for the base address of UNIBUS space 0.

GLOSSARY

GLOSSARY

ACP

See Ancillary Control Process.

adapter control block (ADP)

A structure in the I/O data base that describes either a UNIBUS or MASSBUS adapter.

ADP

See adapter control block.

allocate a device

To reserve a particular device unit for exclusive use. A user process can allocate a device only when that device is not allocated by any other process.

Ancillary Control Process (ACP)

A process that acts as an interface between user software and an I/O driver. An ACP provides functions supplemental to those performed in the driver, such as file and directory management. Three examples of ACPs are: the Files-11 ACP (F11ACP), the magnetic tape ACP (MTAACP), and the networks ACP (NETACP).

assign a channel

To establish the necessary software linkage between a user process and a device unit before a user process can communicate with that device. A user process requests the system to assign a channel and the system returns a channel number.

AST

See Asynchronous System Trap.

ASTLVL

See Asynchronous System Trap Level.

Asynchronous System Trap (AST)

A software-simulated interrupt to a user-defined service routine. ASTs enable a user process to be notified asynchronously with respect to its execution of the occurrence of a specific event. If a user process has defined an AST routine for an event, the system interrupts the process and executes the AST routine when that event occurs. When the AST routine exits, the system resumes the process at the point where it was interrupted.

Asynchronous System Trap Level (ASTLVL)

A value kept in an internal processor register that is the highest access mode for which an AST is pending. The AST does not occur until the current access mode drops in privilege (rises in numeric value) to a value greater than or equal to ASTLVL. Thus, an AST for an access mode will not be serviced while the processor is executing in a more privileged access mode.

backplane interconnect

An internal processor bus that UNIBUS and MASSBUS adapters use to communicate with main memory and the central processor. The backplane interconnect is called the synchronous backplane interconnect (SBI) on the VAX-11/780 processor, and is called the memory interconnect on the VAX-11/750 processor.

base register

A general register used to contain the address of the first entry in a list, table, array, or other data structure.

buffered data path

A UNIBUS adapter data path that transfers multiple bytes of data in a single backplane interconnect transfer.

buffered I/O

See system buffered I/O.

bugcheck

The operating system's internal diagnostic check. The system logs the failure and crashes the system.

call instructions

The processor instructions CALLG (Call Procedure with General Argument List) and CALLS (Call Procedure with Stack Argument List).

CCB

See channel control block.

channel

A logical path connecting a user process to a physical device unit. A user process requests the operating system to assign a channel to a device so the process can communicate with that device. See also controller data channel.

channel control block (CCB)

A structure in the I/O data base maintained by the Assign I/O channel system service to describe the device unit to which a channel is assigned.

channel request block (CRB)

A structure in the I/O data base that describes the activity on a particular controller. The channel request block for a controller contains pointers to the wait queue of drivers ready to access a device through the controller.

configuration register

A control/status register for an adapter, for example a UNIBUS adapter. It resides in the adapter's I/O space.

connect-to-interrupt

A function by which a process connects to a device interrupt vector. To perform a connect-to-interrupt, the process must map to the program I/O space containing the vector.

console

The manual control unit integrated into the central processor. The console includes a serial line interface connected to a hard-copy terminal. This enables the operator to start and stop the system, monitor system operation, and run diagnostics.

console terminal

The hard-copy terminal connected to the central processor console.

context

The environment of an activity. See also process context, hardware context, and software context.

controller data channel

A logical path to which a driver for a device on a multiunit controller must be granted access before it can activate a device.

control/status register (CSR)

A control/status register for a device or controller. It resides in the processor's I/O space.

CRB

See channel request block.

CSR

See control/status register.

data base

(1) All the occurrences of data described by a data base management system.

(2) A collection of related data structures.

data structure

Any table, list, array, queue, or tree whose format and access conventions are well-defined for reference by one or more images.

DDB

See device data block.

DDT

See driver dispatch table.

device data block (DDB)

A structure in the I/O data base that identifies the generic device/controller name and driver name for a set of devices attached to the same controller.

device interrupt

An interrupt received on interrupt priority levels 20 through 23. Device interrupts can be requested only by devices, controllers, and memories.

device register

A location in device controller logic used to request device functions (such as I/O transfers) and/or report status.

device unit

One drive and its controlling logic, for example, a disk drive or terminal. Some controllers can have several device units connected to a single controller; for example, mass storage controllers.

diagnostic

A program that tests hardware, firmware, peripheral operation, logic, or memory and reports any faults it detects.

direct data path

A UNIBUS adapter data path that transfers multiple bytes of data in a single backplane interconnect transfer.

direct I/O

An I/O operation in which VAX/VMS locks the pages containing the associated buffer in physical memory for the duration of the I/O operation. The I/O transfer takes place directly from the process buffer. Contrast with system buffered I/O.

DPT

See driver prologue table.

drive

The electromechanical unit of a mass storage device system on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

driver

The set of code and tables that handles physical I/O operations to a device.

driver dispatch table (DDT)

A table in the I/O driver that lists the entry point addresses of standard driver routines and the sizes of diagnostic and error logging buffers for the device type.

driver fork level

The interrupt priority levels at which a driver fork process executes, that is, IPLs 8 through 11. Every unit control block indicates the driver fork level for its unit.

driver prologue table (DPT)

A table in the driver that describes the driver and the device type to the VAX/VMS procedure that loads drivers into the system.

driver start I/O routine

See start I/O routine.

ECC

Error Correction Code.

error logger

A system process that empties the error log buffers and writes the error messages into the error file. Errors logged by the system include memory system errors, device errors and timeouts, and interrupts with invalid vector addresses.

exception

An event detected by the hardware or software (other than an interrupt or jump, branch, case, or call instruction) that changes the normal flow of instruction execution. An exception is always caused by the execution of an instruction or set of instructions (whereas an interrupt is caused by an activity in the system independent of the current instruction). There are three types of hardware exceptions: traps, faults, and aborts. Examples are: attempts to execute a privileged or reserved instruction, trace traps, compatibility mode faults, breakpoint instruction execution, and arithmetic traps.

executive

The generic name for the collection of procedures included in the operating system software that provide the basic control and monitoring functions of the operating system.

FDT

See function decision table.

FDT routines

Driver routines called by the Queue I/O Request system service to perform device-dependent preprocessing of an I/O request.

fork block

That portion of a unit control block that contains a driver's context while the driver is waiting for a resource. A driver awaiting the processor resource has its fork block linked into the fork queue.

fork dispatcher

A VAX/VMS interrupt service routine that is activated by a software interrupt at a fork interrupt priority level. Once activated, it dispatches driver fork processes from a driver fork queue until no processes remain in the queue for that IPL.

fork process

A fork process is a minimal context process that executes code under a series of constraints: it executes at raised interrupt priority levels; it uses R0 through R5 only (other registers must be saved and restored); it executes in system virtual address space; it is only allowed to refer to and modify static storage that is never modified by higher interrupt priority level code. VAX/VMS uses software interrupts and fork processes to synchronize executive operations.

fork queue

A queue of driver fork blocks that are awaiting activation at a particular IPL by the VAX/VMS fork dispatcher.

function code

See I/O function code.

function decision table (FDT)

A table in the driver that lists all valid function codes for the device and lists the addresses of I/O preprocessing routines associated with each valid function.

function modifier

See I/O function modifier.

generic device name

A device name that identifies the type of device but not a particular unit; a device name in which the specific controller and/or unit number is omitted. When discussing device drivers, the generic device name contains neither the controller designation nor the unit number, for example, DB.

hardware context

The values contained in the following registers while a process is executing: the PC; the PSL; the 14 general registers (R0 through R13); the four processor registers (P0BR, P0LR, P1BR and P1LR) that describe the process virtual address space; the SP for the current access mode in which the processor is executing; plus the contents to be loaded in the SP for every access mode other than the current access mode. While a process is executing, its hardware context is continually being updated by the processor. While a process is not executing, its hardware context is stored in its hardware PCB.

hardware process control block (hardware PCB)

A data structure known to the processor that contains the hardware context when a process is not executing. A process's hardware PCB resides in its process header (PHD).

IDB

See interrupt dispatch block.

interrupt

An event other than an exception or branch, jump, case, or call instruction that changes the normal flow of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs. See also device interrupt, software interrupt, and urgent interrupt.

interrupt dispatch block (IDB)

A structure in the I/O data base that describes the characteristics of a particular controller and points to devices attached to that controller.

interrupt priority level (IPL)

The interrupt level at which a software or hardware interrupt is generated. There are 32 possible interrupt priority levels: IPL 0 is lowest, 31 is highest. The levels arbitrate contention for processor service. For example, a device cannot interrupt the processor if the processor is currently executing at an interrupt priority level greater than the interrupt priority level of the device's interrupt service routine.

interrupt service routine (ISR)

A routine executed when a device interrupt occurs.

interrupt stack (IS)

The system-wide stack used when executing in interrupt service context. At any time, the processor is either in a process context executing in user, supervisor, executive, or kernel mode, or in system-wide interrupt service context operating in kernel mode, as indicated by the interrupt stack and current mode bits in the PSL. The interrupt stack is not context switched.

interrupt stack pointer (ISP)

The stack pointer for the interrupt stack. Unlike the stack pointers for process context stacks, which are stored in the hardware PCB, the interrupt stack pointer is stored in an internal processor register.

interrupt vector

See vector.

I/O data base

A collection of data structures that describes I/O requests, controllers, device units, volumes, and device drivers in a VAX/VMS system. Examples are the driver dispatch table, driver prologue table, device data table, unit control block, channel request block, I/O request packet, and interrupt data block.

I/O driver

See driver.

I/O function

An I/O operation interpreted by the operating system and typically resulting in one or more physical I/O operations.

I/O function code

A 6-bit value specified in a Queue I/O Request system service that describes the particular I/O operation to be performed (such as, read, write, rewind).

I/O function modifier

A 10-bit value specified in a Queue I/O Request system service that modifies an I/O function code (for example, read terminal input no echo).

I/O lockdown

The state of a page such that it cannot be paged or swapped out of memory.

I/O request packet (IRP)

A structure in the I/O data base that describes an individual I/O request. The Queue I/O Request system service creates an I/O request packet for each I/O request. VAX/VMS and the driver of the target device use information in the I/O request packet to process the request.

I/O rundown

An operating system function in which the system cleans up any I/O in progress when an image exits.

I/O space

The regions of physical address space that contain the configuration registers, and device control/status and data registers. These regions are physically discontiguous.

I/O status block (IOSB)

A data structure associated with the Queue I/O Request system service. This service optionally returns a status code, number of bytes transferred, and device/function-dependent information in an I/O status block. The information returned is not returned from the service call, but filled in by VAX/VMS when the I/O request completes.

IPL

See interrupt priority level.

IRP

See I/O request packet.

ISP

See interrupt stack pointer.

ISR

See interrupt service routine.

limit

The size or number of given items requiring system resources (such as mailboxes, locked pages, I/O requests, or open files) that a job is allowed to have at any one time during execution, as specified by the system manager in the user authorization file. See also quota.

locking a page in memory

Making a page in an image ineligible for either paging or swapping. A page stays locked in physical memory until VAX/VMS specifically unlocks it.

logical I/O function

A set of I/O operations (for example, read and write logical block) that allow restricted direct access to device level I/O operations using logical block numbers.

mailbox

A software data structure that is treated as a record-oriented device for general interprocess communication. Communication using a mailbox is similar to other forms of device-independent I/O. Senders write to a mailbox; the receiver reads from that mailbox. Some system-wide mailboxes are defined: the error logger and OPCOM read from system-wide mailboxes.

MASSBUS adapter (MBA)

An interface device between the backplane interconnect and the MASSBUS.

memory interconnect

The internal processor bus for the VAX-11/750.

offset

A fixed displacement from the beginning of a data structure. System offsets for items within a data structure normally have an associated symbolic name used instead of the numeric displacement. Where symbols are defined, programmers always reference the symbolic names for items in a data structure instead of using the numeric displacement.

page frame number (PFN)

The high-order 21 bits of the physical address of a page in physical memory.

page table entry (PTE)

The data structure that identifies the physical location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page frame number needed to map the virtual page to a physical page. When it is not in memory, the page table entry contains the information needed to locate the page on secondary storage (disk).

PCB

See Process Control Block.

PFN

See page frame number.

physical address

The address used by hardware to identify a location in physical memory or on directly-addressable secondary storage devices such as a disk. A physical memory address consists of a page frame number and the number of a byte within the page. A physical disk block address consists of a cylinder or track and sector number.

physical address space

The set of all possible physical addresses that can be used to refer to locations in memory (memory space) or device registers (I/O space).

physical I/O functions

A set of I/O functions that allows access to all device level I/O operations except maintenance mode.

PID

See process identification.

process

The basic entity scheduled by the system software that provides the context in which an image executes. A process consists of an address space and both hardware and software context.

process context

The hardware and software contexts of a process.

process control block (PCB)

A data structure used to contain process context. The hardware PCB contains the hardware context. The software PCB contains the software context, which includes a pointer to the hardware PCB.

process identification (PID)

A 32-bit binary value that uniquely identifies a process. Each process has a process identification and a process name.

process I/O channel

See channel.

process page tables

The page tables used to describe process virtual memory.

process priority

The priority assigned to a process for scheduling purposes. The operating system recognizes 32 levels of process priority, where 0 is low and 31 high. Levels 16 through 31 are used for real-time processes. The system does not modify the priority of a real-time process (although the system manager or process itself may). Levels 0 through 15 are used for normal processes. The system may temporarily increase the priority of a normal process based on the activity of the process.

program section (psect)

A portion of a program with a given protection and set of storage management attributes. Program sections that have the same attributes are gathered together by the linker to form an image section.

PTE

See page table entry.

QIO

Queue I/O Request system service. The VAX/VMS system service that services $QIO and $QIOW requests. The Queue I/O Request system service prepares an I/O request for processing by the driver and performs device-independent preprocessing of the request. This system service also calls driver FDT routines.

quota

The total amount of a system resource, such as CPU time, that a job is allowed to use in an accounting period, as specified by the system manager in the user authorization file. See also limit.

return status code

See status code.

SBI

See Synchronous Backplane Interconnect.

small process

A system process that has no control region in its virtual address space and has an abbreviated context. Examples are the working set swapper and the null process. A small process is scheduled in the same manner as user processes, but must remain resident until it completes execution; that is, it cannot be swapped.

software context

The context maintained by VAX/VMS to describe a process. See software process control block (PCB).

software interrupt

An interrupt generated on interrupt priority level 1 through 15, which can be requested by software.

software process control block (software PCB)

The data structure used to contain a process's software context. The operating system defines a software PCB for every process when the process is created. The software PCB includes the following kinds of information about the process: current state; storage address if it is swapped out of memory; unique identification of the process; and address of the process header (which contains the hardware PCB). The software PCB resides in system region of virtual address space. It is not swapped with a process.

start I/O routine

The routine in a device driver that is responsible for obtaining necessary resources, for example, the controller data channel, and activating the device unit. **status code**

A longword value that indicates the success or failure of a specific function. For example, system services always return a status code in R0 upon completion.

SVA

See system virtual address.

Synchronous Backplane Interconnect (SBI)

The part of the VAX-11/780 hardware that interconnects the processor, memory controllers, MASSBUS adapters, the UNIBUS adapter.

system buffered I/O

An I/O operation, such as terminal or mailbox I/O, in which an intermediate buffer from the system buffer pool is used instead of a process-specified buffer. Contrast with direct I/O.

System Page Table (SPT)

The data structure that maps the system virtual addresses, including the addresses used to refer to the process page tables. The SPT contains one PTE for each page of system virtual memory. The physical base address of the SPT is contained in a processor register called SBR.

system virtual address (SVA)

A virtual address identifying a location mapped to an address in system space.

timeout

The expiration of the time limit in which a device is to complete an I/O transfer. The driver's wait for interrupt request specifies the timeout limit.

timer

A system process that maintains the time of day and the date. It also scans for device timeouts and performs time-dependent scheduling upon request. The timer interrupt service routine creates the timer process.

UCB

See unit control block.

UNIBUS adapter

An interface device between the backplane interconnect and the UNIBUS. On the VAX-11/780, this device is called the UBA. On the VAX-11/750, it is called the UBI.

unit control block (UCB)

A structure in the I/O data base that describes the characteristics of and current activity on a device unit. The unit control block also holds the fork block for its unit's device driver; the fork block is a critical part of a driver fork process. The UCB also provides a static storage area for the driver.

unit initialization routine

The routine that readies controllers and device units for operation. Controllers and device units require initialization after a power fail and during the driver loading procedure.

urgent interrupt

An interrupt received on interrupt priority levels 24 through 31. These can be generated only by the processor for the interval clock, serious errors, and power fail.

vector

(1) An interrupt or exception vector is a storage location known to the system that contains the starting address of a routine to be executed when a given interrupt or exception occurs. The system defines separate vectors for each interrupting adapter and for classes of exceptions. Each system vector is a longword.

(2) For the purpose of exception handling, users can declare up to two software exception vectors (primary and secondary) for each of the four access modes. Each vector contains the address of a condition handler.

(3) A one-dimensional array.

virtual I/O functions

A set of I/O functions that must be interpreted by an ancillary control process.

wait for interrupt request

A request made by a driver's start I/O routine after it activates a device. The request causes the driver fork process to be suspended until the device requests an interrupt or the device times out.

XDELTA

A tool for debugging operating systems and drivers.

INDEX

VAX/VMS
Guide to Writing
a Device Driver
AA-H499C-TE

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

— — Do Not Tear - Fold Here and Tape —

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03061

— — Do Not Tear - Fold Here —