Writing a Device Driver for VAX/VMS

Order Number: AA–Y511B–TE

April 1986

This manual describes how to write a driver for a device connected to a VAX processor. It discusses the required and optional components of a driver, and explains their functions. It details the requirements VAX/VMS imposes upon driver code and includes guidelines for creating, loading, and debugging a driver module. It also describes data structures and other methods by which a driver and the VAX/VMS system communicate information and synchronize their execution.

Revision/Update Information:	This book supersedes the *Guide to Writing a Device Driver for VAX/VMS*, order number AA-Y511A-TE, published September, 1984.
Operating System and Version:	VAX/VMS Version 4.4
Software Version:	VAX/VMS Version 4.4

digital equipment corporation
maynard, massachusetts

April 1986

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	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXBI
DECnet	MASSBUS	VAXcluster
DECsystem–10	PDP	VMS
DECSYSTEM–20	PDT	VT
DECUS	RSTS	
DECwriter	RSX	digital

ZK–2836

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*

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

CANADA

Digital Equipment
of Canada Ltd.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL

Digital Equipment Corporation
PSG Business Manager
c/o Digital's local subsidiary
or approved distributor

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

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

In Canada call 800-267-6215.

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

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a registered trademark of the American Mathematical Society.

Contents

TABLES

Preface

The *Writing a Device Driver for VAX/VMS* volume provides information needed to write a device driver that runs under VAX/VMS Version 4.4 and to load it into the operating system. DIGITAL makes no guarantee that drivers written for earlier versions of VAX/VMS will execute without modification on subsequent versions of the operating system. Although the intent is to maintain the existing interface, some unavoidable changes might 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 their VAX processor and the VAX/VMS operating system. Although its discussion of the environment and components of a device driver can apply to many types of driver, the strategies discussed in the main body of the manual apply predominantly to UNIBUS and MicroVAX Q22 bus drivers. MASSBUS driver writers can find supplementary information in Appendix G.

Structure of This Document

There are three parts to this manual. Part I describes the components and environment of a device driver and provides explanations of VAX/VMS concepts critical to an understanding of a device driver's functions and role in the operating system. Part I contains the following sections:

- Section 1 describes the role and components of a VAX/VMS device driver, and introduces some operating system concepts that have an impact on driver operation.
- Section 2 provides an example of a device driver – a line printer driver, and illustrates the functions of the various parts of this driver and its interaction with the VAX/VMS operating system.
- Section 3 discusses VAX/VMS synchronization mechanisms: interrupt priority levels, fork processes and fork queues, and resource-wait queues.
- Section 4 discusses I/O bus features that govern the operation of direct-memory-access (DMA) transfers and affect the code of DMA device drivers.
- Section 5 provides an overview of I/O processing and discusses the interaction of device drivers with VAX/VMS.

Part II of this document describes how to code each part of a driver, and includes the following sections:

- Section 6 explains some general driver coding rules and conventions, and includes a template of a device driver.
- Section 7 describes how to create driver tables, including the driver-prologue table, driver-dispatch table, and function-decision table (FDT).

- Section 8 explains how to write FDT routines, use VAX/VMS-supplied FDT routines, and transfer control out of I/O request preprocessing.
- Section 9 discusses the components of a driver's start-I/O routine.
- Section 10 describes coding strategies for DMA drivers for UNIBUS and MicroVAX Q22 bus devices.
- Section 11 discusses the functions performed by an interrupt-servicing routine.
- Section 12 describes how to write I/O completion and device timeout routines.
- Section 13 describes unit- and controller-initialization routines, cancel-I/O routines, and error-logging routines.
- Section 14 examines the methods by which a device is logically connected to the processor and by which a driver is loaded into the operating system.
- Section 15 describes the use of XDELTA as a device driver debugging tool.

Part III is a reference section, and includes the following appendixes:

- Appendix A contains a set of figures and tables that describe the contents of each data structure and table in the I/O database.
- Appendix B lists the VAX/VMS macros that invoke the executive routines that perform work for the driver.
- Appendix C describes the context, synchronization, and input/output requirements of these routines.
- Appendix D discusses the environment of each of a driver's entry points.
- Appendix E includes a sample driver that operates an RL01/RL02 type disk on the UNIBUS, MicroVAX II Q22 bus, or MicroVAX I Q22 bus.
- Appendix F contains a sample driver for two connected DR11 controllers on the UNIBUS or MicroVAX II Q22 bus.
- Appendix G contains information that further describes strategies for producing a MASSBUS device driver.
- Appendix H describes the connect-to-interrupt driver interface that is available to real-time users.
- The Glossary at the end of this manual defines the vocabulary that pertains to device drivers and their environment.

Associated Documents

Before reading the *Writing a Device Driver for VAX/VMS* volume, you should have an understanding of the material discussed in the following documents:

- *VAX Hardware Handbook*
- I/O-related portions of the *VAX/VMS System Services Reference Manual*
- The section on VAX/VMS naming conventions in the *Guide to Creating Modular Procedures on VAX/VMS*
- *VAX/VMS I/O User's Reference Manual: Part I* and *VAX/VMS I/O User's Reference Manual: Part II*

You may also find useful some of the material in your processor's hardware documentation, as well as in the following books:

- *VAX/VMS System Dump Analyzer Reference Manual*
- *VAX/VMS System Manager's Reference Manual*
- *VAX/VMS Internals and Data Structures*
- *VAX/VMS Delta/XDelta Utility Reference Manual*

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.

Typographical conventions used in this book include the following:

- Generally, terms that are further explained in the glossary of this manual first appear in italic print. For example:

  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.

- Terms that serve as arguments to macros appear in boldface in the text of the manual. For example:

  If an at-sign character (@) precedes the **oper** argument, then the **exp** argument describes the address of the data with which to initialize the field.

- A symbol with a one- to six-character abbreviation indicates that you press a key on the terminal, for example, RET .

- In examples, the phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key: for instance, CTRL/C .

- A horizontal ellipsis indicates that additional parameters, values, or information can be entered. For example:

```
$LINK  /NOTRACE MYDRIVER1[,MYDRIVER2,...],-
       MYDRIVER.OPT/OPTIONS,-
       SYS$SYSTEM:SYS.STB/SELECTIVE_SEARCH
```

- Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)

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

- Command examples show in black letters all output lines or prompting characters that the system prints or displays. All user-entered commands are shown in red letters. For example:

```
>>>DEPOSIT R3 0
>>>@DMAXDT
SYSBOOT>
SYSBOOT>CONTINUE
```

- A vertical ellipsis means either that not all the data that the system would display in response to the particular command is shown or that not all the data a user would enter is shown. For example:

```
JSB     @UCB$L_FPC(R5)          ; Restore the driver process.
   .
   .
   .
;Between these instructions, the interrupt-servicing routine
;can make no assumptions about the contents of R0 through R4.
   .
   .
   .
POPR    #^M<R0,R1,R2,R3,R4,R5> ; Restore interrupt registers.
```

New and Changed Features

This manual applies to Version 4.4 of the VAX/VMS operating system. The following list summarizes the major changes to the previous edition of the manual:

- The manual now incorporates specific information on the MicroVAX I, MicroVAX II, VAX 8600, VAX 8650, VAX 8200, and VAX 8800. The discussion of the VAX 8200 and VAX 8800 processors focuses on the writing of a driver for a device on the UNIBUS (connected to the VAXBI by means of a BI-to-UNIBUS adapter (BUA)).
- Appendix H contains the discussion of mapping I/O space and using the connect-to-interrupt facility previously published in the *VAX/VMS Release Notes, Version 4.0* and the *VAX/VMS Real-Time User's Guide*.
- The following routines have been added to Appendix C.

Routine	Function
EXE$ALONPAGVAR	Allocates a system buffer from general nonpaged pool, making no attempt to allocate it from the lookaside lists
IOC$LOADMBAMAP	Loads the MASSBUS adapter's mapping registers as required by a DMA transfer
MMG$UNLOCK	Unlocks the pages of a process buffer when an attempt to lock IRP extension buffers for a direct-I/O transfer fails

- The macro CPUDISP has been added to Appendix B. CPUDISP provides a method of implementing run-time conditional code based on the CPU-type of the executing VAX processor.
- Modifications to DLDRIVER (in Appendix E) and XADRIVER (in Appendix F) since VAX/VMS Version 4.0 have been incorporated in the code listings.
- The appendix on device driver entry points (previously Appendix I) has become Appendix D, and now contains a description of the driver-unloading routine. Routines in this appendix now appear in alphabetical order.
- The appendix comparing UNIBUS and MicroVAX I DMA drivers (previously Appendix D) has been integrated into the main text of the book—most notably into Sections 1, 4, and 10.
- The appendix listing the UNIBUS addresses for the various VAX processors (previously Appendix H) has been expanded and appears as Table H-2.
- Sections 1, 3, 4, 10, 14, and 15 have been revised, rewritten, and/or reorganized. Cross-references have been added to help with the location of new and revised material.
- A more thorough description of the driver's interaction with the VAX/VMS operating system in servicing an I/O request appears throughout Section 5, and is reflected in Figure 5-2.

- The glossary contains new entries for the terms: *VAXBI, configuration-control block, direct-memory-access (DMA) transfer, nexus, node, programmed-I/O (PIO) transfer, Q22 bus,* and *scatter-gather map.*
- Various minor revisions, as well as some reorganization of material, may be apparent throughout the book. The index has been expanded to help you find the relocated material.

PART I The VAX/VMS Device Driver Environment

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.

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:

- A Queue I/O request ($QIO) system service that preprocesses an I/O request by performing those functions and checks that are common to all devices; for example, validating those arguments of the I/O request that are not device specific
- Many operating system routines that drivers can call to perform I/O preprocessing, allocate and deallocate resources, and synchronize driver execution
- A VAX/VMS I/O postprocessing routine that performs 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 servicing those aspects of an I/O operation that vary from device type to device type. In addition, drivers can call VAX/VMS system routines to perform many functions that are common to several, but not all, devices.

A device driver does not run sequentially from beginning to end. Rather, the operating system uses driver tables and other information maintained by itself and the driver to determine which driver routines to activate and when they should be activated. Because little sequential processing of driver code occurs, the VAX/VMS operating system must assume the responsibility for synchronizing the execution of the various driver routines, as well as the execution of all drivers in the system. A major purpose of this book is to describe the conventions that all VAX/VMS drivers must follow to maintain this synchronization and cooperate with the operating system in I/O request processing.

This section first defines the general functions and purposes of a VAX/VMS device driver. It then introduces VAX/VMS concepts crucial to an understanding of how device drivers work within the operating system and integral to the process of successfully writing one. It concludes with a discussion of VAX hardware pertinent to device driver strategies and a brief example of the flow of an I/O request involving a driver.

1.1 Driver Functions

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

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

1.2 Driver Components

Normally, a device driver module can consist of the routines and tables discussed in this section. With a few exceptions, which are noted throughout Section 7, the order of the various routines and tables within the driver module is not important.

1.2.1 Driver Tables

The following tables appear in every driver.

The **driver-prologue table** (DPT) defines the identity and size of the driver to the system routine that loads the driver into virtual memory and creates the associated database. With the information provided in the DPT, the driver-loading procedure can both load and reload drivers and perform the I/O-database initialization that is appropriate to either situation.

Section 7.1 describes the procedure for creating a DPT and further discusses its functions. Figure A-8 illustrates the DPT and Table A-8 describes its contents.

The **driver-dispatch table** (DDT) lists the addresses of the entry points of standard routines within the driver, and records the size of the diagnostic and error-log buffers for drivers that perform error logging. You can find additional information and instructions on how to specify a DDT in Section 7.2. An illustration of the DDT appears in Figure A-7; Table A-7 describes its contents.

The **function-decision table** (FDT) lists all valid function codes for the device, and associates valid codes with the addresses of I/O preprocessing routines, called **FDT routines**. The driver contains device-dependent FDT routines, and the VAX/VMS operating system itself provides routines (described in Section 8.5) that perform request preprocessing common to many I/O functions.

When a user process calls the $QIO system service, the system service uses the I/O-function code specified in the request to traverse the FDT and select one or more of these preprocessing routines for execution, as appropriate to the function. To prepare for the actual I/O operation, FDT routines perform such tasks as allocating buffers in system space, locking pages in memory, and validating the device-dependent arguments (**p1** through **p6**) of the I/O request. Section 7.3 provides further discussion of the FDT, and Section 8 details strategies and rules for writing, specifying, and exiting from an FDT routine.

1.2.2 Driver Routines

In addition to any FDT routines it may contain, a device driver generally contains both a start-I/O routine and an interrupt-servicing routine.

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

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

When a device posts an interrupt, its driver's **interrupt-servicing routine** determines whether the interrupt is expected or unexpected, and takes appropriate action. If the interrupt is expected, the interrupt-servicing routine reactivates the driver's start-I/O at the point of suspension. The general course of action of driver mainline code at this time is to perform device-dependent I/O postprocessing and to transfer control to the VAX/VMS operating system for device-independent I/O postprocessing. VAX/VMS synchronization plays a large part in the execution of the start-I/O routine and interrupt-servicing routine, and is discussed later in this chapter and throughout this book.

Details on writing start-I/O routines and interrupt-servicing routines appear in Sections 9 and 11, respectively.

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

The **unit-initialization routine** and **controller-initialization routine** prepare a device or controller for operation 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 needed by devices and controllers varies according to the device type. Section 13.1 provides additional information about device driver initialization routines.

A **timeout-handling routine** retries I/O operations and performs other error handling when a device fails to complete a transfer in a reasonable period of time. 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, because it is offline or some error has occurred, the system timer calls the driver's timeout handler.

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

Timeout-handling routines are discussed in Section 12.2. Register-dumping routines and driver error handling are discussed in Section 13.3.

The VAX/VMS operating system calls a driver's **cancel-I/O routine** when a user process issues a Cancel I/O on Channel ($CANCEL) system service for the device. It may also call the routine when the device's reference count goes to zero, which occurs when all users that have had assigned channels to the device have deassigned them. The discussion of the cancel-I/O routine appears in Section 13.2.

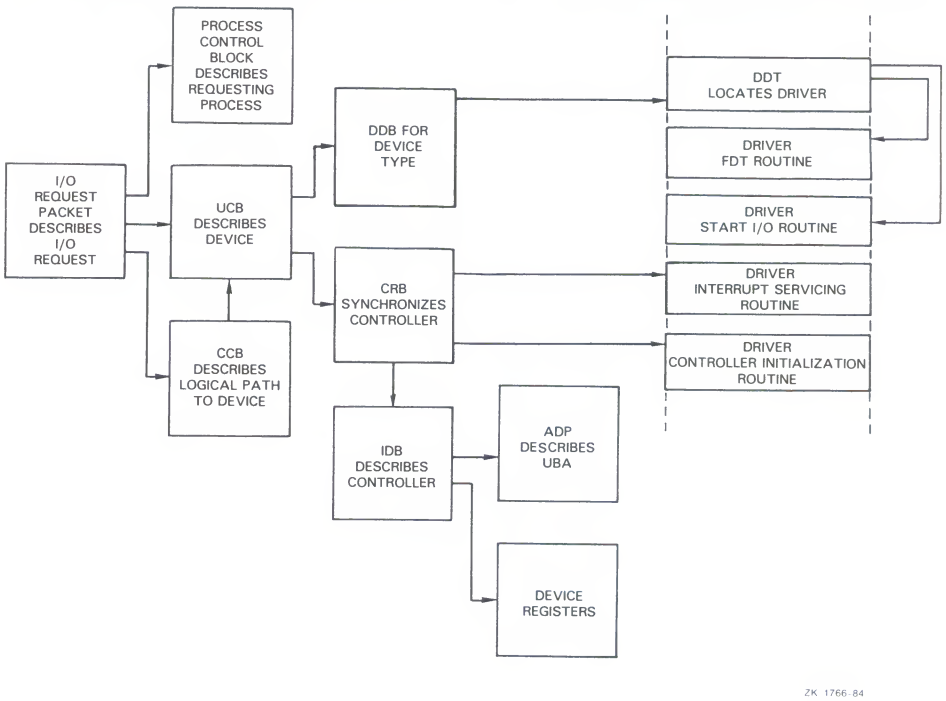
1.3 The I/O Database

Because a driver and the operating system cooperate to process an I/O request, they must have a common and current source of information about the request. This is the function of the I/O database. Under the VAX/VMS operating system, the I/O database consists of these three parts:

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

Illustrations of I/O database structures and detailed descriptions of their fields appear in Appendix A. Figure 1–1 illustrates some of the relationships among VAX/VMS I/O routines, the I/O database, and a device driver.

Figure 1–1 The I/O Database



1.3.1 Driver Tables

The three driver tables—driver-prologue table, driver-dispatch table, and function-decision table—are defined in every driver. Section 1.2 lists these tables among the other components of a device driver, and Section 7 is dedicated to a discussion of their contents.

1.3.2 Data Structures

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

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

UCBs are the focal point of the I/O database. When a driver is suspended or interrupted, the UCB keeps the context of the driver in a set of fields collectively known as a *fork block*. (See the discussion of fork blocks and fork processes in Section 1.4.1.) In addition, the UCB contains the listhead for the queue of pending I/O-request packets for the unit.

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

The operating system creates a **channel-request block** (CRB) for each controller. A CRB defines the current state of the controller and lists the devices waiting for the controller's data channel. It also contains the code that dispatches a device interrupt to the interrupt-servicing routine for that unit's driver.

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

An **adapter-control block** (ADP) defines the characteristics and current state of an I/O adapter, such as the VAX UNIBUS adapters and MASSBUS adapter, and the MicroVAX Q22 bus interface. An ADP contains the queues and allocation bit maps necessary to allocate adapter resources. VAX/VMS provides routines that drivers can call to interface with the appropriate adapter.

The **channel-control block** (CCB) describes the logical path between a process and the UCB of a specific device unit.[1] Each process owns a number of CCBs. When a process issues the Assign I/O Channel ($ASSIGN) system service, the system writes a description of the assigned device to the CCB.

Unlike the data structures mentioned earlier, a CCB is not located in nonpaged system space, but in the process' control region (P1 space).

1.3.3 I/O-Request Packets

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

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

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

[1] Channel-request blocks and channel-control blocks are two separate data structures. To help distinguish the two, it may be helpful to think of the channel-request block as the "controller-request" block because it describes the hardware controller. In contrast, the channel-control block helps manage the logical channel (the **channel** argument to the $ASSIGN and $QIO system services) by means of which a process and a device unit accomplish I/O operations.

1.4 Driver Context

Device driver code executes in a privileged access mode with a raised priority. Although FDT routines execute in *process context* and access process space (P0 and P1), the remainder of driver code must execute in *interrupt (or system) context*, and must refer only to system (S0) space. Such code cannot incur exceptions, including page faults, without a bugcheck. Code executing in interrupt context is serviced on the interrupt stack, and synchronizes execution through interrupt priority levels (discussed in Section 1.5.1) and resource-wait queues (discussed in Section 1.5.3).

1.4.1 Fork Processes

An additional restriction imposed upon drivers results from their need to save adequate (but minimal) context when their execution is suspended, and to synchronize individual aspects of I/O processing of varying importance with other privileged system and driver code.

After preprocessing an I/O request, a driver executes as a *fork process*, with a context that consists of:

- Three general registers
- The program counter (PC)
- A unit-control block in the I/O database that describes the target device of the I/O request

This context is preserved across the suspension of driver code, predominantly in a portion of the unit-control block known as a *fork block*. The system automatically saves registers for interrupted fork processes and restores these registers when the process is reactivated. Because the fork block and all data about the fork process reside in nonpaged system memory, the operating system cannot swap fork processes.

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

To minimize the number of interruptions, fork processes execute at raised interrupt priority levels, and even raise their priority level to 31 to block all other interrupts, if necessary. In addition, whenever it may be necessary to *lower* its priority level to give more important code a chance to execute, a fork process can preserve its context in the fork block, place the fork block in a fork queue at one of the interrupt-priority levels reserved for that purpose, and request a software interrupt at that level. When that interrupt is ultimately serviced, driver fork processing resumes at the lower level.

1.4.2 Example of Driver Context-Switching

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:

1. A user process executing in user mode calls the $QIO system service to write data to a device.
2. The $QIO system service gains control in process context but in kernel mode. It performs device-independent preprocessing of the I/O request.
3. The system service uses the driver's function-decision table to call the appropriate preprocessing routines. These FDT routines execute in full process context in kernel mode.
4. When preprocessing is complete, a VAX/VMS routine creates a fork process to execute the driver's start-I/O routine in kernel mode.
5. 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.
6. When the device completes the data transfer, it issues an interrupt. The interrupt causes the system to activate the driver's interrupt-servicing routine.
7. The interrupt-servicing routine executes to handle the device interrupt. It then causes the start-I/O routine to resume in interrupt context.
8. 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.
9. 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.
10. VAX/VMS I/O postprocessing performs processing at a software interrupt priority level and then issues a special kernel-mode asynchronous system trap (AST) for the user process requesting I/O.
11. When the special kernel-mode AST is delivered, the AST routine executes in full process context in kernel mode to deliver data and status to the process. If the original request specified a user-mode AST, the special kernel-mode AST queues it.
12. When the user process gains control, the user's AST routine executes in full process context in user mode.

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

1.5 Synchronization of Driver Activity

The VAX/VMS operating system uses hardware and software interrupt priority levels—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.

1.5.1 Interrupt Priority Levels

The VAX processor defines 32 interrupt priority levels (IPLs). The higher numbered IPLs (16 through 31) are reserved for hardware interrupts, such as those posted by devices. The VAX/VMS operating system uses the lower numbered IPLs (0 through 15). Code that executes at a higher IPL always takes precedence over code that executes at a lower IPL.

The following IPLs are of particular interest to drivers:

- Hardware device IPLs (20 through 23); driver interrupt-servicing routines execute at these IPLs.
- Fork-processing IPLs (8 through 11); a driver's fork process executes at one of these IPLs.
- All access to systemwide data structures, including the I/O database, must occur at IPL$_SYNCH, IPL 8.
- 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 $QIO system service also executes at this IPL.

Section 3.1 provides a thorough discussion of IPLs as used by driver code; you can find full information on the use of IPL in the *VAX Hardware Handbook* or your processor's hardware documentation.

1.5.2 Fork Queues

When an interrupt-servicing 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 the driver's 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 the driver's fork IPL.

Each fork IPL has an associated fork queue. A VAX/VMS macro queues the driver's fork block to 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 fork queue corresponding to the IPL at which the interrupt was granted and reactivates the driver at the point following the macro invocation. Refer to Section 3.2 for a detailed discussion of fork dispatching.

1.5.3 Resource-Wait Queues

Drivers compete for such shared resources as:

- The central processor
- The I/O adapter's mapping registers (if the device is a direct-memory-access (DMA) device)
- The UNIBUS adapter's buffered data paths (if the device is a UNIBUS DMA device)
- The controller's data channel (if the device is attached to a multiunit controller)

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

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

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

1. Remove the next UCB fork block from the resource-wait queue.
2. Restore fork process context to the registers.
3. Reactivate the suspended driver's fork process.

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

Additional discussion of the synchronization method of resource-wait queues appears in Section 3.3.

1.6 Hardware Considerations

The VAX/VMS operating system runs on any of the following VAX processors: the VAX 8800, VAX 8650, VAX 8600, VAX 8200, VAX–11/785, VAX–11/782, VAX–11/780, VAX–11/750, VAX–11/730, VAX–11/725, MicroVAX II, and MicroVAX I.

Although these processors employ the same operating system and conform to the VAX architecture, there are some differences in design among the machines that merit consideration in device driver coding, installation, and debugging. For instance, VAX processors differ in the amount of physical address space available and in the location of device registers. Also, VAX/VMS systems support different and various combinations of I/O buses to which a nonstandard device can be connected.

If you follow the conventions described in this manual when writing your driver, your driver should, with little modification, drive the same device attached to a corresponding I/O bus of another VAX processor. For specific processor design and device configuration information, refer to your processor's technical reference or hardware manual or the *VAX Hardware Handbook*.

1.6.1 Processor Considerations

This section outlines some of the general differences among the processors that have a bearing upon the development of driver code. The main thrust of the discussion is to provide a brief summary of the layout of the I/O subsystems of the VAX processors, define a general terminology, and, when necessary, direct device driver writers to documentation particular to the I/O configuration of their device.

1.6.1.1 VAX–11/780, VAX–11/782, VAX–11/785, VAX 8600, and VAX 8650

The VAX–11/780, VAX–11/782, VAX–11/785, VAX 8600, and VAX 8650 processors, from the viewpoint of I/O architecture, are SBI-based systems. That is, the *synchronous backplane interconnect* (SBI) is the bus by which I/O adapters communicate with main memory and the central processor (see Figure 1–2). I/O adapters supported by the SBI include the UNIBUS adapter (UBA), MASSBUS adapter (MBA), and the DR780 interface. Correspondingly, peripheral devices attach to either the UNIBUS, MASSBUS or DR32 device interconnect. Main memory shares the SBI with the I/O adapters on the VAX–11/780, VAX–11/782, and VAX–11/785. The VAX 8600 and VAX 8650 employ a separate bus to which main memory is attached and both can be configured with up to two SBIs for I/O adapters.

For these processors, nonstandard devices are commonly attached to the UNIBUS, although some nonstandard devices connect to the MASSBUS and DR32 device interconnect (DDI). The components of UNIBUS and MASSBUS drivers are identical and the strategies for producing driver code are similar; writers of either type of driver will profit from reading the bulk of this manual. In addition, MASSBUS driver writers should pay careful attention to the differences between UNIBUS and MASSBUS drivers outlined in Appendix G. DIGITAL supplies a device driver and an application library for the DR32 device; the *VAX/VMS I/O User's Reference Manual: Part II* discusses the DR32 interface driver in detail.

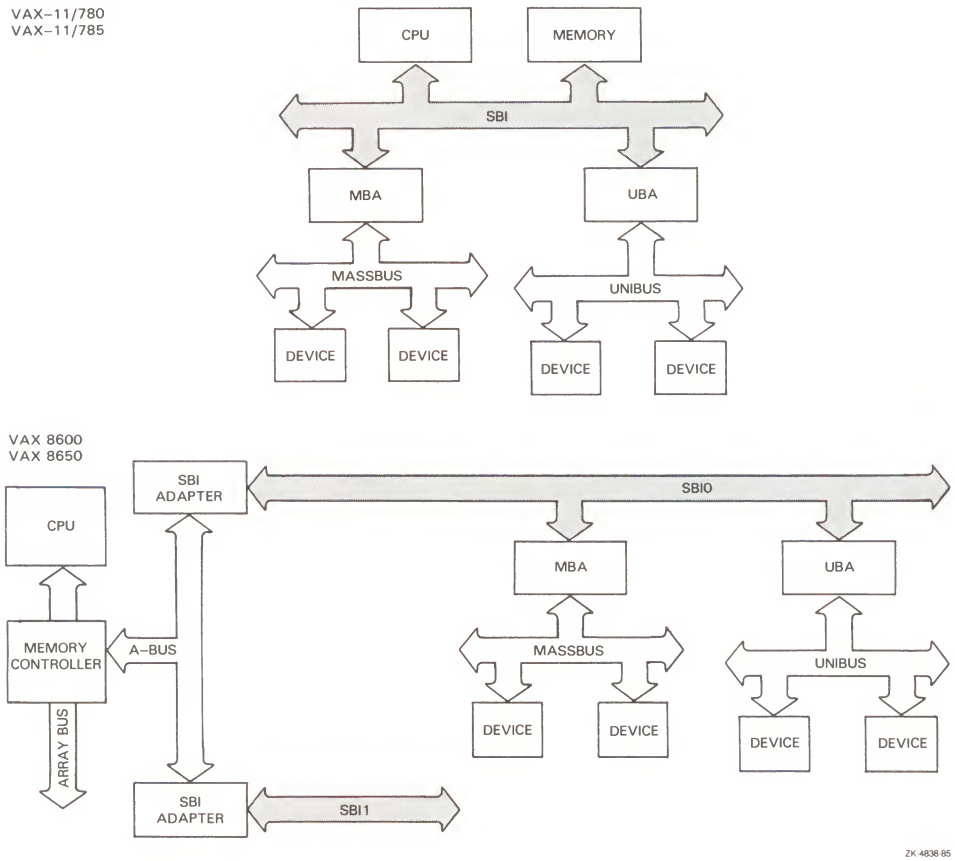
A final note on terminology regarding these processors is pertinent. For the purposes of the discussion in this book, the term *VAX–11/780* refers to the family of VAX processors that includes the VAX–11/780, the VAX–11/782, and the VAX–11/785; the term *VAX 8600* refers to both the VAX 8600 and VAX 8650; and the term *backplane interconnect* represents the SBI.

1.6.1.2 VAX–11/750

The VAX–11/750 processor resembles the VAX–11/780-type processors in that it supports both UNIBUS and MASSBUS peripheral devices (see Figure 1–2). The backplane, or CPU-to-memory interconnect (CMI), by which I/O adapters communicate with the processor and main memory is integral to the processor, as are the UNIBUS interface (UBI) and MASSBUS adapter (MBA). Peripheral devices connect to either the UNIBUS or MASSBUS. A separate memory interconnect provides an interface between main memory and the rest of the system.

For the VAX–11/750, nonstandard devices are commonly connected to the UNIBUS, although some nonstandard devices attach to the MASSBUS. The components of UNIBUS and MASSBUS drivers are identical, and the strategies for developing driver code are similar. Writers of either type of driver will profit from reading this manual. In addition, MASSBUS driver writers should pay careful attention to the differences between UNIBUS and MASSBUS drivers outlined in Appendix G.

Figure 1–2 SBI-Based System Configurations



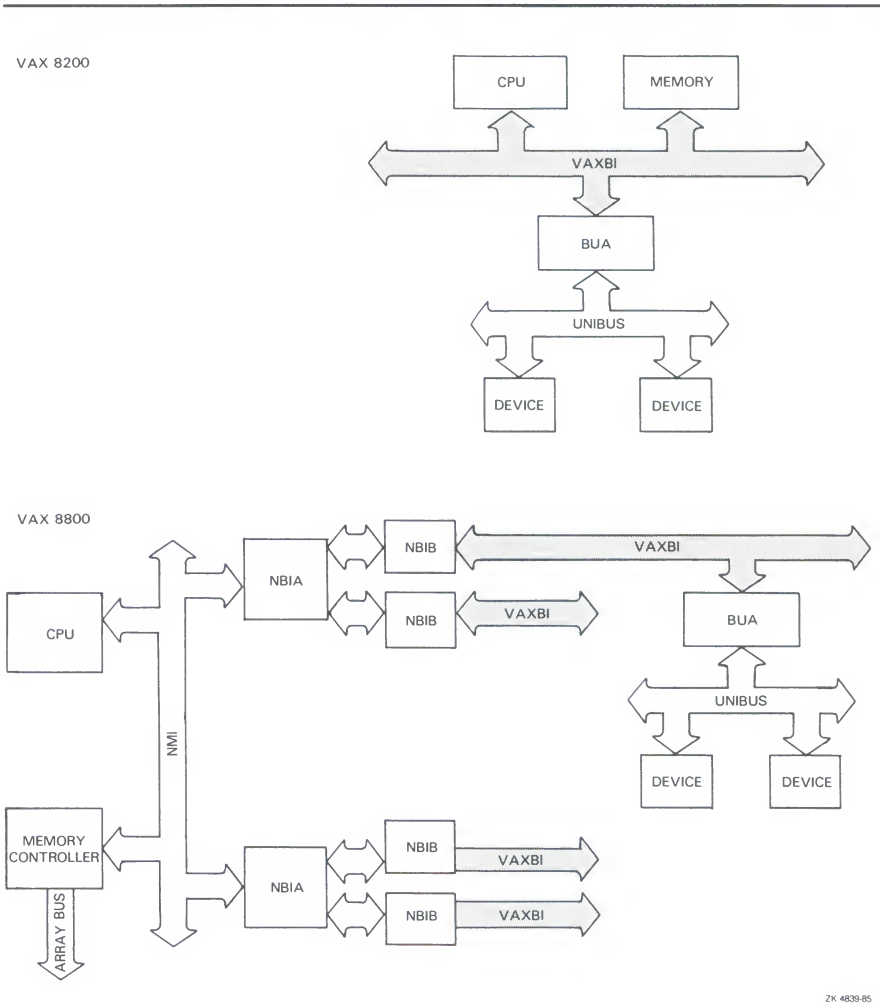
1.6.1.3 VAX–11/730 and VAX–11/725

The VAX–11/730 and VAX–11/725 processors, like the VAX–11/750, incorporate an integral UNIBUS adapter to control transactions between UNIBUS peripheral devices, the processor, and the main memory interface. The VAX–11/730 and VAX–11/725, however, do not support MASSBUS devices. For the purposes of the discussion in this book, the term *VAX–11/730* refers to both the VAX–11/730 and the VAX–11/725.

1.6.1.4 VAX 8200 and VAX 8800

The VAX 8200 and VAX 8800 are VAXBI-based systems; that is, the VAXBI is the bus by which I/O adapters communicate with main memory and the central processor (see Figure 1–3). The VAXBI supports UNIBUS peripherals by means of the BI-to-UNIBUS adapter (BUA). In the VAX 8200 configuration, main memory and the BUA are both connected directly to the VAXBI. The VAX 8800, by contrast, employs a separate memory interconnect to service main memory and can provide up to four VAXBIs for I/O adapters.

Figure 1–3 VAXBI-Based System Configurations



For these processors, nonstandard devices are attached to the UNIBUS.

A final note on terminology regarding these processors is pertinent. For the purposes of the discussion in this book, the term *UNIBUS adapter* includes the BUA, and the term *backplane interconnect* represents the VAXBI.

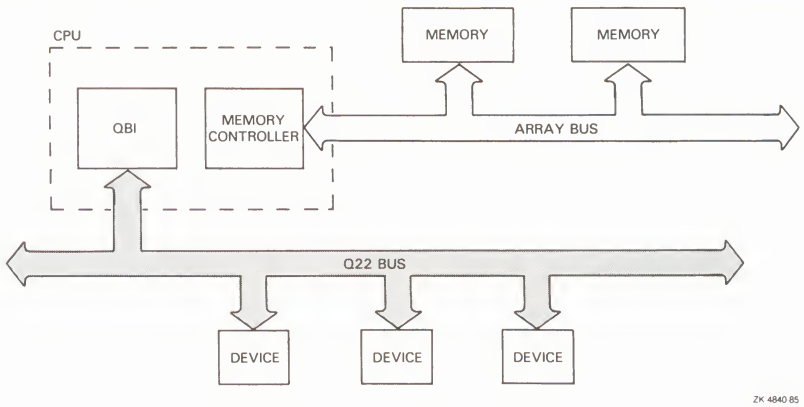
1.6.1.5 MicroVAX II and MicroVAX I

The *MicroVAX II* and *MicroVAX I* are Q22 bus-based systems. On these systems, the Q22 bus is the bus by which peripheral devices communicate with main memory and the processor.[2] Q22 bus device drivers are sufficiently similar to those that drive UNIBUS devices that most of the discussion of UNIBUS drivers in this book can equally pertain to the writing of Q22 bus device drivers (see Section 4 for a discussion of the similarities and differences).

As you can see in Figure 1–4, MicroVAX II main memory and I/O devices reside on separate interconnects. The MicroVAX II processor implements a scatter-gather map that allows devices to perform multiple-block direct-memory-access (DMA) transfers.[3]

MicroVAX I main memory and I/O devices, by contrast, exist together on the same bus (see Figure 1–5). The effects of the absence of a scatter-gather map on DMA device drivers are discussed in Section 10.7.[4]

Figure 1–4 MicroVAX II System Configuration

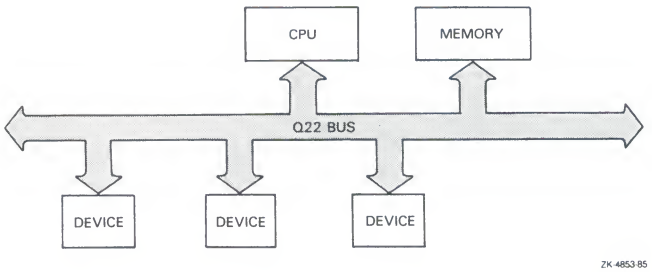


[2] DMA controllers attached to the Q22 bus must be capable of 22-bit addressing.

[3] On the MicroVAX II, the 4MB of Q22 bus memory is located from physical address 30000000 to 303F0000 hex. Because only the first 1/4 MB of this space is the area used by the scatter-gather map, the remaining 3 3/4 MB of Q22 bus memory can be used as memory local to controllers (for instance, a bit map). Such controllers should therefore be installed only after physical address 30040000 hex to avoid contention with mapped Q22 bus memory. (See Chapter 4 of the *MicroVAX II 630QB Technical Manual* for complete configuration information.) This restriction may be removed in a future release so that Q22 bus memory on a MicroVAX II can be installed at the same address as UNIBUS memory on a VAX-11/780. If you use some of the first 1/4 MB of Q22 bus memory for memory local to controllers, then MicroVMS will probably boot but will not be able to take crash dumps.

[4] The MicroVAX I uses the 22-bit Q22 bus to address both main memory and Q22 bus memory. Because MicroVAX I main memory shares the Q22 bus with I/O devices, the maximum amount of address space available for main memory (4MB at most) is correspondingly *decreased whenever* controllers containing memory are attached to the Q22 bus. For instance, if a controller containing a 256K bit map is installed on the Q22 bus, 3 3/4 MB would remain for main memory. MicroVMS is effectively prevented from using as main memory those locations addressable as controller memory by the appropriate setting of the SYSGEN parameter *PHYSICALPAGES*. In the above example, *PHYSICALPAGES* would be set to 7680 to prevent the double mapping of the 256K bit map as both main memory and controller memory.

Figure 1–5 MicroVAX I System Configuration



For the purposes of discussion in this manual, the term *backplane interconnect* represents the Q22 bus in both the MicroVAX II and MicroVAX I implementations. The terms *I/O adapter* or *Q22 bus interface* represent those functions performed by the MicroVAX II processor that resemble those performed by the UNIBUS adapter of other VAX processors.

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

Devices are equipped with various registers that initiate, control, and monitor the transfer of data to and from memory. When a transfer is complete, the device posts an interrupt to the processor. The size of the transfer concluded by a device interrupt depends upon the capabilities of the device.

1.7.1 Programmed I/O

Drivers for relatively slow devices, such as printers, card readers, terminals, and some disk and tape drives, must transfer data to a device register a byte or a word at a time. These drivers must themselves keep a record of the location of the data buffer in memory, as well as a running count of the amount of data that has been transferred to or from the device. Thus, these devices perform *programmed I/O* (PIO) in that the transfer is largely conducted by the driver program. This type of transfer is also known as *buffered I/O* because the data registers of certain PIO devices can buffer several bytes or words and transfer those bytes to the device as a group. When this is the case, the driver monitors a device status register to determine when the device buffer is full.

Examples of UNIBUS devices that do PIO transfers are the LP11 and the DZ11. Corresponding Q22 bus devices that perform PIO transfers are the LPV11 and the DZV11.

Section 2 outlines the action of the LP11 driver. The LP11 driver transfers data from a system buffer to the line printer data buffer register a byte at a time, while maintaining a count of the number of bytes left to transfer. When the line printer data buffer is full, the line printer sets a "not ready" bit in its status register. If the driver, while examining this register, sees this bit set, it enables interrupts from the printer, and then suspends itself in the expectation that the printer will post an interrupt to the processor. While the driver remains suspended, the printer prints the data from its buffer and interrupts the processor when it is done. With the interrupt handled by the system interrupt dispatcher and the driver interrupt-servicing routine, driver execution resumes. The driver repeats both its byte-by-byte transfer to the

printer data buffer, as well as the entire routine described above, until it determines that all the data has been transferred as requested.

Drivers performing PIO transfers are generally not concerned with the operation of I/O adapters. However, drivers that perform direct-memory-access (DMA) transfers must take into account I/O adapter functions, as discussed below.

1.7.2 Direct-Memory-Access I/O

Devices that perform *direct-memory-access* (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. The responsibilities of a driver for a DMA device involve supplying a device register with the starting address of the buffer containing the data to be transferred, a byte offset into the buffer, and the size of the transfer. By setting the appropriate bit or bits in the device control and status register (CSR), the driver activates the device. The device then automatically transfers the specified amount of data to or from the specified address. The VAX/VMS drivers DLDRIVER and XADRIVER are examples of DMA drivers, and appear in full in Appendixes E and F, respectively.

For DMA transfers, UNIBUS drivers and MicroVAX II drivers must first map the transfer from main memory to I/O bus memory space. The result of this mapping is a set of contiguous addresses in UNIBUS or Q22 bus space that the DMA device can access to successfully perform a DMA transfer. To accomplish this, a driver must first obtain mapping registers, and, optionally for UNIBUS drivers, a buffered data path. The driver calls VAX/VMS routines that interface with the I/O adapter to allocate these resources on behalf of the driver. Section 4 discusses the operation of the UNIBUS adapter and the Q22 bus. Section 10 provides instructions on how to write a DMA driver for UNIBUS and Q22 bus devices.

The MicroVAX I Q22 bus has no mapping registers, so no mapping of physical bus addresses to virtual memory addresses is possible. As a result, a driver for a device attached to the MicroVAX I Q22 bus that performs DMA transfers must include special logic that either allocates a physically contiguous buffer from nonpaged pool for use in the transfer or segments the transfer at page boundaries. Section 10.7 discusses the strategies for producing MicroVAX I DMA drivers.

Some controllers that can do DMA transfers on the Q22 bus have microcode that allows the controller itself to do physical-to-virtual address mapping. This allows such controllers to do scatter-gather mapping, eliminating the need for transfers to be made to or from physically contiguous main memory. The RD/RX controller, which MicroVAX I uses for its system disk, is such a controller.

1.8 Buffered and Direct I/O

Because the buffer specified in the original user I/O request is in process space, it is not automatically accessible to the driver fork process that executes in system context. As a result, for any function that involves data transfer, the driver must select a strategy that supplies a buffer that the fork process can address. The VAX/VMS operating system allows FDT routines a choice between allocating a system buffer (buffered I/O) or locking the process buffer (direct I/O).

A driver employs *buffered I/O* to allocate a buffer from nonpaged pool. It can later refer to the buffer using addresses in system space. For a write request, the driver FDT routine must move data from the user buffer to the allocated system buffer. For a read request, the system ultimately delivers the data from the system buffer to the user buffer by means of a special kernel-mode AST at driver postprocessing. Drivers most often use buffered I/O for PIO devices such as line printers and card readers.

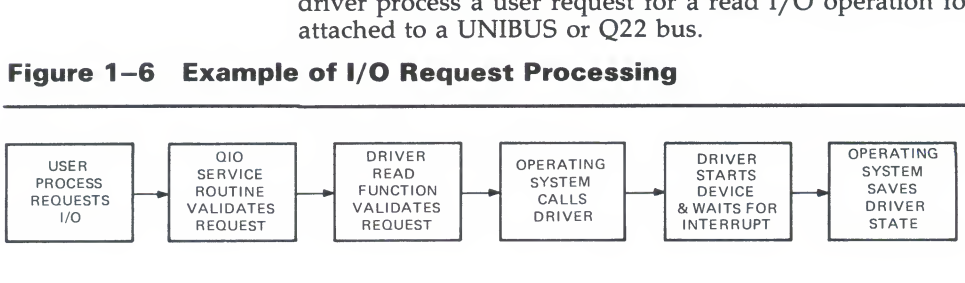
With *direct I/O*, the driver locks the pages of the user buffer in physical memory and refers to them using page-frame numbers (PFNs). Normally, a driver uses direct I/O for DMA transfers.

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. Sections 7.3.2 and 8.4 provide additional information.

1.9 Example of an I/O Request for a UNIBUS or Q22 Bus Device

Figure 1–6 illustrates how the VAX/VMS operating system and the device driver process a user request for a read I/O operation for a DMA device attached to a UNIBUS or Q22 bus.

Figure 1–6 Example of I/O Request Processing



The processing of the sample I/O request illustrated in Figure 1–6 occurs in the following steps:

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

A user process requests data from the device by issuing either a $QIO system service call or an RMS get-record function call (which results in a call to the $QIO system service).

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

2 The operating system performs I/O preprocessing.

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

3 The driver performs I/O preprocessing.

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

4 VAX/VMS creates a driver's fork process.

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

5 The driver readies the I/O adapter.

For DMA transfers, the driver's fork process calls VAX/VMS routines that enable the I/O adapter hardware to map I/O bus addresses into physical addresses for the transfer. (Note that the MicroVAX I processor does not have this capability, as discussed in Section 10.7.)

6 The driver activates the device.

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

7 The driver waits for an interrupt.

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

8 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-servicing routine.

9 The driver services the interrupt.

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

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

The driver requests that it again be suspended, to be reactivated later at a lower software IPL.

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

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

12 The driver completes the I/O operation.

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

13 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.

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

The example above simplifies the processing of an I/O operation by ignoring such issues as:

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

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

2 Discussion of a Queue-I/O Request

This chapter describes what takes place during the processing of a queue-I/O request. For simplicity, the device chosen is the LP11 printer.

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

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

This chapter describes two aspects of printer I/O processing:

- The portions of the VAX/VMS device driver for an LP11 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, this discussion assumes that it is not.

The first-time reader of this document might 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 printer driver to satisfy this I/O request.

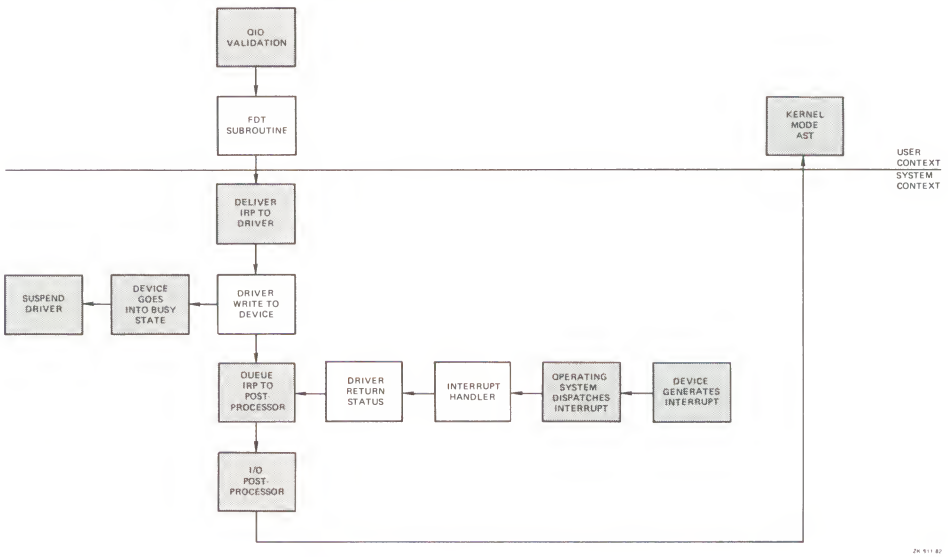
The unshaded boxes in Figure 2–1 indicate processing performed by driver subroutines. Boxes shown above the solid line indicate processing in the context of the user process. Boxes below the line indicate processing in fork or interrupt context.

2.1 Driver Code for the LP11 Write Function

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

- An FDT routine that reformats the user-supplied data
- A start-I/O routine that writes data to the device print buffer until the printer enters a busy state as it prints the buffer's contents
- Code that modifies a device register to enable interrupts from the printer
- An interrupt-servicing routine that returns control to the driver's fork process after a hardware interrupt from the printer
- Code that returns I/O status to a VAX/VMS I/O completion routine

Figure 2–1 A Printer Write Function



2.2 A User Process' I/O Request

A user process writes a line to the printer by calling the Queue I/O Request ($QIO) system service, specifying 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
```

p1, **p2**, and **p4** are device-dependent arguments.

2.3 Device-Independent I/O Preprocessing by VAX/VMS

The $QIO system service first validates that the I/O request is correctly specified. The I/O request must meet the following criteria:

- The location CHANNEL_NUMBER must contain a number that serves as a valid index into the process' channel list. This means that the process must have previously assigned the printer to this process channel using the Assign I/O Channel system service. Once $QIO locates the assigned channel-control block, it can retrieve the address of the unit-control block (UCB) of the target device of the request. Ultimately, it obtains the address of the driver's function-decision table, by way of a chain of longword pointers within the I/O database:

  CCB → UCB → DDT → FDT

- The driver 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 $QIO system service to perform a buffered-I/O request without exceeding the process' quotas.
- The process must have write access to location STATUS_BLOCK, specified in the request for use as an I/O-status block.

If all of the checks described above succeed, the $QIO system service creates an I/O-request packet (IRP) in nonpaged system address space. The service then writes all known details about the I/O request into the IRP.

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

2.4 Device-Dependent I/O Preprocessing by the Driver

Once it has validated the I/O request, the $QIO system service scans the function-decision table for an entry that associates the IO$_WRITELBLK function code with an FDT routine. The system service calls the routine, which in the case of the printer driver is a device-specific routine located in the 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 IRP, and decreases the current process byte count quota.
- It then synchronizes with other possible subprocesses[1] to read and write fields of the printer's UCB.
- It reads the description of the printer's current line and page position from the device's UCB.
- It reformats the data from the process buffer into the system buffer, adding carriage control characters, as specified in argument **p4** to the I/O request, 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.
- 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 when the request completes.

[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.

- Finally, the routine transfers control to a VAX/VMS routine that queues the IRP 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 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 IPLs that synchronize the driver's handling of the device, as described in Section 3.1.

2.5 Queuing the I/O-Request Packet to the Driver

Before queuing the IRP to the proper driver, the VAX/VMS queuing routine raises the IPL to the driver's fork level as identified in the UCB (in UCB$B_FIPL). Raising IPL to fork level synchronizes the driver's access to the UCB.

If the device is idle, which is to say that if the busy bit (UCB$V_BSY) in the I/O status word of the UCB 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 database to the driver-dispatch table, as follows:

UCB → DDT → start-I/O routine

If the device unit is busy with another transfer, VAX/VMS inserts the IRP in a queue of packets waiting for the unit. The UCB 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 Activating the Printer

The LP11 printer controller accepts data into a 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 printer sets a busy bit in the device's control and status register (CSR). Then a device driver sets the interrupt-enable bit in the device's CSR and waits for the printer to interrupt. When the printer requests a hardware interrupt, the driver can resume putting characters in the print buffer.

The driver routine writes to the 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 UCB.

UCB → CRB → IDB → CSR address

The CSR address is always the address of the printer's CSR, and all other device registers are at fixed offsets from this address. In contrast to many other devices, such as disks, the LP11 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 CSR 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 printer data buffer and decreases the count of bytes to transfer. It then repeats step 2.

4 If the device is not ready, which is to say that if the device's internal buffer is full, the driver raises IPL to 31 to block all interrupts and sets the interrupt-enable bit in the device's CSR.

After setting the interrupt-enable bit, the driver invokes a VAX/VMS wait-for-interrupt macro to suspend driver processing until the 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 UCB
- Calculating the time at which the device will time out
- Setting bits in the device's UCB to indicate that the driver expects a device interrupt within a specified time period

VAX/VMS then drops IPL back to 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 printer requests a hardware interrupt.
- VAX/VMS reports a device timeout because the 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 Handling Interrupts

When the LP11 printer requests a hardware interrupt, the interrupt dispatcher passes the interrupt to the LP11 driver's interrupt-servicing routine.

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

1 Restores the address of the UCB in R5

2 Confirms that the interrupt was expected by examining bits in the device's UCB

3 Restores the saved registers (R3 and R4) from the device's unit-control block

4 Transfers control to the driver PC address stored in the device's UCB

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

1 Saving driver context (R3, R4, and the driver PC address) in the device's UCB

2 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 fork level.

After creating the fork process, the system returns control to the driver's interrupt-servicing routine, which restores the registers saved at the time of the device interrupt and dismisses the interrupt.

2.9 I/O Postprocessing by the Driver

When the VAX/VMS fork dispatcher reactivates the driver's fork process, the driver obtains the number of characters left to transfer from the unit-control block. If there are still characters to transfer, the driver and printer repeat the procedures outlined in Sections 2.6 through 2.8, until the transfer is complete. When all characters have been transferred, the driver code branches to the driver's I/O-completion code.

The driver's I/O-completion code stores a success status code and the number of bytes transferred in R0, then transfers control to VAX/VMS to complete the I/O request.

2.10 I/O Postprocessing by VAX/VMS

The operating system inserts the IRP into an I/O postprocessing queue and requests an interrupt at IPL$_IOPOST. If another IRP is queued to the UCB for the device unit, VAX/VMS dequeues that packet and calls the driver start-I/O routine to process it. When IPL drops to IPL$_IOPOST, the processor grants the I/O postprocessing interrupt request. The I/O postprocessing dispatcher dequeues the packet for the printer I/O request and performs the following steps:

1 Increases the use count of the process' buffered I/O requests because the current operation is complete. The use count is maintained for accounting purposes.

2 Deallocates the system buffer used for the reformatted user data.

3 Increases the process' current byte count quota.

4 Sets an event flag to indicate that the I/O operation is complete.

5 Queues a special kernel-mode AST routine that will deallocate the IRP and stores I/O status into the user's I/O-status block.

The user process determines when the I/O operation is complete by the setting of the event flag and/or the filling of the I/O status block, according to the method defined in the I/O request. The Queue I/O Request and Wait ($QIOW) system service completes synchronously and returns control and status to the user process only after the I/O operation has been completed. The Synchronize ($SYNCH) system service checks the completion status of an I/O request that completes asynchronously to user process activity.

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-servicing routines
- Driver fork processes, fork blocks, and fork queues
- Resource-wait queues

When developing driver code, 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 processor defines 32 levels of hardware priority, called interrupt priority levels (IPLs). The higher-numbered IPLs (16 through 31) are reserved for hardware interrupts, and the lower-numbered IPLs (1 through 15) are reserved for software interrupts. User-mode software runs at IPL 0. Because a high IPL takes precedence over a lower IPL, a routine executing at one IPL can block interrupts at the selected IPL and all lower IPLs. This allows the operating system to assign the higher IPLs to system activities that must be dispatched quickly and with little chance of interruption, and use specific IPLs to synchronize access to shared data structures.

The hardware IPLs (16 through 31) are used for device interrupts (IPLs 20 through 23), timer interrupts, urgent conditions like power failure, and such serious errors as a machine check. Those IPLs that have a bearing on driver execution are discussed in Sections 3.1.2 and 3.1.3. For specific hardware IPL information, see your processor's hardware documentation or the *VAX Hardware Handbook.*

The software IPLs (1 through 15) are defined by VAX/VMS as illustrated in Table 3–1.

Table 3–1 IPLs Defined by VAX/VMS

IPL	Symbolic Name	Use
0	—	User-mode software
1	—	Reserved
2	IPL$_ASTDEL	Servicing of AST-delivery interrupts
3	IPL$_SCHED	Servicing of scheduler interrupts
4	IPL$_IOPOST	Servicing of I/O-postprocessing interrupts
5	—	Servicing of XDELTA interrupts on a single-processor system
6	IPL$_QUEUEAST	Fork level processing for queuing ASTs

Table 3–1 (Cont.) IPLs Defined by VAX/VMS

IPL	Symbolic Name	Use
7	IPL$_TIMERFORK	Fork level processing of timer interrupts
8	IPL$_SYNCH	Synchronizing access to system database
11	IPL$_MAILBOX	Fork level processing synchronizing access to mailboxes
8–11	—	Fork level processing for executing driver code
12–14	—	Reserved
15	—	Servicing of XDELTA interrupts on a multiprocessor system

3.1.1 Interrupt-Servicing Routines

Many IPLs have an associated interrupt-servicing routine. The processor responds to software or hardware interrupts at these IPLs by transferring control to the appropriate interrupt-servicing routine. The interrupt-servicing routine processes the interrupt and, when finished, dismisses the interrupt with an REI instruction. Execution of an REI instruction is a common way that IPL is lowered during normal execution. Because a change in IPL can alter the deliverability of pending interrupts, execution of an REI instruction triggers the delivery of many hardware and software interrupts.

The VAX/VMS operating system uses interrupt-servicing routines that gain control when the processor grants an interrupt at the levels described above, thus causing interrupts to be processed according to the following priorities:

- Device interrupts (highest priority)
- Device drivers' 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 (IRP) in the I/O postprocessing queue and requesting an interrupt at IPL 4, the I/O postprocessing IPL. When the current IPL drops below IPL 4, the processor grants the requested interrupt and transfers control to the IPL 4 interrupt-servicing routine, which completes processing of the IRP. Because VAX/VMS handles interrupts for devices, fork processes, I/O postprocessing, and AST delivery at different IPLs, it should be clear how asynchronous the processing of a single I/O request is, in that no I/O postprocessing can be performed at IPL 4 if there is a driver fork process to execute at IPL 8.

Device drivers themselves contain an interrupt-servicing routine which handles device interrupts at an appropriate device IPL (20 through 23). Also, some driver code following the device interrupt executes as a fork process, at a much lower IPL, by virtue of an interrupt-servicing routine running at a fork IPL (8 through 11). (See Sections 3.1.2.3 and 3.2 for additional information.)

Interrupt-servicing routines run in a reduced context. They can only refer to system space (S0) and are serviced on the interrupt stack. They should observe the following rules:

- Interrupt-servicing routines generally can use only registers R0 through R5. Using registers other than R0 through R5 is not recommended. However, if the interrupt-servicing routine *does* use other registers, it must save their contents before use and restore them after use.
- If the interrupt-servicing routine pushes any elements onto the stack, it must remove them before dismissing the interrupt.
- Although it can elevate IPL, an interrupt-servicing routine cannot lower IPL below the level at which the original interrupt occurred.

Refer to Section 11 for a discussion of rules and strategies for writing a driver interrupt-servicing routine.

3.1.2 IPL Use During I/O Processing

I/O processing occurs mainly at the IPLs discussed in this section.

3.1.2.1 IPL 2 (IPL$_ASTDEL)

The AST delivery interrupt-servicing routine is associated with IPL$_ASTDEL. When a system service for which an AST was specified is completed, the system service queues the AST and causes a software interrupt to be requested at IPL$_ASTDEL. The AST delivery interrupt-servicing routine gains control when IPL drops below IPL$_ASTDEL, and delivers the AST to the process that is currently scheduled. Any code executing at IPL$_ASTDEL blocks the execution of this interrupt-servicing routine.

To block the delivery of ASTs—specifically the kernel-mode AST that causes process deletion—I/O preprocessing, from the time that the $QIO system service allocates an IRP through the execution of the last FDT routine, occurs at IPLs no lower than IPL$_ASTDEL. In effect, any driver routine (such as an FDT routine) that allocates or deallocates dynamic system pool space while running in the context of a process 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 process register; if an AST that deletes the process were to occur, the allocated memory would be lost from the pool.

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

3.1.2.2 IPL 4 (IPL$_IOPOST)

The IPL$_IOPOST interrupt-servicing routine performs device-independent postprocessing of an I/O request. As appropriate to the I/O request, it adjusts process quota use, queues a special kernel-mode AST to write status and data into the process' address space, and deallocates system memory.

After they have completed whatever device-dependent postprocessing is required, drivers request I/O postprocessing by calling a VAX/VMS routine that inserts an IRP in the postprocessing queue and requests a software interrupt at IPL$_IOPOST. When the interrupt is granted, the IPL$_IOPOST interrupt-servicing routine performs all I/O-completion processing that can occur without reference to the device's unit-control block (UCB) and, thus, can occur at an IPL lower than fork IPL.

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. Because I/O postprocessing queues ASTs to processes, the scheduler might preferentially reschedule a waiting process because of a pending AST to the process.

3.1.2.3 IPL 8 through IPL 11 (Fork IPLs)

For each of the IPLs from 8 to 11, there exists a queue of fork blocks waiting to be processed. Each fork block contains the context of a suspended fork process. The interrupt-servicing routine that executes at each of these IPLs dequeues a fork block, restores the context of the fork process, and resumes its execution at the saved PC location. (Refer to Section 3.2 for a discussion of fork blocks and fork processes.)

All driver routines, except for most FDT routines, execute at fork IPL or higher. Usually driver routines should not read or alter fields of the UCB unless IPL is at fork level or higher. The fork IPL at which any individual driver fork process executes depends upon the contents of the UCB field UCB$B_FIPL. The drivers for all devices on a single I/O adapter should specify the same fork IPL if they actively compete for shared I/O adapter resources such as mapping registers and data paths.

3.1.2.4 IPL 20 through IPL 23 (Device IPLs)

Each of the IPLs from 20 to 23 is used to service a device interrupt. The UCB$B_DIPL field in the device's UCB contains an IPL value at which the device requests hardware interrupts. When a device interrupt occurs, the system transfers control to the driver's interrupt-servicing routine with IPL set to the device interrupt level. This IPL is in the range 20 through 23 because device interrupts usually need to interrupt most user and VAX/VMS software functions.[1]

In addition, device drivers sometimes raise IPL to UCB$B_DIPL or higher before reading and writing certain device registers.

[1] IPLs 20 through 23 generally correspond with the four bus request levels (BR4 through BR7) of the UNIBUS and Q22 bus. UNIBUS device IPLs are independent of the position of the devices on the bus; Q22 bus devices with higher IPL are configured closer to the CPU than devices with lower IPL.

The MicroVAX II also has four interrupt request lines (BIRQ4 through BIRQ7) but only one interrupt-acknowledge line (BIAK). In order to guarantee proper synchronization of device interrupts, the MicroVAX II central processor honors interrupts based on the correct BIRQ level of the interrupting device, but services them all at the highest device IPL (23_{10}).

Because 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. Interrupts from MicroVAX II, MicroVAX I, VAX 8200, and VAX 8800 devices, in fact, can block hardware interrupts from the processor's interval timer if they occur at or above IPL 22. To prevent the loss of an interval-timer interrupt, these drivers, when raising IPL to 22 or above, must lower IPL below 22 within 10 milliseconds. (See Sections 3.1.7 and 3.2 for a discussion of techniques for lowering IPL.)

3.1.2.5 IPL 31 (IPL$_POWER)

The highest IPL, IPL$_POWER (IPL 31) locks out all other interrupts. Many VAX/VMS routines and drivers raise IPL to IPL$_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's control register.

3.1.3 Additional IPLs

In addition to the IPLs discussed above that directly concern I/O operation, VAX/VMS defines the IPLs described in this section.

3.1.3.1 IPL 3 (IPL$_SCHED)

When the system wishes to reschedule processes, a VAX/VMS routine requests a software interrupt at IPL$_SCHED. The scheduler interrupt-servicing routine gains control at this IPL. Drivers never use IPL$_SCHED.

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

3.1.3.2 IPL 6 (IPL$_QUEUEAST)

IPL$_QUEUEAST is a fork-level IPL used predominantly by drivers written prior to V4.0 of the VAX/VMS operating system. A driver fork process originating at an IPL between 8 and 11 would use IPL$_QUEUEAST when it needed to synchronize access to the scheduler's database at IPL$_SYNCH—for instance, to queue an AST. Because IPL$_SYNCH before V4.0 was not yet a fork IPL, the only way that such a driver could maintain proper synchronization was to first call a system routine that created a fork block at IPL$_QUEUEAST. Once the IPL$_QUEUEAST fork dispatcher dequeued the fork block and resumed execution of the driver, the driver fork process could then raise IPL to IPL$_SYNCH and access the system database.

Because versions of the VAX/VMS operating system after V4.0 implement IPL$_SYNCH as a fork IPL, a similar driver fork process needs only to fork to IPL$_SYNCH. It is the IPL$_SYNCH fork dispatcher that dequeues the driver fork block and resumes execution of the driver, thereby allowing it to access the system data structures with proper synchronization.

3.1.3.3 IPL 7 (IPL$_TIMERFORK)

A timer-queue interrupt-servicing routine fields interrupts requested at IPL$_TIMERFORK. The hardware clock's interrupt-servicing routine requests a software timer interrupt at IPL$_TIMERFORK when the current process has exceeded its processor time quantum or when the first entry in the timer queue is due. The timer's interrupt-servicing routine immediately raises IPL to IPL$_SYNCH to synchronize its access to the system database, dequeues the first timer-queue entry, and takes appropriate action if it has expired.

3.1.3.4 IPL 8 (IPL$_SYNCH)

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

3.1.3.5 IPL 11 (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.

3.1.3.6 IPL 5 or IPL 15 (XDELTA IPLs)

To stop the operating system for debugging purposes, you can halt the operating system from the console terminal and request a software interrupt. (The procedure for requesting a software interrupt to load XDELTA is described in Table 3–3.) The interrupt-servicing routine that loads XDELTA runs at IPL 5 on VAX single-processor systems and at IPL 15 on VAX multiprocessing systems. The processor must be executing below the requested IPL for the interrupt to take effect.

3.1.4 Overview of IPL Use in an I/O Operation

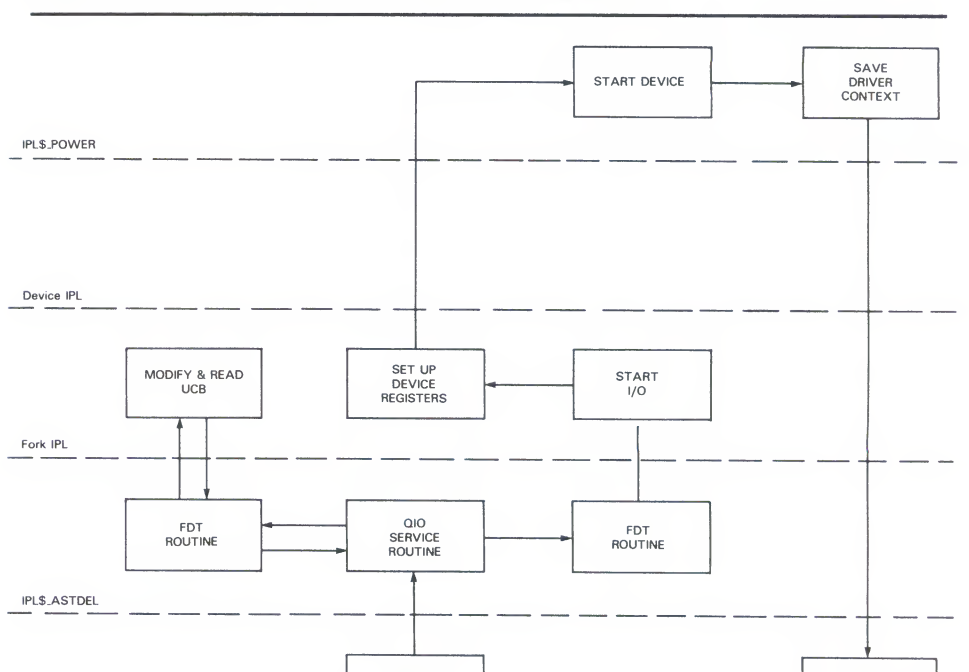
Figure 3–1 illustrates the normal IPL flow during the processing of an I/O request.

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

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

Figure 3–2 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-servicing routine. The interrupt-servicing routine reactivates the driver's fork process with IPL still at device IPL.

Figure 3–1 IPL Flow During I/O Processing

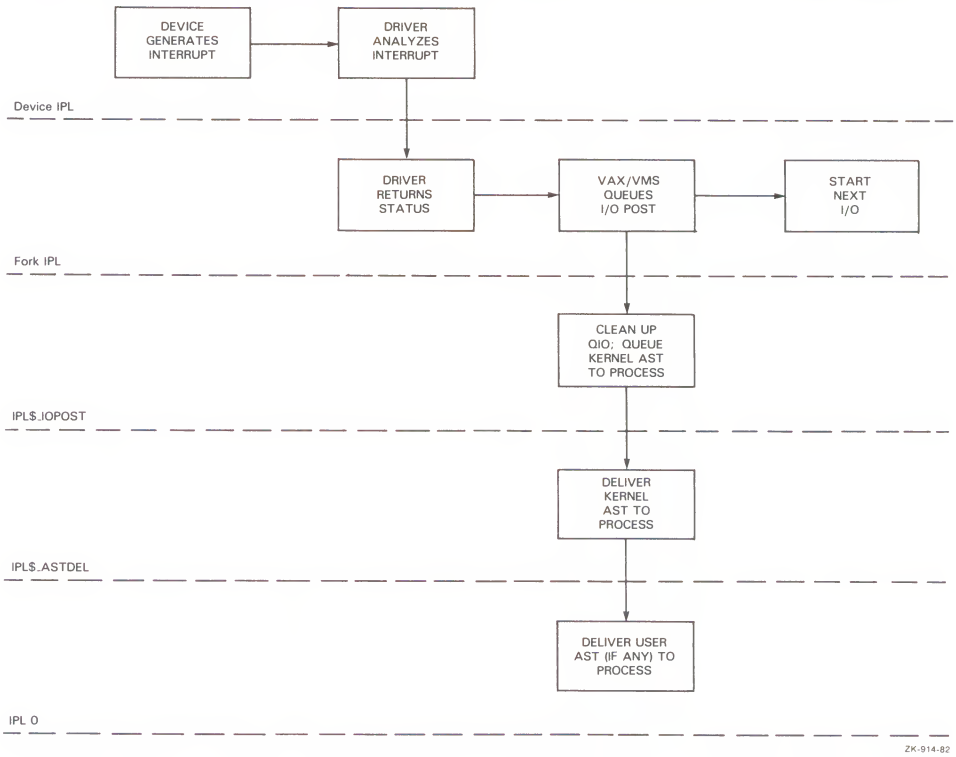


The fork process briefly examines or saves the contents of the device's registers, but soon requests that VAX/VMS insert a fork block describing its context into one of the fork queues for drivers' fork IPLs (8 through 11). When the fork process regains control at the driver's fork IPL, the process analyzes the success of the I/O operation and writes status into R0 and R1. Then, still at fork IPL, VAX/VMS inserts the IRP 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 special kernel-mode AST to the process that issued the original $QIO request.

The special kernel-mode AST routine executes at IPL$_ASTDEL. It can queue a user-mode AST routine that eventually executes at IPL 0. I/O postprocessing continues at IPL$_IOPOST until all entries in the postprocessing queue have been serviced.

Figure 3–2 IPL Flow During I/O Completion



3.1.5 Dispatching Device Interrupts

VAX 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 grants the interrupt, and then transfers control to an interrupt-servicing routine for the device. If the processor is executing at a higher or equal IPL, the interrupt remains pending.

The *interrupt dispatcher* is a combination of hardware and software that routes interrupts from devices on the UNIBUS, Q22 bus, or MASSBUS to the appropriate device driver's interrupt-servicing routine. The interrupt dispatcher's routing mechanism works differently depending upon whether the VAX processor in use accepts direct vector or nondirect vector I/O-bus interrupts.

3.1.5.1 Direct Vector Interrupts

The VAX–11/750, VAX–11/730, VAX 8200, and VAX 8800 processors employ direct vector UNIBUS adapters. The MicroVAX I and MicroVAX II also provide for direct vector interrupt dispatching from the Q22 bus. On a configuration that supports direct vector interrupts, the I/O adapter does not dispatch the interrupt. Instead, the processor locates the device's interrupt-servicing routine by using the system-control block (SCB).

The SCB consists of two or more pages of addresses. Page 1 lists the exception vectors; pages 2 and 3 contain the list of addresses in the channel-request block (CRB) that point to the interrupt-servicing routines for devices attached to the first UNIBUS and, for the VAX–11/750, an optional second UNIBUS. The SCB base register (SCBB), an internal processor register, marks the base of the SCB.

The processor obtains the vector address of the device that requested the interrupt,[2] and uses it as an index into page 2 (or page 3) of the SCB. The processor then transfers control to the interrupt-dispatching code in the device's CRB. On direct vector configurations, the interrupt-dispatching code saves registers R0 through R5 then transfers control to the device's interrupt-servicing routine.

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

3.1.5.2 Nondirect Vector Interrupts

The VAX–11/780 and VAX 8600 processors employ nondirect vector UNIBUS adapters. A device interrupt to a *nondirect vector adapter* causes the adapter to post an interrupt that is dispatched through the SCB to the interrupt-servicing routine for the UNIBUS adapter of the device that requested the interrupt.[3] It is the adapter's interrupt-servicing routine that ultimately locates and transfers control to the appropriate device driver's interrupt-servicing routine.

The UNIBUS adapter's interrupt-servicing routine performs the following actions:

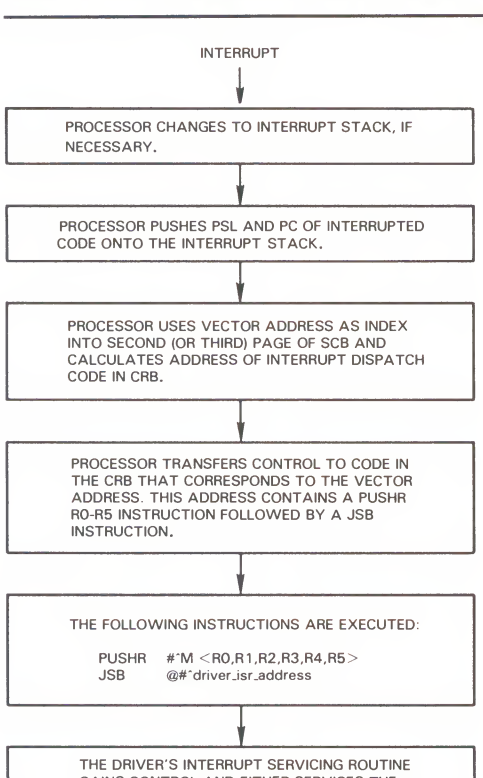
1. Saves R0 through R5 on the interrupt stack.
2. Reads a UNIBUS adapter register to determine the vector address of the device requesting the interrupt.
3. Uses the vector address as an index into a vector-jump table within the adapter-control block. The vector-jump table contains a list of addresses within CRBs that point to interrupt-servicing routines for devices attached to that UNIBUS.
4. Transfers control to the CRB address that corresponds to the vector address. The CRB address contains a JSB instruction that passes control to the device's interrupt-servicing routine.

Figure 3–4 contains a flowchart that illustrates nondirect vector interrupt dispatching.

[2] The vector addresses of direct vector interrupts can range from 0 to 777_8.

[3] The MASSBUS adapter is also a nondirect vector adapter. The MASSBUS adapter's interrupt dispatcher performs the functions described in Section G.4 before transferring control to the driver's interrupt-servicing routine.

Figure 3–3 Dispatching a Direct Vector Interrupt

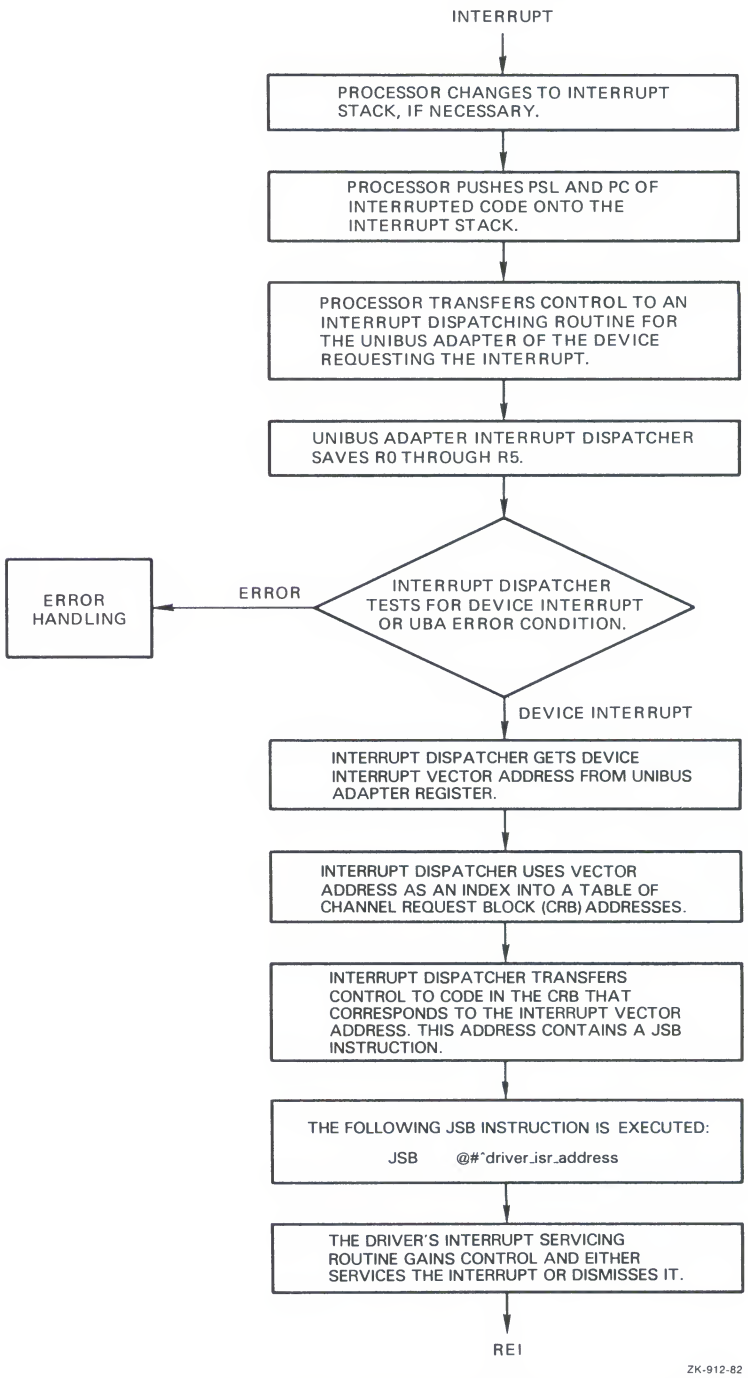


3.1.6 Transferring Control from the Device Interrupt to the Fork Process

When a device driver receives an expected interrupt from a device, the driver interrupt-servicing routine executes in the context of an interrupt; it is not executing in 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.

Figure 3–4 Dispatching a Nondirect Vector Interrupt



- 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 and status register (CSR).
- The stack also contains the 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. A driver's interrupt-servicing routine generally determines whether to service the interrupt by examining the I/O database. If the UCB for the device that currently owns the controller indicates that the interrupt is expected, the interrupt-servicing routine takes the following steps to transfer control to the driver's start-I/O routine:

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

The driver might need to execute a few instructions in the context of the interrupt. For example, the driver might copy device-status information from the device's registers into the device's UCB.

When a driver gains control, it might execute a few instructions at device 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 UCB to determine fork IPL for the driver
- Creates a driver's fork process and queues it for execution at the appropriate fork IPL
- Requests a software interrupt at that IPL

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

3.1.7 Modifying IPL in Driver Code

Code running in kernel mode can raise its IPL to lock out context switching and to block interrupts. VAX/VMS software interrupt-servicing 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-servicing routine that gains control at IPL 3; however, it raises IPL to 8 to read and modify the system database. Subsequent sections 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.

Driver code can change the IPL at which it executes by calling a VAX/VMS routine that raises or lowers IPL or invoking a VAX/VMS macro to request explicitly a change in IPL.

Normally, a driver uses the macros discussed in this section to raise IPL before initiating a transfer. Drivers typically raise IPL to check for a power failure, to 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 doing so blocks all interrupts at lower IPLs.

When lowering IPL, a driver either restores IPL to a previously-saved value or requests a software interrupt at a fork IPL at which it has queued a fork block (as described in Section 3.2). A driver cannot lower IPL below the level at which the thread of execution resumed.

The sections that follow describe the macros that drivers can call to change IPL:

- SETIPL
- DSBINT
- ENBINT
- SOFTINT

3.1.7.1 SETIPL Macro

The SETIPL macro moves the specified IPL into the processor IPL register (PR$_IPL).

Format

```
SETIPL  [ipl=31]
```

Argument

[ipl=31]

Interrupt priority level. If no value is specified in the **ipl** argument, the SETIPL macro moves the value 31 into PR$_IPL. Setting IPL to 31 blocks all interrupts.

3.1.7.2 DSBINT Macro

The DSBINT macro saves the current IPL in the specified destination and moves the specified IPL into the processor IPL register (PR$_IPL). Procedures invoke this macro to raise IPL.

Format

```
DSBINT  [ipl=31] [,dst=-(SP)]
```

Arguments

[ipl=31]

Interrupt priority level. If no value is specified in the **ipl** argument, DSBINT moves the value 31 into PR$_IPL, thus blocking all interrupts.

[dst=–(SP)]

Location at which the current IPL is to be saved. If no value is specified in the **dst** argument, DSBINT stores the current IPL on the top of the stack.

3.1.7.3 ENBINT Macro

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

Format

```
ENBINT  [src=(SP)+]
```

Argument

[src=(SP)+]
Location containing the IPL to be restored. If no value is specified in the **src** argument, ENBINT moves the value on the top of the stack into the PR$_IPL.

3.1.7.4 SOFTINT Macro

The SOFTINT macro moves the specified IPL into the software interrupt request processor register (PR$_SIRR) 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), it immediately transfers control to a software interrupt-servicing routine for the appropriate IPL.

If the processor is executing at or above the specified IPL, it does not transfer control to the software interrupt-servicing routine until IPL drops below the specified IPL.

Format

```
SOFTINT  ipl
```

Argument

ipl
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 might be suspended to wait for a device interrupt or shared resources. To ensure that the resulting context-switching is fast, the VAX/VMS operating system forces driver routines to execute in a minimal, fork process context consisting of a device's UCB, called a fork block, and a few registers.

Fork processes are created in either of the following situations:

- Once the preprocessing of an IRP 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 IRP for the driver to process later.
- Either the driver's interrupt-servicing routine or the driver postprocessing routine creates a fork process to perform device-dependent I/O postprocessing.

When the system creates a fork process to execute the start-I/O routine, the newly created fork process can execute immediately because the IRP has been preprocessed by the $QIO system service and driver's FDT routines, and because the device is idle.

When the driver's interrupt-servicing routine or the driver's postprocessing routine creates a fork process, it does so to lower the IPL at which the driver's code is executing. Either the interrupt-servicing routine or the start-I/O routine 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-Servicing Routine for Fork Dispatching

One interrupt-servicing 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.

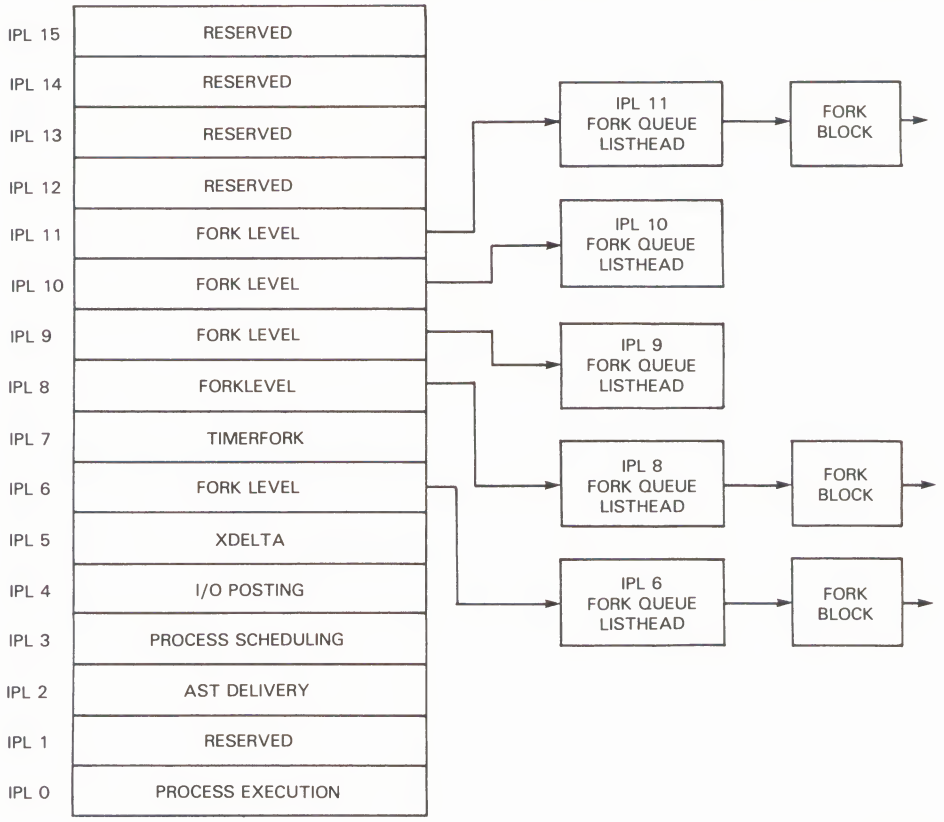
When that fork process is completed, the dispatcher regains control, removes the next entry from the queue, restores its fork process context, and reactivates it. This sequence is repeated until the fork queue is empty. When the queue is empty, the fork dispatcher restores R0 through R5 from the stack and dismisses the interrupt with an REI instruction.

Figure 3–5 illustrates the fork queue structure.

A newly activated 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 the driver's fork IPL and IPL$_POWER; it must not lower IPL below the driver's fork IPL 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 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 Queue Structure



3.3 Resource-Wait Queues

The processing of an I/O request often requires shared system resources such as memory and I/O adapter mapping registers. The $QIO system service and fork processes call VAX/VMS routines to allocate and deallocate these resources. Because the resources are limited, I/O processing might 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 the processing of I/O requests.

For example, the $QIO system service calls a VAX/VMS routine to allocate nonpaged system space for an IRP. 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). 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 fork process execution at elevated 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 database. The driver needs only the address of the device's UCB, which is the key to the rest of the database. Therefore, VAX/VMS routines that control driver resources, such as mapping registers,

use 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 fork process; the UCB also contains the driver's 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 in a waiting state any driver that requests an unavailable resource, drivers are unaware of execution being suspended and subsequently reactivated. Drivers must not leave anything on the stack, or in general purpose registers other than R3, R4, and R5, when calling a routine that might suspend the driver's execution.

3.3.1 Competing for a Controller's Data Channel

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

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

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

1. Retrieve the virtual address of the data to be written and the number of bytes to transfer from the device's UCB
2. Retrieve the virtual address of the device's CSR from the IDB
3. Calculate the address of the line printer's data buffer register by adding a constant offset to the CSR address
4. 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's 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's fork process must gain control of the controller's 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:

1. The start-I/O routine requests the controller's data channel by invoking a VAX/VMS channel arbitration routine.
2. The VAX/VMS routine tests the CRB mask field to determine whether the controller's data channel is available.
3. If the channel is available, the VAX/VMS routine allocates the channel to the fork process and returns the address of the device's CSR 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's channel-wait queue.
4. When the fork process resumes execution, the process owns the controller channel. The fork process can then modify the device's registers to activate the device.
5. The driver's start-I/O routine then requests the VAX/VMS operating system to suspend driver processing in anticipation of an interrupt or timeout and to release the channel.
6. The VAX/VMS channel-releasing routine assigns channel ownership to the next fork process in the channel-wait queue, loads the CSR address into a general register, and reactivates the suspended fork process.
7. 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 CRB. The fork process must always request and release the controller's data channel by invoking these routines. Once the driver owns a controller's data channel, the driver is free to read and modify the device's registers.

4 I/O Adapter Functions

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's registers using system virtual addresses
- Translates 18-bit UNIBUS addresses to physical addresses in main memory
- Provides a data-transfer path to randomly ordered physical pages in main memory
- Provides buffered data transfer paths to consecutively increasing physical addresses, thus optimizing CPU-to-UNIBUS data transfers.
- Permits byte-aligned buffers for UNIBUS devices requiring word-aligned buffer addresses

The MicroVAX Q22 bus closely resembles the UNIBUS. For MicroVAX II or MicroVAX I devices attached to the Q22 bus, special processor logic implements a Q22 bus interface that similarly allows drivers access to device registers and manages device interrupts. Additional logic in the MicroVAX II processor establishes a scatter-gather map that translates 22-bit Q22 bus addresses to physical addresses. However, neither MicroVAX II nor MicroVAX I implements buffered data paths. (Table 4–1 compares the UNIBUS and Q22 bus systems of the various VAX and MicroVAX processors.)

The protocol a VAX processor uses to enable communications between its I/O bus and backplane permits its devices and device drivers to exchange data without much awareness of the intervening hardware. First of all, both the UNIBUS adapter and the Q22 bus interface provide access to device registers using an address mapping scheme that is invisible to the driver. In addition, whenever the configuration of the I/O interface has an impact on the control of a data transfer, the driver can call one of the many VAX/VMS routines that handles the details of the interface.

The functional differences between I/O adapters are irrelevant to devices that do not perform DMA transfers. A driver that performs non-DMA transfers for a device on the UNIBUS can, with no alteration, perform the same services for an equivalent device on a Q22 bus.

On the other hand, the differences between the functions of the UNIBUS adapter and the interfaces provided by the MicroVAX II and I to the Q22 bus are significant to those drivers that manage DMA device operations.[1] A driver that performs block DMA transfers for a UNIBUS device or Q22 bus device must set up any mapping or buffering mechanisms required by the processor's I/O interface. For UNIBUS DMA drivers, this involves setting

[1] The Q22 bus supports only those DMA controllers that are capable of 22-bit addressing.

up sufficient mapping registers and, perhaps, a buffered data path prior to the transfer. MicroVAX II DMA drivers, likewise, must allocate and fill a set of mapping registers. By contrast, MicroVAX I DMA drivers—because the MicroVAX I has no scatter-gather map—cannot map the many and scattered pages of a block DMA transfer to a contiguous set of addresses in the I/O adapter's address space. As a result, when it is loaded into the system, a MicroVAX I DMA driver must reserve enough physically contiguous memory to accommodate its largest possible DMA transfer (see Section 10.7).

Section 10 describes the means by which device drivers set up DMA transfers, according to any of these interfaces. If DMA driver that must drive similar devices on various VAX processors must secure some measure of machine-independence, it can include some run-time conditional code that branches to appropriate routines in the driver that accomplish the machine-dependent work. See the description of the CPUDISP macro in Appendix B and the sample drivers that appear in Appendixes E and F for guidance.

This following sections discuss the functions of the UNIBUS adapter and similar Q22 bus interface functions:

- The discussion of reading and writing device registers in Section 4.1 applies to UNIBUS, MicroVAX II, and MicroVAX I drivers.
- The description of mapping I/O bus addresses in Section 4.2 pertains only to UNIBUS and MicroVAX II DMA drivers.
- The description of buffering data transfers in Section 4.3 relates in the main to UNIBUS drivers, although the section on direct data paths (Section 4.3.1) contains information relevant to MicroVAX II and MicroVAX I drivers as well.

Table 4–1 Features of the I/O Bus Adapters of the VAX Processors

Processor	Adapter	Memory References (Physical Address)	Direct Data Path	Buffered Data Paths	Mapping Registers	Interrupt Dispatcher
VAX–11/780 VAX–11/782 VAX–11/785 VAX 8600 VAX 8650	UBA	30-bit (via SBI)	1, no byte-aligned transfers	15, 8-byte buffer, byte-aligned transfers, LWAE,[3] prefetch	496	Nondirect vector
VAX–11/750	UBI	24-bit (via CMI)	1, byte-aligned transfers	3, 4-byte buffer,[2] byte-aligned transfers, LWAE,[3] no prefetch	512[4]	Direct vector

[2]Buffered data paths on the VAX–11/750 only buffer four bytes of data. Because the data paths do not perform a prefetch, they can always reference longwords at random.

[3]LWAE (longword access enable) refers to the capability to reference random longword aligned data in a bus transfer.

[4]The VAX/VMS operating system makes only 496 of these mapping registers available.

Table 4–1 (Cont.) Features of the I/O Bus Adapters of the VAX Processors

Processor	Adapter	Memory References (Physical Address)	Direct Data Path	Buffered Data Paths	Mapping Registers	Interrupt Dispatcher
VAX–11/730 VAX–11/725	UBA	24-bit	1, byte-aligned transfers	None	512[4]	Direct vector
VAX 8200 VAX 8800	BUA	30-bit (via VAXBI)	1, byte-aligned transfers	5, 8-byte buffer, byte-aligned transfers, LWAE,[3] no prefetch	512[4]	Direct vector
MicroVAX I	—	22-bit	1, no restrictions on data alignment[1]	None	None	Direct vector
MicroVAX II	—	24-bit	1, no restrictions on data alignment[1]	None	8192[4]	Direct vector

[1]The MicroVAX II and MicroVAX I implementations of the Q22 bus provide no byte-offset register, so, on Q22 bus devices that are only capable of word-aligned transfers, only word-aligned transfers are possible.

[3]LWAE (longword access enable) refers to the capability to reference random longword aligned data in a bus transfer.

[4]The VAX/VMS operating system makes only 496 of these mapping registers available.

4.1 Reading and Writing Device Registers

Each I/O controller or device directly attached to a UNIBUS or Q22 bus has a control and status register (CSR) and set of data registers. These registers are assigned physical addresses in the 8K allocated for this purpose from the 256K UNIBUS address space or from the Q22 bus I/O space. Device drivers obtain the device's status and activate the device by reading and writing to these registers.

Because the VAX/VMS operating system maps this I/O space into virtual address space, 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's register as though the device's register were a location in memory. The driver must use instructions within the restrictions described in Section 6.2.

Before a driver for a device that shares a controller can gain access to a device's registers, it must first obtain a controller channel, as described in Sections 3.3.1 and 9.3.1.

4.2 Mapping Registers

DMA devices read and write data from and to memory locations using 18-bit UNIBUS addresses or, for the MicroVAX II and MicroVAX I, 22-bit Q22 bus addresses. The UNIBUS adapter and the Q22 bus interface translate the bus addresses into main memory addresses, thus allowing the operating system, I/O drivers, and UNIBUS devices to access the same physical address space. DMA devices connected to either a UNIBUS or MicroVAX II Q22 bus can access a block of memory directly by means of the scatter-gather map supplied by the UNIBUS adapter or MicroVAX II processor, respectively. The mapping registers provided allow the device to access scattered, physical memory addresses as contiguous, physical addresses in I/O space.[2]

When a device driver performs a DMA transfer, it allocates mapping registers and a buffered data path (an option available to devices on the UNIBUS of some VAX processors), and sets up the transfer by means of the device's registers. The device then accesses memory directly by means of the I/O bus, transferring all the data requested. When the transfer is complete, the device notifies the driver by requesting an interrupt.

Consider a buffer, for example, that consists of virtual pages 400, 401, 402, and 403, which are physical pages 1003, 204, 1190, and 240, respectively. For a UNIBUS device to access this buffer, the driver requests four mapping registers, then places the physical addresses of these pages in the mapping registers. Assume the driver has allocated four mapping registers, 127 through 130. The driver loads them as follows:

Mapping Register	Contents (physical address)
127	1003
128	204
129	1190
130	240

The device and the UNIBUS can transfer data into or out of these physical pages without intervention by the driver. The device requests an interrupt only when all the data in these four pages has been transferred.

Generally, a mapping register exists for each page of I/O space. Because the UNIBUS address space consists of 256K of memory, minus the 8K reserved for device-control registers, 496 mapping registers are available for UNIBUS DMA transfers. MicroVAX II DMA devices can also use up to 496 of the mapping registers corresponding to 248K of the 4MB Q22 bus I/O space.[3]

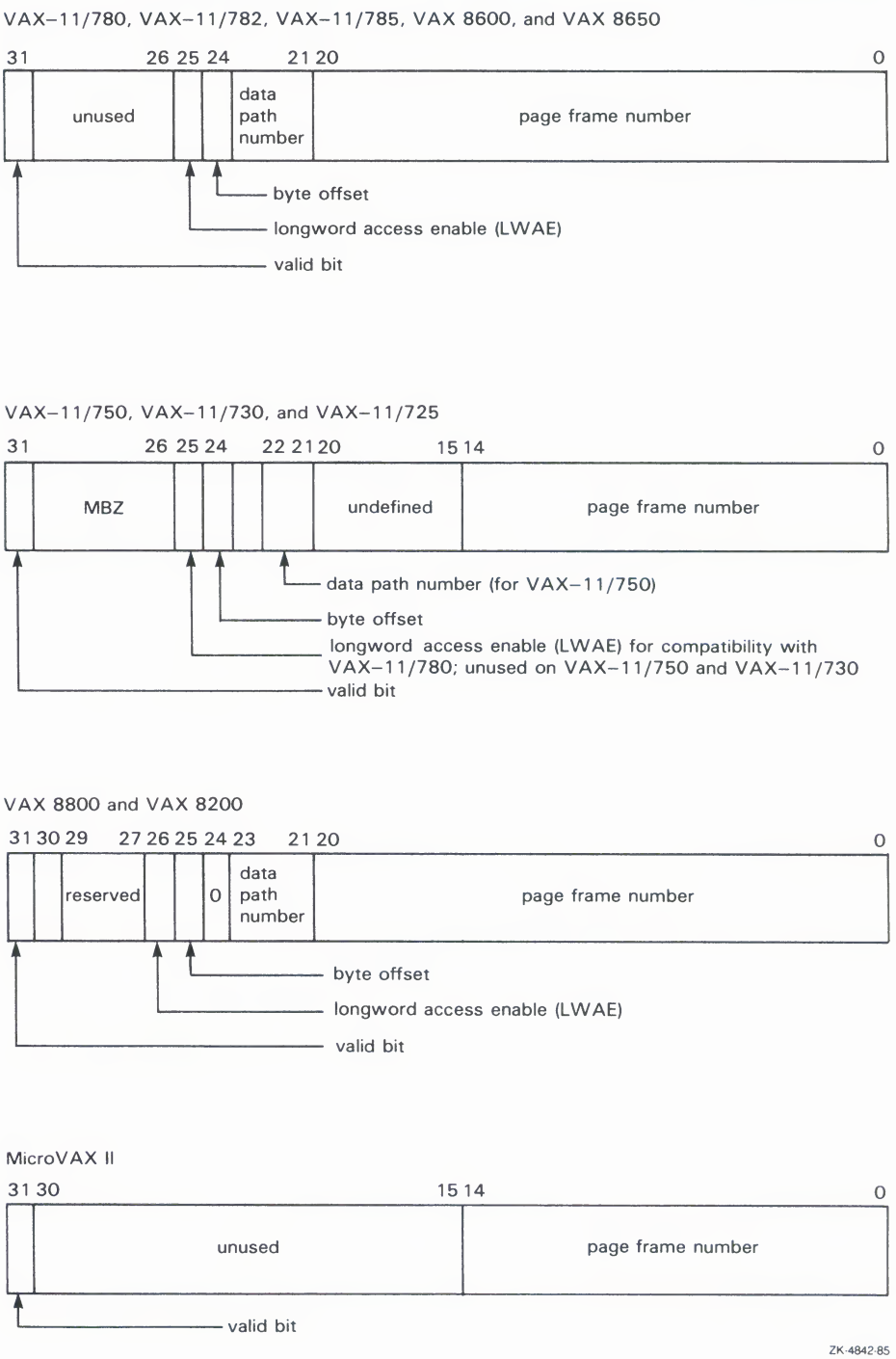
---

[2] The MicroVAX I does not provide a scatter-gather map, thereby requiring its DMA drivers to reserve a contiguous portion of physical memory in its controller-initialization routine to provide for its largest possible DMA transfer. See the discussion in Section 10.7 for details.

[3] There are actually 8192 mapping registers which correspond to Q22 bus I/O space. To maintain compatibility with software that accesses UNIBUS adapter mapping registers, only 496 of these registers are currently enabled. The remaining registers are unavailable for driver use. A field in the mapping register identifies the page-frame number corresponding to the UNIBUS space or Q22 bus space address that the mapping register represents (see Figure 4–1).

If a driver must explicitly access memory local to the MicroVAX II Q22 bus, it cannot access it by means of mapping registers. Instead, it must first map the desired region into system space using the Create and Map Section ($CRMPSC) system service and specifying its PFNMAP option (see Section H.3 and the *VAX/VMS System Services Reference Manual* for additional information). In addition, it must disable those mapping registers that correspond to the Q22 bus addresses for this memory.

Figure 4–1 UNIBUS and Q22 Bus Mapping Registers



Drivers call VAX/VMS routines to fill as many mapping registers with valid page-frame addresses as needed for a DMA transfer up to 127 pages. The DMA device puts an address on the I/O bus. The UNIBUS adapter or Q22 bus interface receives the address and translates it using the following information (see Figures 4–2 and 4–3)[4] :

- In *UNIBUS addresses,* the 9-bit UNIBUS page address field (bits 9 through 17 of the UNIBUS address) identifies the UBA mapping register.

  In *Q22 bus addresses,* the 13-bit Q22 bus page address field (bits 9 through 22 of the Q22 bus address) identifies the MicroVAX II mapping register.

- The page-frame-number (PFN) field in the mapping register specifies the high-order bits of the physical address. (The PFN field is 15 bits long for the MicroVAX II, VAX–11/750, and VAX–11/730; and 21 bits long for the VAX–11/780, VAX 8600, VAX 8200, and VAX 8800.)

- From *UNIBUS addresses,* bits 2 through 8 map 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.

  From *Q22 bus addresses,* bits 0 through 8 map to bits 0 through 8 of the physical address. The resulting physical address locates the byte that is the target of the transfer.

Each UNIBUS adapter or Q22 bus mapping register also contains a bit called the mapping-register valid bit. The UNIBUS adapter or Q22 bus interface tests this bit every time the mapping register is used. If the bit is not set, the UNIBUS adapter or Q22 bus interface aborts the transfer. This bit is clear whenever the register is not mapped to a physical address.

4.3 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 for each 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.

When a UNIBUS device begins a DMA transfer by placing an address on the UNIBUS, the UNIBUS adapter mapping register not only performs address mapping but also provides the number of the data path to be used for the transfer (see Figure 4–1). Each UNIBUS adapter mapping 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 data path registers of the various VAX processors are pictured in Figure 4–4.)

[4] The page-frame address is 15 bits long on the VAX–11/750, VAX–11/730, VAX–11/725, and MicroVAX II processors; the physical addresses resulting from the mapping are 24 bits long. The page-frame addresses of other VAX processors are each 21 bits long, with a resulting 30-bit physical address. The disposition of the lowest two bits of the UNIBUS address depends on the processor. For instance, the VAX–11/780 uses them to construct a byte-selection mask and function to be transmitted across UNIBUS lines that modify the I/O transaction.

Figure 4–2 Mapping a UNIBUS Address to a Physical Address

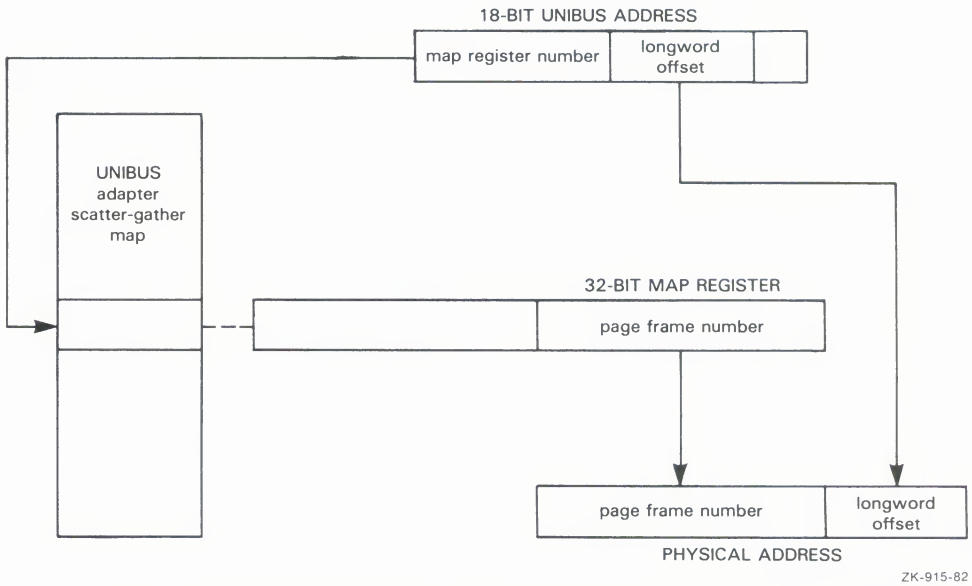
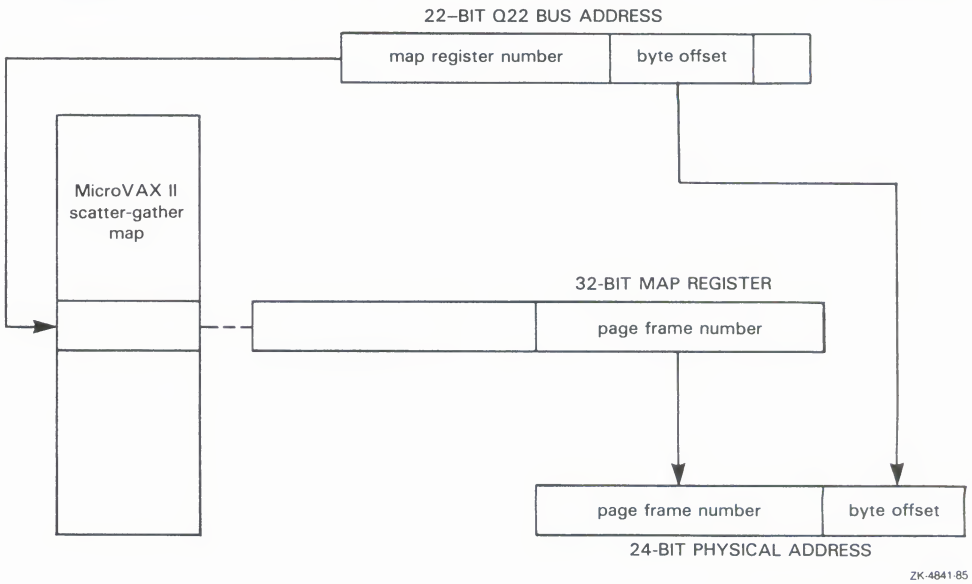


Figure 4–3 Mapping a Q22 Bus Address to a Physical Address

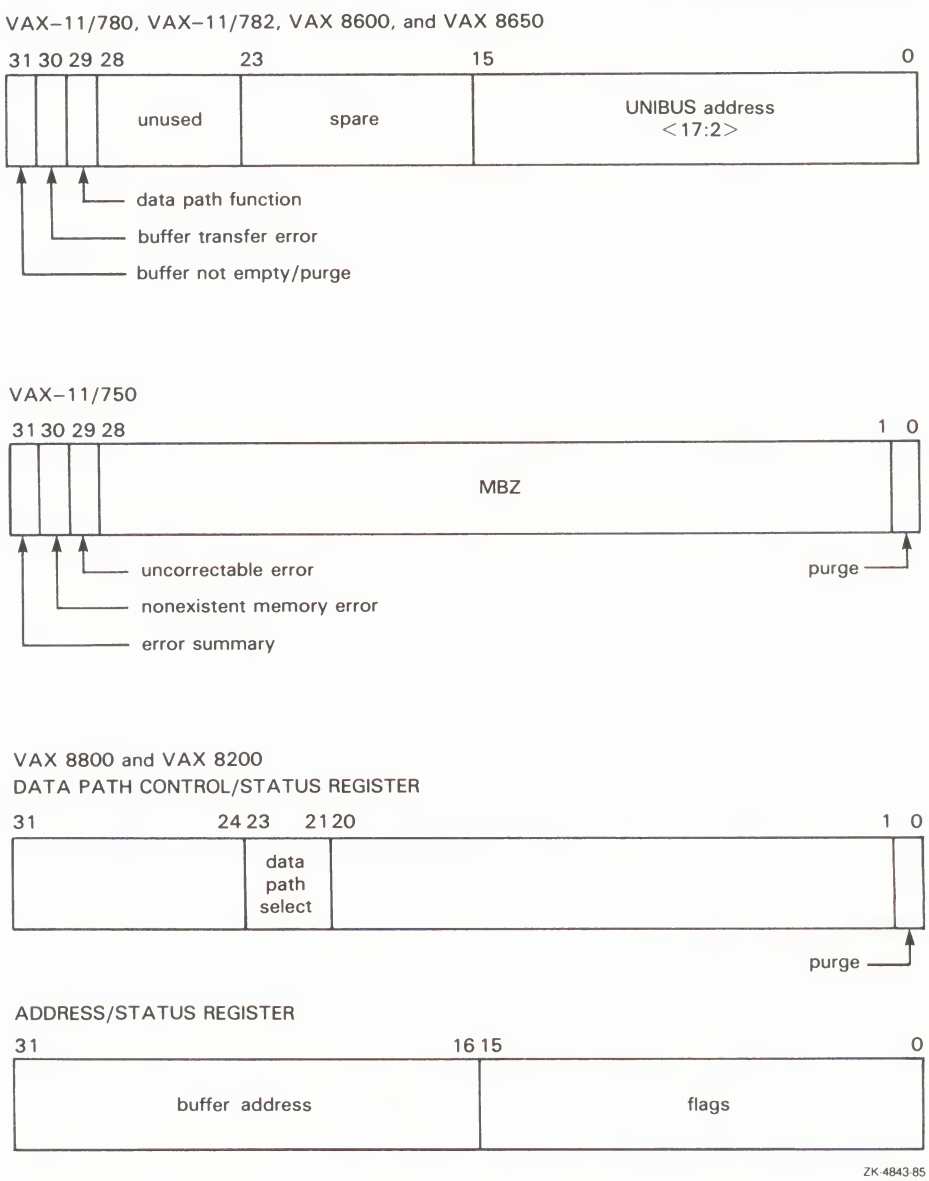


The sequence below describes a UNIBUS-device DMA transfer.

1 The UNIBUS device puts an address on the UNIBUS.

2 The UNIBUS adapter locates the UNIBUS adapter mapping register that corresponds to the UNIBUS address.

Figure 4–4 UNIBUS Data Path Registers



3 The UNIBUS adapter verifies that the mapping register has the mapping-register valid bit set.

4 The UNIBUS adapter maps the UNIBUS address to a page-frame number.

5 The UNIBUS adapter extracts the number of the data path to be used for the transfer from the mapping register.

6 The data path translates the UNIBUS function to a backplane interconnect function by reading the UNIBUS control lines.

7 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.

4.3.1 Direct Data Path

Since the direct data path performs a backplane interconnect transfer for every UNIBUS transfer, it 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 for each backplane interconnect cycle. On some hardware configurations, the direct data path is unable to transfer a word of data to an odd-numbered 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.[5]

A UNIBUS device may choose to use a direct data path rather than a buffered data path 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 does not need to call the VAX/VMS routine that allocates the data path. It performs the following actions:

1 Uses the REQMPR macro to allocate a set of mapping registers

2 Uses the LOADUBA macro to load the mapping registers with physical address mapping data and the number of the direct data path (0). The VAX/VMS routine called in the expansion of the LOADUBA macro (IOC$LOADUBAMAP) also sets the valid bit in every mapping register except the last, which remains invalid to prevent a wild transfer.

3 Loads the starting address of the transfer in a device register.

4 Loads the transfer byte or word count in a device register.

5 Sets bits in the device control register to initiate the transfer.

[5] The MicroVAX II and MicroVAX I implementations of the Q22 bus provide no byte-offset register. As a result, on Q22 bus devices that are only capable of word-aligned transfers, only word-aligned transfers are possible.

4.3.2 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.

A UNIBUS device may choose to use a buffered data path rather than a direct data path to perform the following functions:

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

A single 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, it performs a sequence of steps similar to those performed by a driver that uses the direct data path, with the exception that it uses a macro that calls a VAX/VMS routine that allocates a free buffered data path. The following are among the actions of the driver fork process:

1. Uses the REQMPR macro to allocate a set of mapping registers.
2. Uses the REQDPR macro to allocate a free buffered data path.
3. Uses the LOADUBA macro to load the mapping registers with physical address mapping data and the number of the allocated buffered data path. The VAX/VMS routine called in the expansion of the LOADUBA macro (IOC$LOADUBAMAP) also sets the valid bit in every mapping register except the last, which remains invalid to prevent a wild transfer.
4. Load the starting address of the transfer in a device register.
5. Load the transfer byte or word count in a device register.
6. Set bits in the device control register to initiate the transfer.

The UNIBUS adapter hardware of certain processors restricts normal buffered data paths to referring only to consecutively increasing addresses. Through a special mode of operation, these UNIBUS adapters can also refer to 32-bit data at randomly-ordered, longword-aligned locations in physical memory. Other processors do not impose this restriction. In order for a device driver to run on both types of processors, it must observe three rules:

- All transfers within a block must be of the same function type (DATI or DATO/DATOB).
- Normal buffered data paths must always transfer data to consecutively increasing addresses.
- To reference 32-bit data at random, longword-aligned locations in physical memory, the longword-access-enable bit (LWAE) must be set.

A buffered data path stores data from the UNIBUS in a buffer until multiple words of data have been transferred (except in longword-aligned, 32-bit, random-access mode as discussed in Section 4.3.5). 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 from a device to memory is broken into individual steps.

1. The UNIBUS device transfers one word of data to the buffered data path.
2. The buffered data path stores the word of data and completes the UNIBUS cycle.
3. The buffered data path sets its buffer-not-empty flag to indicate that the buffer contains valid data.
4. The UNIBUS device repeats the first three steps until the buffer is full.
5. When the UNIBUS device addresses the last byte or word in the buffer, the UNIBUS adapter recognizes a complete data-gathering cycle.
6. The buffered data path requests a backplane-interconnect-write function to write the data from the buffered data path to memory.
7. 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 operation that transfers data from main memory to a device varies according to the type of UNIBUS adapter. Those adapters that can perform a prefetch function complete UNIBUS reads from memory more quickly than those that cannot. The prefetch feature accomplishes this improved performance by automatically filling the data path buffer after the buffer's contents are transferred to the UNIBUS.

The following paragraphs discuss the UNIBUS read operation with and without the prefetch function. Device drivers that adhere to the conventions outlined in this manual will execute properly whether or not the device is associated with a UNIBUS adapter that provides prefetch functionality.

1. The UNIBUS device initiates a read operation from a buffered data path.
2. The buffered data path checks to see if its buffers contain valid data.
3. If the buffers do not contain valid data, the buffered data path initiates a read function to fill the buffers with data from main memory. The transfer completes before the UNIBUS adapter begins a UNIBUS transfer.
4. 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.
5. The buffered data path sets its buffer-not-empty flag to indicate that the buffers contain valid data.
6. 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 buffer-not-empty flag to indicate that the buffer no longer contains valid data.
7. The buffered data path then initiates a read function to prefetch data from memory.
8. When the prefetch is complete, the buffered data path sets the buffer-not-empty flag to indicate that the buffers now contain valid data.

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

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

1. The UNIBUS device initiates a read operation from a buffered data path.
2. The buffered data path checks to see if its buffers contain valid data.
3. 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.
4. 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.

4.3.3 Byte-Offset Data Transfers

The UNIBUS adapter has a byte-offset register; thus, words that are not word-aligned can be transferred to and from any device on the UNIBUS regardless of whether the device supports non-word-aligned 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 mapping registers indicates byte-aligned data to the hardware. If the bit is set, the hardware increments physical addresses. A VAX/VMS subroutine that loads mapping registers determines whether the data is word- or byte-aligned and sets the byte-offset bit accordingly.

4.3.4 Purging a Buffered Data Path

Because prefetches can 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 performs two functions:

- 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 memory, then clears the flag.
- Notifies the driver's fork process of any error that occurs during the purge.

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

4.3.5 Longword-Aligned, 32-Bit, Random-Access Mode

Another method of transferring data over a buffered data path is the use of longword-aligned, 32-bit, random-access mode. This mode essentially prevents the UNIBUS prefetch operation, thereby allowing a device that reads data from or writes data to memory to reference longword-aligned locations in memory at random, in longword multiples.

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

For a UNIBUS read operation that transfers data from main memory to a device, the function occurs as follows:

1. The driver's fork process initiates a read function on the UNIBUS device.
2. The UNIBUS adapter clears the buffer-not-empty flag in the assigned buffered data path.
3. The UNIBUS adapter requests a read-from-memory operation on the backplane interconnect.
4. The UNIBUS adapter stores the longword of data in the buffered data path and sets the buffer-not-empty flag.
5. The UNIBUS adapter initiates two UNIBUS read operations to transfer two words of data.

For a UNIBUS write operation that transfers data from a device to main memory, the function occurs as follows:

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

To ensure that random-access mode works correctly regardless of processor type, a buffered data path should not repeatedly address the same longword. 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 transfer over a direct data path and somewhat slower than a normal transfer using a buffered data path.

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 ($QIO) system service gains control and coordinates preprocessing of the request. The last driver FDT routine called by the $QIO system service calls a VAX/VMS routine that creates a driver fork process to execute the driver's start-I/O routine. This routine activates the device.

When the transfer is completed, the device requests an interrupt that results in execution of the driver's interrupt-servicing 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.

The $QIO system service is dispatched by means of a corresponding system service vector in process P1 space. This vector essentially contains a CHMK instruction that causes an exception which alters the process' access mode to kernel and dispatches to the service-specific procedure, EXE$QIO. VAX/VMS system service dispatching is described in detail in the *VAX/VMS Internals and Data Structures* manual. For the purposes of the discussion in this section, as well as the rest of the book, Figure 5–2 portrays the flow of an I/O request from its system service entry point to its servicing by VAX/VMS executive routines and driver code. Discussion of other entry points appears in Sections 9, 11, and 12.

5.1 Preprocessing an I/O Request

EXE$QIO performs device-independent preprocessing of an I/O request and calls driver FDT routines to perform device-dependent preprocessing. To preprocess an I/O request, EXE$QIO 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 database
- 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 (IRP)
- Calls driver FDT routines to perform device-dependent preprocessing

Figure 5–1 Sequence of Driver Execution

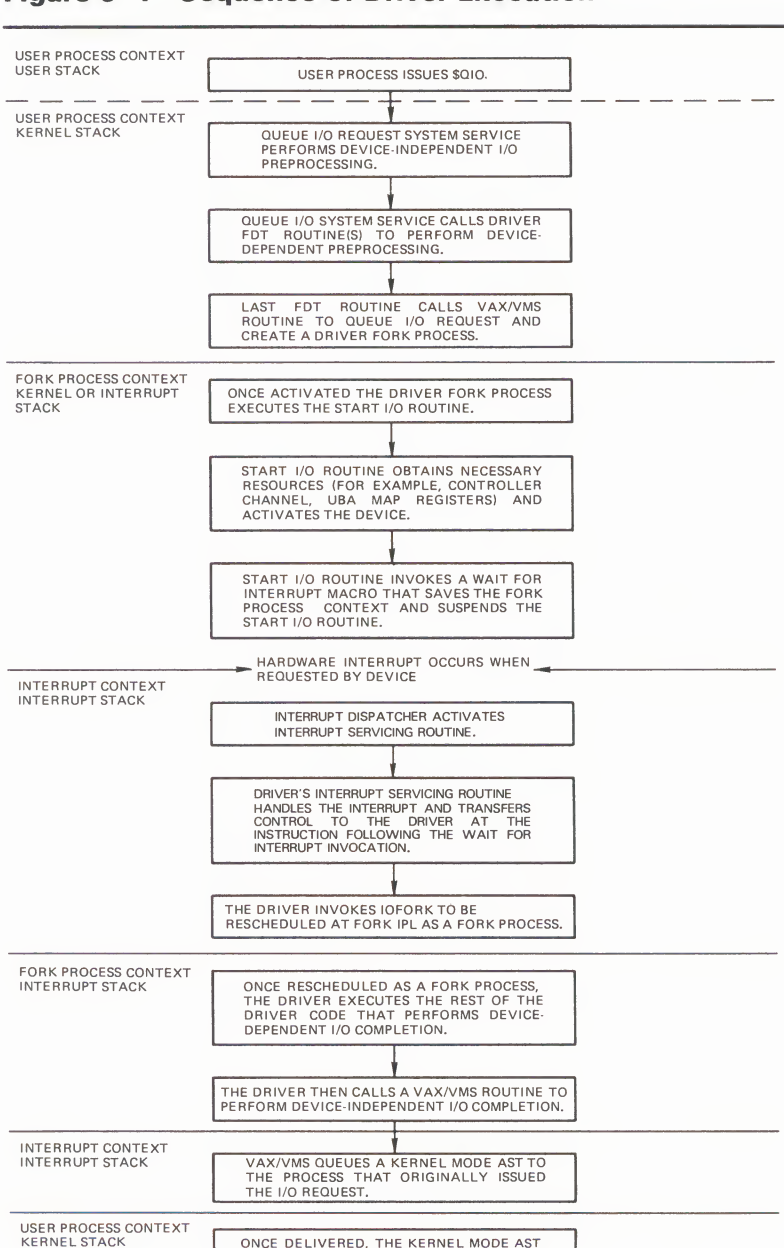
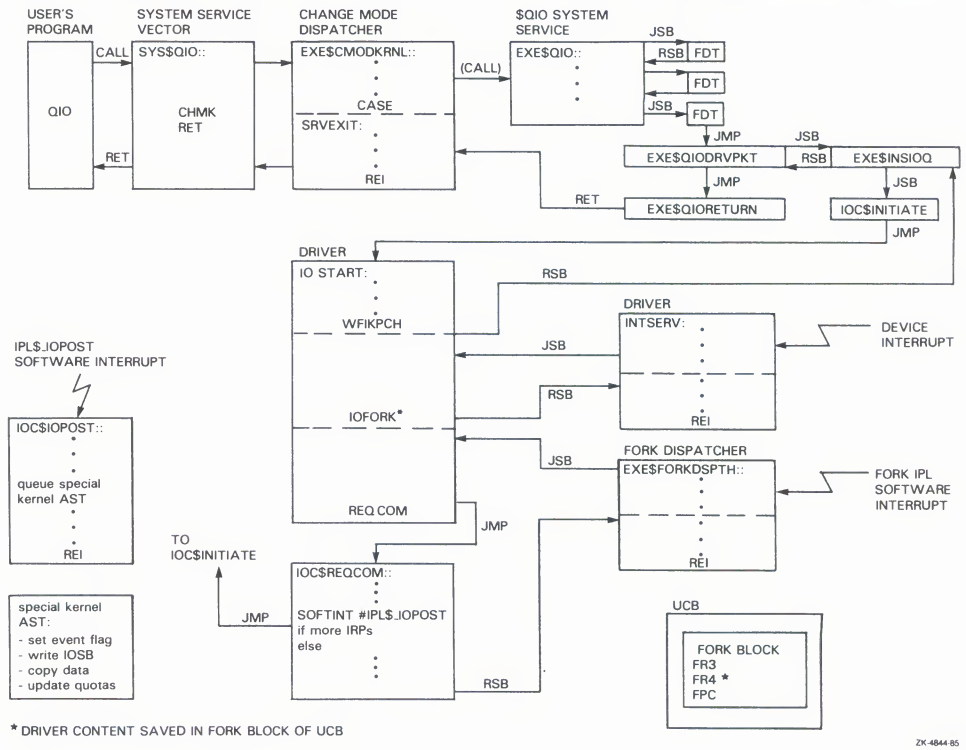


Figure 5–2 Detailed Sequence of VAX/VMS I/O Processing



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 ($ASSIGN) system service call. The $ASSIGN 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, EXE$QIO verifies that the channel number specified is associated with a device and locates the unit-control block associated with the specified channel using the field CCB$L_UCB.

Refer to Figure A-3 and Table A-3 for an illustration of the channel-control block and a description of its contents.

5.1.2 Locating a Device Driver in the I/O Database

A unit-control block (UCB) that describes a device unit exists for each device in the system. The UCB indicates the current state of the device unit by recording such information as:

- Whether the device is active (UCB$V_BSY in UCB$L_STS)
- What I/O request is being processed (UCB$L_IRP)
- Where transfer buffers are located (UCB$L_SVAPTE)

Because drivers run as fork processes and cannot use process address space to store additional context, drivers use the UCB for temporary data storage during I/O processing.[1]

The UCB 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.

Using information in the UCB, a driver can find other I/O data structures associated with the device, including the channel-request block, interrupt-dispatch block, and the device-data block.

Figure A-13 represents a UCB and Table A-13 describes its fields.

5.1.2.1 Channel-Request Block

The channel-request block (CRB) allows the operating system to manage the controller data channel. Among its contents are:

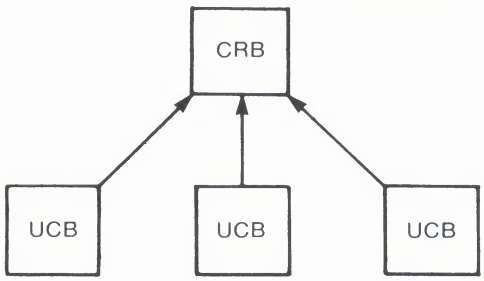
- Code that transfers control to a driver's interrupt-servicing routine (CRB$L_INTD)
- Addresses of a driver's unit and controller initialization routines (CRB$L_INTD+VEC$L_UNITINIT, CRB$L_INTD+VEC$L_INITIAL)
- A pointer to the interrupt-dispatch block (IDB), which further describes the controller (CRB$L_INTD+VEC$L_IDB)

Controllers can be either multiunit or dedicated.

All UCBs describing device units attached to a single *multiunit controller* contain a pointer to a single CRB (UCB$L_CRB). For these controllers, a VAX/VMS routine uses fields in the CRB (CRBL_WQFL, CRBB_MASK) and IDB (IDB$L_OWNER) 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. Figure 5–3 illustrates the data structures required to describe three devices on a multiunit controller.

[1] Section 7.1 describes how you can allocate additional UCB space for storing data or device-dependent driver context. The template in Section 6.4 and the macro descriptions in Appendix B demonstrate how you can define driver-specific fields in a UCB extended in this manner using the $DEFINI, $DEF, $DEFEND, $VIELD, and _VIELD macros.

Figure 5–3 Data Structures for Three Devices on One Controller



The VAX/VMS operating system does not use the CRB to synchronize I/O operations for a *dedicated controller,* as the controller manages but a single device. Nevertheless, the CRB still is present and used by drivers and operating system routines.

See Figure A-4 and Table A-4 for an illustration of the CRB and a description of its contents.

5.1.2.2 Interrupt-Dispatch Block

The CRB contains a pointer to an interrupt-dispatch block (IDB) (CRB$L_INTD+VEC$L_IDB). The IDB contains the addresses of these three critical data structures:

- The UCB of the device unit, if any, that currently owns the controller data channel (IDB$L_OWNER)
- The control and status register (IDB$L_CSR); it is the key to access to device registers
- The adapter-control block (IDB$L_ADP) that describes the adapter of the I/O bus to which the controller is attached

A detailed description of the fields in the IDB appears in Table A-9; Figure A-9 shows its structure.

Figure 5–4 illustrates the relationship between the data structures that describe a group of equivalent devices on two separate controllers. In this figure, one controller has a single device unit, and the other controller has two device units. Devices on both controllers share the same driver code.

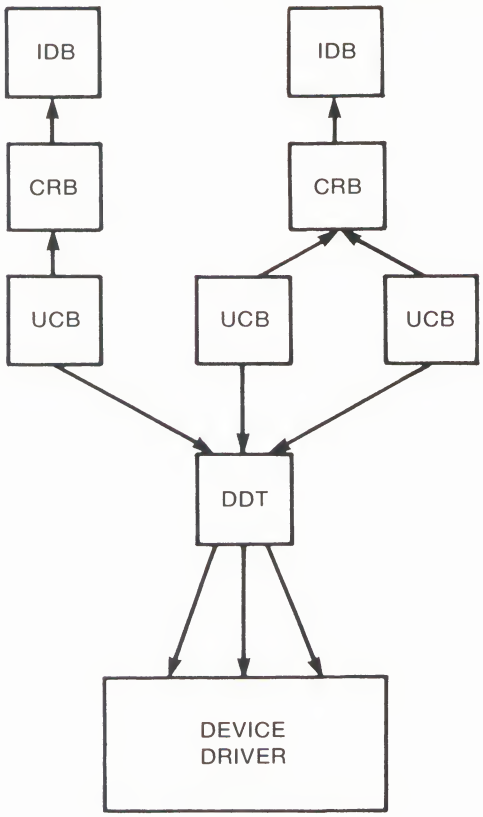
5.1.2.3 Device-Data Block

All UCBs describing device units attached to a single controller contain a pointer (UCB$L_DDB) to a single device-data block (DDB). The DDB contains two fields that identify the device and its driver:

- The generic device/controller name (DDB$T_NAME)
- The name of the device's driver as obtained from the driver-prologue table (DDB$T_DRVNAME).

Table A-6 further describes the fields of the DDB. For a representation of its structure, see Figure A-6.

Figure 5–4 I/O Database for Two Controllers



ZK-1765-84

5.1.3 Validating the I/O Function

Using the I/O data structures described above, EXE$QIO locates the address of the driver's function-decision table by following a chain of pointers that begins in the UCB of the target device:

UCB → DDT → FDT

EXE$QIO then uses data in the function-decision table to analyze the I/O function. The procedure 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

EXE$QIO determines whether the I/O request being readied will cause the process to exceed its quota for outstanding direct or buffered I/O requests. If the process' requests remain under quota, the system service allows it to continue I/O preprocessing. In the case where quota is exceeded, the procedure examines the process' resource wait flag (PCB$V_SSRWAIT in PCB$L_STS).

If the flag is clear, EXE$QIO aborts the I/O request. However, if the flag is set, it places the process in a wait state until the number of requests drops below quota. When this occurs, process execution resumes, at which time EXE$QIO charges 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, EXE$QIO determines whether the process issuing the request has write access to the status-block locations specified. If the process has write access, EXE$QIO fills the quadword with zeros. If the process does not have write access, the procedure 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, EXE$QIO allocates a block of nonpaged system memory to contain an IRP.

Before EXE$QIO allocates an IRP, it raises the IPL of the processor to IPL$_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 IRP.

EXE$QIO attempts to allocate an IRP from a *lookaside list* containing preallocated IRPs. If no preallocated packets exist, the procedure calls a VAX/VMS routine that allocates an IRP from general nonpaged pool. This allocating routine synchronizes with the rest of the system so that it can allocate the memory needed. (The *VAX/VMS Internals and Data Structures* manual describes the allocation routines in detail.)

EXE$QIO resumes I/O preprocessing by writing a description of the I/O request into the fields of the IRP as follows. Note that this data encompasses the *device-independent* information associated with the request. It is up to the device driver's FDT routines or VAX/VMS common FDT routines to fill in the *device-dependent* portions of the IRP as described in Section 5.1.7 and Section 8.

Data	Field(s)
Size in bytes of the IRP	IRP$W_SIZE
Identification of the block as an IRP	IRP$B_TYPE
Access mode of the process at the time of the request	IRP$B_RMOD

Data	Field(s)
Process ID of the requesting process	IRP$L_PID
Address of an AST routine (if specified in the request) and its parameter[1]	IRPL_AST, IRPL_ASTPRM
For file-structured devices, address of a window-control block (WCB) that describes the physical location of part of the file	IRP$L_WIND
Address of the target device's UCB	IRP$L_UCB
I/O-function code[2]	IRP$W_FUNC
Number of event flag to set when processing of the I/O request is complete	IRP$B_EFN
Base software priority of the requesting process	IRP$B_PRI
Address of an I/O-status block (if specified in the request)	IRP$L_IOSB
Process I/O channel index number	IRP$W_CHAN
A flag indicating whether the I/O function is for buffered or direct I/O	IRP$V_BUFIO in IRP$W_STS
A flag indicating whether the I/O request is an input request	IRP$V_FUNC in IRP$W_STS
A flag indicating whether the I/O function is a physical-I/O function	IRP$V_PHYSIO in IRP$W_STS
Address of a diagnostic buffer (if specified in the request)[3] and a flag indicating that the buffer is present	IRP$L_DIAGBUF, IRP$V_ DIAGBUF in IRP$W_STS

[1]If the request specifies an AST, EXE$QIO also verifies that the request would not cause the process to exceed its AST quota. If it would, EXE$QIO aborts the request.

[2]For nonfile devices, EXE$QIO reduces read- and write-virtual-block functions to their equivalent read- and write-logical-block functions before storing a code.

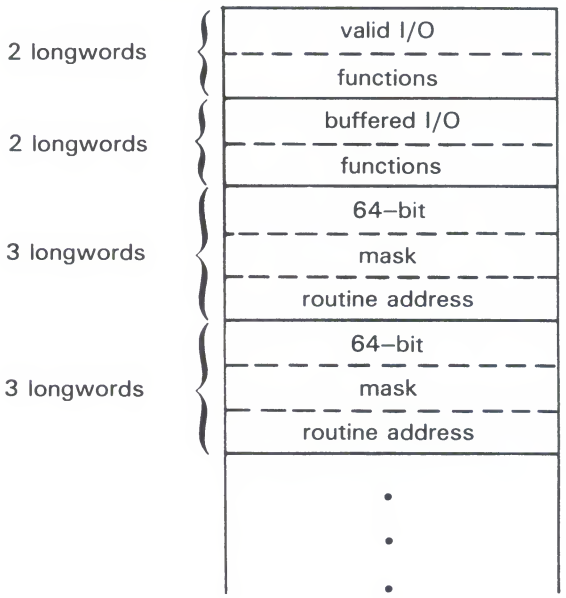
[3]The size of the diagnostic buffer is specified in the driver-dispatch table of the driver servicing the device unit to which the request is made. See Section 7.2 for more information.

Figure A-10 illustrates the format of an IRP; Table A-10 describes each of its fields.

5.1.7 FDT Processing

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

Figure 5–5 Layout of a 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) that permits you to define 64 unique I/O function codes for every device type. Table 7-1 lists the function codes defined by VAX/VMS. Section 7.3.1 describes how you can define device-specific function codes.
- A 10-bit I/O-function modifier (bits 6 through 15). In subsequent processing of the I/O request, the driver's start-I/O routine uses both I/O-function code and I/O-function modifier, as stored in IRP$W_FUNC, to create a device-specific function code to use in device activation.

The first two entries of a function-decision table are two longwords (64 bits) each. The first quadword entry is the *legal function bit mask* of all I/O-function codes that are valid for the device. The second quadword entry is the *buffered function bit mask* of those valid I/O-functions that are also buffered-I/O functions.

EXE$QIO uses the value of the low-order six bits of the I/O-function code to determine which bit to check in each of these bit masks. For example, if the function code has a value of 22, the procedure checks the twenty-third bit (bit 22) of each bit mask. Thus, EXE$QIO determines whether the I/O-function code is valid for the device and is able to charge against the appropriate quota of the requesting process for a direct- or buffered-I/O operation.[2]

[2] For physical- and logical-I/O operations, EXE$QIO also verifies that the process making the I/O request has suitable privileges.

Subsequent entries in the function-decision table are three-longwords long, and it is these entries that EXE$QIO uses to dispatch to the appropriate I/O preprocessing routine (FDT routine) for the requested function. Again, the first quadword is a 64-bit bit mask, and is checked by EXE$QIO in exactly the same way as the legal function bit mask and the buffered function bit mask. These *action routine bit masks,* however, contain the address of an FDT routine in the subsequent longword, and it is to this FDT routine that EXE$QIO transfers control when it discovers the bit corresponding to the I/O-function set in the quadword.

Some FDT routines are present in the operating system because they provide common services for many devices. Section 8.5 describes these routines. Other routines are included in the device driver because they perform device-dependent services.

EXE$QIO uses the action routine bit mask entries in the function-decision table to call FDT routines in the driver or system, according to the following strategy:

1 If the bit corresponding to the function code is set in the action routine bit mask, EXE$QIO calls the FDT routine whose address appears in the following longword.

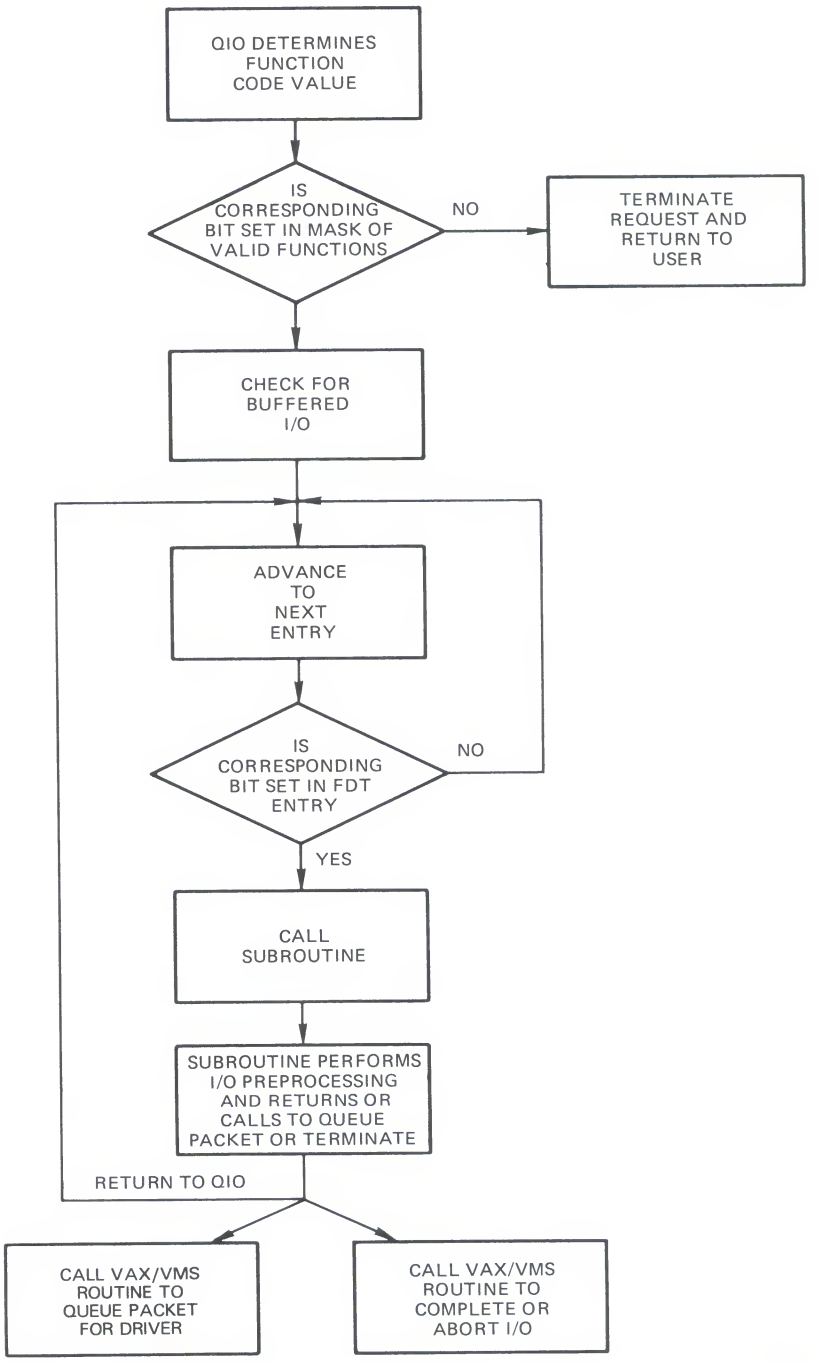
   - If this I/O-function requires additional preprocessing after this particular FDT routine completes its activity, the FDT routine returns control to EXE$QIO with an RSB instruction. When EXE$QIO regains control, it advances to the next action routine bit mask and repeats Step 1.
   - If this FDT routine completes all necessary preprocessing for this particular I/O-function, then it transfers control to a VAX/VMS routine that queues the IRP or completes the request.

2 If the bit corresponding to the function code is not set, EXE$QIO advances to the next action routine bit mask in the table and repeats Step 1.

Note: A single function-decision table can specify that EXE$QIO call more than one FDT routine to perform the many and varied steps in the preprocessing of a single I/O-function. However, it is the responsibility of the FDT routine that ultimately completes the preprocessing to end EXE$QIO's scan of the function-decision table. An FDT routine accomplishes this by transferring control to either a VAX/VMS routine that queues the I/O request for the driver's start-I/O routine or one that completes or aborts the request (see Figure 5–2). In other words, for each valid I/O-function code for a device, an FDT entry must contain the address of a routine that ends I/O preprocessing.

FDT routines execute in the 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-servicing routine.

Figure 5–6 FDT Routines and I/O Preprocessing



5.2 Handling Device Activity

When I/O preprocessing is complete, the last-called FDT routine generally jumps (with a JMP instruction) to a routine called EXE$QIODRVPKT.[3] EXE$QIODRVPKT, in turn, transfers control (using a JSB instruction) to EXE$INSIOQ, the VAX/VMS routine that queues IRPs and arbitrates device activity. (See Figure 5–2 for a representation of the flow of I/O-request processing at this juncture.)

5.2.1 Creating a Driver Fork Process to Start I/O

EXE$INSIOQ creates only one driver fork process at a time for each device unit on the system. As a result, only one IRP per device unit is serviced at one time. EXE$INSIOQ 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 EXE$INSIOQ immediately calls IOC$INITIATE to create and transfer control to a driver fork process to execute the driver's start-I/O routine.
- If the device is busy, a driver fork process already exists for the device, servicing some other I/O request. In this case, EXE$INSIOQ calls EXE$INSERTIRP to insert the IRP into a queue of IRPs waiting for the device unit. The routine queues the IRP according to the base priority of the caller. Within each priority, IRPs are in first-in/first-out order. The completion of the current I/O request triggers the servicing of the I/O request that is first in the queue, according to the procedure described in Section 12.1.2.3.

In the latter case, by the time the driver's start-I/O routine gains control to dequeue the IRP, the originating user's process context is no longer available. Because the context of the process initiating the I/O request is not guaranteed to a driver's start-I/O routine, the driver must execute in the reduced context available to a fork process.

IOC$INITIATE always initiates the driver's start-I/O routine with a context that is appropriate for a fork process. VAX/VMS establishes this context by performing the following steps:

1. Raising IPL to driver fork IPL (UCB$B_FIPL)
2. Loading the address of the IRP into R3
3. Loading the address of the device's UCB into R5
4. Transferring control (with a JMP instruction) to the entry point of the device driver's start-I/O routine

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.

[3] The rules for exiting from FDT preprocessing, including descriptions of EXE$QIODRVPKT and other FDT exit routines, appear in Sections 8.2 and 8.6.

- 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 fork IPL, 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

Depending on the device type supported by the driver, the start-I/O routine performs some or all of the following steps:

1. Analyzes the I/O function and branches to driver code that prepares the UCB and the device for that I/O operation
2. Copies the contents of fields in the IRP into the UCB
3. Tests fields in the UCB to determine whether the device and/or volume mounted on the device are valid
4. If the device is attached to a multiunit controller, obtains the controller data channel
5. If the I/O operation is a DMA transfer, obtains a I/O adapter resources such as mapping registers and a UNIBUS adapter data path
6. Loads all necessary device registers except for the device's control and status register (CSR)
7. Raises IPL to IPL$_POWER (saving the value of fork IPL on the stack) and confirms that a power failure that would invalidate the device operation has not occurred
8. Loads the device's CSR to activate the device
9. Invokes a VAX/VMS routine (using either the WFIKPCH or WFIRLCH macro) to suspend the driver fork process until a device interrupt or timeout occurs

As it suspends the driver, IOC$WFIKPCH or IOC$WFIRLCH saves the driver's context in the UCB. This context consists of the following information:

- A description of the I/O request and the state of the device
- The contents of R3 and R4 (UCBL_FR3, UCBL_FR4)
- The implicit contents of R5 as the address of the UCB
- A driver return address (UCB$L_FPC)
- The address of a device timeout handler (at UCB$L_FPC)
- The time at which the device will time out (UCB$L_DUETIM)

By convention, R4 often contains the address of the 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 database that is relevant to the current I/O operation.

Once the driver's start-I/O routine initiates the transfer, the driver invokes a VAX/VMS routine (with a macro such as WFIKPCH or WFIRLCH) to wait for the device to interrupt. This routine (IOC$WFIKPCH or IOC$WFIRLCH) expects to find, among the items it inherits on the stack, the driver's fork IPL, as placed there by the start-I/O routine in Step 7 above. Having removed the driver's start I/O routine's return address from the stack and stored it in UCBL_FPC, IOCWFIKPCH (or IOC$WFIRLCH) restores IPL to fork IPL from the stack and exits with an RSB instruction. Thus, IOC$WFIKPCH (or IOC$WFIRLCH) effectively passes control to the caller of its caller. In this case, the caller of the driver start-I/O routine that called IOC$WFIKPCH is EXE$INSIOQ. The flow back from EXE$INSIOQ to a user process that asynchronously requested the I/O operation is shown in Figure 5–2.

You can find additional information on the context of a start-I/O routine in Section 9.

5.2.3 Handling a Device Interrupt

When the device requests an interrupt, the interrupt dispatcher transfers control to the driver interrupt-servicing routine. The driver's interrupt-servicing routine runs at a high interrupt priority level so that the routine can service interrupts quickly. A driver interrupt-servicing routine usually performs the following processing:

1. For multiunit device controllers, determines which device unit generated the interrupt
2. Examines the UCB for the device to confirm that the driver fork process expects the interrupt
3. Saves device registers
4. Reactivates the suspended driver fork process

If necessary, the reactivated driver fork process executes at the high IPL of the interrupt-servicing 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 (by means of the IOFORK macro) that performs the following steps:

1. Disables the timeout that was specified in the wait-for-interrupt routine
2. Saves R3 and R4 (these are the registers needed to execute as a fork process) (UCBL_FR3, UCBL_FR4)

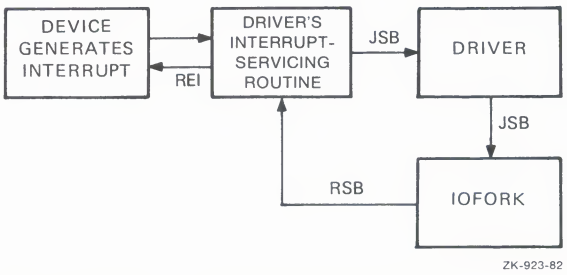
3 Saves the address of the instruction following the IOFORK request in the UCB fork block (UCB$L_FPC)

4 Places the address of the UCB fork block from R5 in a fork queue for the driver's fork level

5 Returns to the driver's interrupt-servicing routine

The interrupt-servicing routine then cleans up the stack, restores registers, and dismisses the interrupt. Figure 5–7 illustrates the flow of control in a driver that creates 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 these actions:

1 Removes a driver fork block from the fork queue

2 Restores fork context

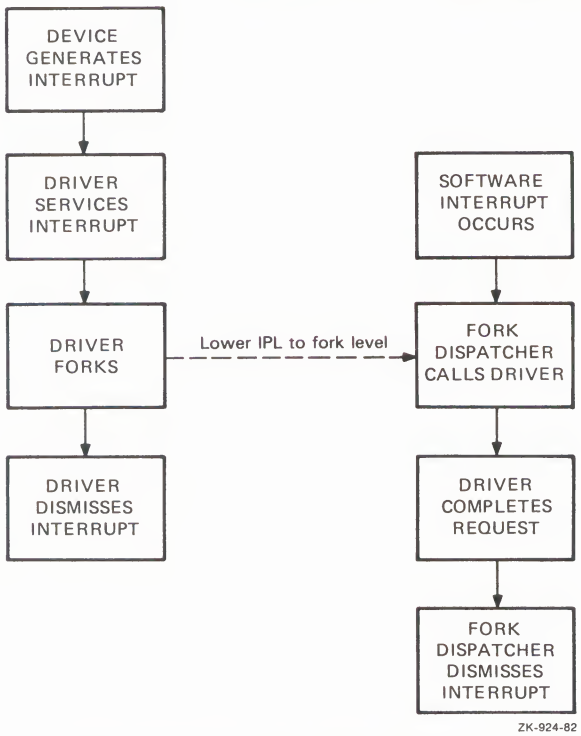
3 Transfers control back to the fork process

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 execution, the fork dispatcher removes the next entry, if any, from the fork queue, restores its fork process context, and reactivates it. This sequence is repeated 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 Completing an I/O Request

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

1. Releases shared driver resources, such as I/O adapter mapping registers, UNIBUS adapter data path, and controller ownership
2. 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:

1. Writes return status from the driver into the IRP
2. Inserts the finished IRP in the I/O-postprocessing fork-queue and requests an interrupt at IPL$_IOPOST
3. Creates a new fork process for the next IRP in the device's pending I/O queue
4. 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-servicing routine. This VAX/VMS routine completes device-independent processing of the I/O request.

Using the IRP as a source of information, the IPL$_IOPOST dispatcher executes the sequence below for each IRP in the postprocessing queue:

1. Removes the IRP from the queue
2. If the I/O function was a direct I/O function, adjusts the recorded use of the issuing process' direct I/O quota and unlocks the pages involved in the I/O transfer
3. If the I/O function was a buffered I/O function, adjusts the recorded use of the issuing process' buffered I/O quota and, if the I/O was a write function, deallocates the system buffers used in the transfer
4. Posts the event flag associated with the I/O request
5. Queues a special kernel-mode-AST routine to the process that issued the $QIO system service call

The queuing of a special 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' address space. The special kernel-mode-AST routine writes the following data into the process' 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-mode-AST routine, the special 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 IRP. The first part of an IRP is the AST-control block for user requested ASTs.

PART II Writing a Device Driver

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 mapping 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.

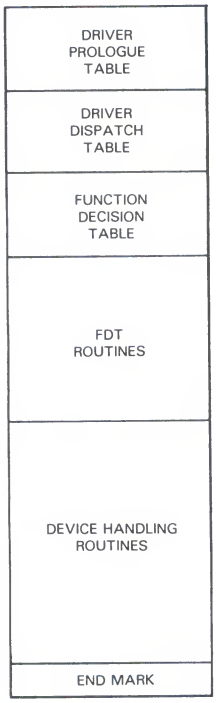
Note that the "how to" chapters describe a common approach to the design of various driver routines; they are examples. They do not present the only approach that can be taken to writing a driver.

6 Template for a Device 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 part of a driver. The order of the remaining tables and routines varies from driver to driver.

Figure 6–1 Driver Organization



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. Section 14 describes the driver-loading procedure.

Some drivers contain no device-dependent, FDT 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-servicing 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 might 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 database by loading the address of a data structure into a general register and using displacement addressing to the fields of the data structure.

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 the Use of Device-Register I/O Space

The programmer of a device driver for a UNIBUS device must observe the following restrictions on the use of 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 on line?
```

- 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 bit field instruction to test a bit in a device register.
- Drivers cannot use string-handling instructions.
- Drivers can use only those instructions that modify or write to a maximum of one 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 reread the device register, which causes undesirable device side effects or data loss.
- To access I/O space, use only the following instructions. These instructions cannot be interrupted 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
BICPSW
BIS(B,W,L)2
BIS(B,W,L)3
BISPSL
BISPSW
BIT(B,W,L)
CASE(B,W,L)
CHM(K,E,S,U)
CLR(B,W,L)
CMP(B,W,L)
CVT(BW,BL,WB,
WL,LB,LW)
DEC(B,W,L)
INC(B,W,L)
MOVAQ
MOVPSL
MOVZ(BW,BL,WL)
MTPR
PROBE(R,W)
PUSHA(B,W,L)
PUSHAQ
PUSHL
SBWC
SUB(B,W,L)2
SUB(B,W,L)3
TST(B,W,L)
XOR(B,W,L)2
XOR(B,W,L)3

6.3 Implementing Conditional Code in a Driver

When writing a DMA driver to function for equivalent devices on different I/O bus implementations, DIGITAL recommends that you use the CPUDISP macro in code paths that need to differentiate between the systems.

The CPUDISP macro (defined in SYS$LIBRARY:LIB.MLB) provides a means for indirectly distinguishing between bus structures based on the type of the VAX processor that currently uses that bus structure. Use CPUDISP when it is necessary to conditionally execute pieces of code, for instance, the allocation and loading of mapping registers for those processors (for example, MicroVAX II, VAX–11/780, VAX 8200) whose I/O space contains mapping registers, or the allocation of a physically contiguous buffer for a DMA transfer on the MicroVAX I (which cannot map such a transfer).

CAUTION: **CPUDISP exists as a temporary means of dispatching to code conditional to the type of the executing processor. Although, it currently functions to distinguish between the I/O bus configurations used by each processor, it most likely will not continue to do so as processors migrate to the various I/O bus configurations.**

CPUDISP builds a case table, first forming the appropriate symbolic constants (PR$_SID_TYPE*xxx*) as displacement values and branching to a transfer address according to the contents of global symbol EXE$GB_CPUTYPE. Currently, the only values accepted for CPU-type are 8NN (for VAX 8800), 790 (for VAX 8600 and VAX 8650), 8SS (for VAX 8200), 780, 750, 730, UV1 (for MicroVAX I), or UV2 (for MicroVAX II). For example:

```
        CPUDISP  <<UV1,5$>,-
                 <UV2, 1$>>,-
                 CONTINUE=YES     ;for all other types of CPU continue
                                  ;for 790,785,780,750,730,8SS,8NN
    .
    .
    .
        BRB 10$
1$:                              ;for UV2
    .
    .
    .
5$:                              ;for UV1
    .
    .
    .
10$:                             ;for all others
```

Appendixes E and F contain examples of drivers that use the CPUDISP macro and other techniques (for example, a longword of bit flags in an extension to the UCB) to provide conditional code in a driver. See also the description of the CPUDISP macro in Appendix B.

6.4 Driver Template

The following pages list the VAX/VMS template driver. You can obtain a machine-readable copy of it from SYS$EXAMPLES:TDRIVER.MAR.

```
.TITLE  TDRIVER - VAX/VMS TEMPLATE DRIVER
        .IDENT  'V04-000'

;
;
;
;  Copyright (c) 1978, 1980, 1982, 1984
;  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-servicing 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 three places:
;
;               At system startup
;               During driver loading and reloading
;               During recovery from a power failure
;
; Inputs:
;
;       R4      - address of the CSR (control/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 (control/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 on line
        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 (processor 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 (control/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
```

7 Writing Device-Driver Tables

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

- **Driver-prologue table**—describes the device type, driver name, and fields in the I/O database to be initialized during driver loading and reloading.
- **Driver-dispatch table**—lists some of the driver's entry points to which VAX/VMS transfers control. The channel-request block and function-decision table list other entry points.
- **Function-decision table**—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. Argument values for such macros can be expressed in the following format:

```
KEYWORD=argument-value
```

The *VAX MACRO and Instruction Set Reference Volume* describes the syntax rules for keyword macros in detail. The sections that follow provide examples of macro usage.

7.1 Driver-Prologue Table

The driver-prologue table (DPT) is the first part of 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 data structures for the new device(s) in the I/O database. The loading procedure also links the new DPT into a list of all DPTs known to the system. Section 14 describes how the driver-loading procedure decides which data structures to build for a given device.

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

Figure A-8 illustrates the DPT data structure, and Table A-8 describes its contents. 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.

7.1.1 DPTAB Macro

To create a DPT, the driver invokes the DPTAB macro.

Format

```
DPTAB  end ,adapter ,[flags=0] ,ucbsize ,[unload] ,[maxunits=8] -
       ,[defunits=1] ,[deliver] ,[vector] ,name
```

Arguments

end

Name of the label at the end of the driver module.

adapter

Adapter type. The adapter type can be any of the following:

UBA	UNIBUS adapter or MicroVAX II or MicroVAX I Q22 bus interface
MBA	MASSBUS adapter
DR	DR device
NULL	No actual device for driver

[flags=0]

Flags used in loading the driver. These flags include the following:

DPT$M_SVP	When set, causes the driver-loading procedure to allocate a permanent system page-table entry (PTE) for the device. The index to the virtual address of the permanently allocated system PTE is stored in UCB$L_SVPN when the UCB is created. A driver can calculate the system virtual address of the page described by this index by using the formula: $(index * 200_{16}) + 80000000_{16}$ Disk drivers use this system PTE during ECC error correction, and when using the system routines IOC$MOVFRUSER and IOC$MOVTOUSER, described in Appendix C.
DPT$M_NOUNLOAD	When set, indicates that the driver cannot be reloaded except in the event of a system bootstrap.

ucbsize

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

[unload]

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=8]
Maximum number of units that this driver supports on a controller. This field affects the size of the interrupt-dispatch block created by SYSGEN's CONNECT command. If this field is omitted, the default is eight units. You can override the default by appending the /MAXUNITS qualifier to the CONNECT command.

[defunits=1]
Number of units created by default for each controller that SYSGEN's AUTOCONFIGURE command processes on behalf of this driver. The unit numbers created are 0 through **defunits**–1. 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 a unit-delivery routine that AUTOCONFIGURE calls to determine whether or not to create each unit automatically.

[deliver]
Address of a unit-delivery routine that AUTOCONFIGURE calls to determine which units to configure automatically for the device supported by this driver.

[vector]
Address of a driver-specific transfer vector. Use of this argument is reserved to DIGITAL.

name
Name of the device driver module. The driver-loading procedure will permit the loading of only one copy of the driver associated with this name. 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

Most device drivers need to initialize certain fields of the I/O database 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 DPT 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 a CONNECT command causes SYSGEN to build I/O database data structures and when the driver is reloaded
- Fields to be initialized only when SYSGEN is given the RELOAD command, causing the driver to be reloaded

The DPTAB macro stores the relative addresses of these two lists, called initialization and reinitialization data, in the DPT. The list of one or more invocations of the DPT_STORE macro must appear after the *DPTAB* 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

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-servicing 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 and Q22 bus devices.

The DPT_STORE macro either declares an assembly language label or describes a field to be initialized.

Format

```
DPT_STORE  type ,offset ,oper ,exp [,pos] [,size]
```

Arguments

type

Type of data structure into which the data is to be stored (CRB, DDB, IDB, ORB, or UCB); or label denoting a table marker. A label can be any 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.

If this argument is a label, no other argument is allowed. In this case, only fields in the DPT are affected.

offset

Unsigned offset into the data structure. 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.

oper

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 **pos** and **size** arguments as operands of an INSV instruction.

An at sign (@) preceding the **oper** argument indicates that the **expression** argument that follows is the address of the initialization data.

exp

Expression to be stored in the data structure or, if an at sign (@) is specified preceding the **oper** argument, 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
```

[pos]

Starting bit position within the specified field. This argument is specified only for V operations.

[size]

Number of bits in the field. This argument is specified only for V operations.

7.1.3 Example of DPTAB and DPT_STORE Macros

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

```
DPTAB -                              ; Define DPT
      END=XX_END,-                   ; End of driver
      ADAPTER=UBA,-                  ; Adapter type
      UCBSIZE=UCB$K_XX_LENGTH, -     ; Size of UCB
      NAME=XXDRIVER                  ; Name of driver module
DPT_STORE INIT                       ; Start of data structure
                                     ; initialization values
DPT_STORE UCB,UCB$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 data structure
                                     ; 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

The driver-dispatch table (DDT) lists some of the entry points for driver routines to be called by VAX/VMS for I/O processing. Every driver must create a DDT.

The routines listed in the DDT 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 DDT contains relative addresses for routines located in the driver module and absolute addresses for routines located in the operating system. At loading time, the driver-loading procedure changes the relative addresses of driver routines to absolute addresses.

The driver creates the DDT by invoking the macro DDTAB. The driver-loading procedure writes the address of the DDT table, as specified in a DPT_STORE macro, into the device-data block. Figure A-7 illustrates the structure of a DDT and Table A-7 describes its contents.

7.2.1 DDTAB Macro

The DDTAB macro creates a DDT. 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
```

The DDTAB macro writes the address of the VAX/VMS routine IOC$RETURN into routine address fields of the DDT that are not supplied in the macro invocation (with the exception of the **mntver** argument). IOC$RETURN simply executes an RSB instruction.

Format

```
DDTAB  devnam ,[start=IOC$RETURN] ,[unsolic=IOC$RETURN] ,functb -
       [,cancel=IOC$RETURN] [,regdmp=IOC$RETURN] [,diagbf=0] -
       [,erlgbf=0] [,unitinit=IOC$RETURN] [,altstart=IOC$RETURN] -
       [,mntver=IOC$MNTVER] [,cloneducb=IOC$RETURN]
```

Arguments

devnam
Generic name of the device.

[start=IOC$RETURN]
Address of the driver's start-I/O routine.

[unsolic=IOC$RETURN]
Address of the routine that services unsolicited interrupts from the device. Only MASSBUS device drivers use this field.

functb
Address of the driver's function-decision table.

[cancel=IOC$RETURN]
Address of the driver's cancel-I/O routine.

[regdmp=IOC$RETURN
Address of the routine that dumps the device registers to an error log buffer or to a diagnostic buffer.

[diagbf=0]
Length in bytes of the diagnostic buffer used for this device.

[erlgbf=0]
Length in bytes of the error log buffer used for this device.

[unitinit=IOC$RETURN]
Address of the device unit-initialization routine, if one exists. MASSBUS drivers should use this field rather than CRB$L_INTD+VEC$L_UNITINIT. UNIBUS or Q22 bus drivers can use either one.

[altstart=IOC$RETURN]
Address of the alternate start-I/O routine. To initiate this routine, a driver FDT routine exits by means of VAX/VMS routine EXE$ALTQUEPKT instead of EXE$QIODRVPKT (see Section 8.6).

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

[cloneducb=IOC$RETURN]
Address of a VAX/VMS routine to call when a UCB is cloned by the $ASSIGN system service

7.2.2 Example of DDTAB Macro

In the following example of using the DDTAB macro, 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.

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

7.3 Function-Decision Table

The function-decision table (FDT) 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 an FDT 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 FDT specify one or more FDT routines for each valid I/O-function code for the device. The FDT routines must terminate the I/O preprocessing for each type of function by transferring control out of the $QIO 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 Section 8 for information on the writing of FDT routines.

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

Table 7–1 VAX/VMS I/O-Function Codes

Function	Codes Defined	Description
Physical I/O	IO$_AVAILABLE	Set device available (required by all disk drivers)
	IO$_DIAGNOSE	Diagnose
	IO$_DRVCLR	Drive clear
	IO$_ERASETAPE	Erase tape
	IO$_NOP	No operation
	IO$_OFFSET	Offset read heads
	IO$_PACKACK	Pack acknowledge (required by all disk drivers)
	IO$_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 (required by all disk drivers)
	IO$_WRITECHECK	Write check data

Table 7–1 (Cont.) VAX/VMS I/O-Function Codes

Function	Codes Defined	Description
	IO$_WRITECHECKH	Write check header and data
	IO$_WRITEHEAD	Write header and data
	IO$_WRITEMARK	Write tape mark
	IO$_WRITEPBLK	Write physical block
	IO$_WRITETRACKD	Write track data
Logical I/O	IO$_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 I/O	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

7.3.1 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 an FDT by invoking the FUNCTAB macro. Each invocation of the FUNCTAB macro creates a 2- or 3-longword entry in the FDT. 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 a routine that is to perform preprocessing for those functions. These invocations create 3-longword entries.

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

7.3.2 Defining Buffered-I/O Functions

The second entry in an FDT is a *buffered function bit mask* that indicates which legal functions the driver handles 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, an interrupt-servicing routine for a terminal) cannot gain access to the data if direct I/O is used. Therefore, transfers that require data manipulation must be buffered I/O.
- VAX/VMS handles the quotas differently for direct I/O and buffered I/O, as described in the *VAX/VMS System Manager's Reference Manual*.
- Generally, DMA devices use direct I/O, while programmed I/O devices use buffered I/O.

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

7.3.3 FUNCTAB Macro

The FUNCTAB macro creates an FDT for a driver.

Format

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

Arguments

[action]

Address of a routine to call during I/O preprocessing of the specified I/O-function code or codes. A 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 routines.

codes
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

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
               >
```

8 Writing FDT Routines

The $QIO system service uses the driver's function-decision table (FDT) 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, described in Section 8.5, whenever possible. This practice encourages the use of well debugged routines and minimizes driver size.

8.1 Context of FDT Routine Execution

The $QIO system service executes in the context of the process that issues the I/O request, but in kernel mode and at IPL$_ASTDEL. The process is executing in kernel mode because the dispatching of the $QIO system service executes a CHMK instruction. Virtual addresses are mapped according to the process page tables. This mapping allows FDT routines access to user-specified virtual addresses. The $QIO system service expects FDT routines to preserve this context. Therefore, an FDT routine observes the following conventions:

- It cannot call VAX/VMS system services or VAX RMS services.
- 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 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 8.2.

Before calling an FDT routine, the $QIO system service sets up the contents of certain registers, as described in Table 8–1.

Table 8–1 Registers Loaded by the $QIO System Service

Register	Content
R0	Address of FDT routine being called
R3	Address of IRP for current I/O request
R4	Address of process-control block (PCB) of *current process*
R5	Address of UCB of device assigned to user-specified process-I/O channel
R6	Address of CCB that describes user-specified process-I/O channel
R7	Bit number of user-specified I/O-function code

Table 8–1 (Cont.) Registers Loaded by the $QIO System Service

Register	Content
R8	Address of current entry in FDT
AP	Address of first function-dependent argument (**p1**) specified in I/O request

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 PUSHR and POPR instructions to save registers on the stack and later restore them.

8.2 Transferring Into and Out of an FDT Routine

To transfer control to an FDT routine, the $QIO system service loads the address of the FDT routine into a register and executes a JSB instruction, as follows:

```
JSB     (R0)
```

Each FDT routine chooses an exit path on the basis of 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

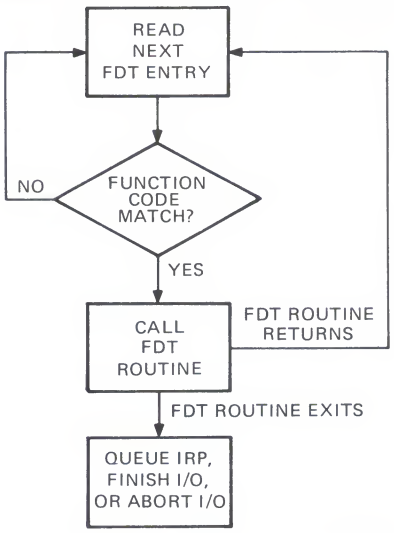
The FDT routines, as illustrated in Figure 8–1, must transfer control out of the FDT processing loop and into a VAX/VMS routine that queues an IRP, completes an I/O request, or aborts an I/O request. The $QIO system service does not stop scanning the FDT. Therefore, you must ensure that all valid function codes in a driver's FDT eventually call an FDT routine that does not return control to the $QIO system service.

An FDT routine can exit using any of the following methods:

- RSB
- JMP G^EXE$QIODRVPKT
- JSB G^EXE$ALTQUEPKT
- JMP G^EXE$FINISHIO or JMP G^EXE$FINISHIOC
- JMP G^EXE$ABORTIO

These methods are described below, and you can find additional details on the routines they involve in Section 8.6. The first method listed returns to the $QIO system service. All other methods jump to VAX/VMS routines that take the appropriate action.

Figure 8–1 $QIO Scan of a Function-Decision Table



RSB

Returns to the $QIO system service. The FDT routine returns to the system service because the routine knows that the FDT 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

Transfers control to a VAX/VMS routine that queues an IRP 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 this routine, no driver code that further processes the I/O request can refer to the process virtual address space.

EXE$QIODRVPKT is the standard method used to queue an I/O request for device activity. This routine initiates driver action only if the device unit is currently idle, if no I/O request is being processed. If the device unit is busy, EXE$QIODRVPKT queues the request to the unit so that the driver will process it when the unit becomes available.

JSB G^EXE$ALTQUEPKT

Transfers control to a VAX/VMS routine that calls an alternate start-I/O routine in the driver that synchronizes requests for activity on a device unit and initiates the processing of I/O requests. A driver that can handle two or more I/O requests simultaneously uses this exit method.

The FDT routine uses this exit method when it has successfully completed all driver preprocessing and the request requires device activity. However, in contrast to EXE$QIODRVPKT, EXE$ALTQUEPKT initiates driver action

at this driver's alternate start-I/O routine entry point without regard for the device unit's activity status.

JMP G^EXE$FINISHIO or JMP G^EXE$FINISHIOC

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 IRP (IRP$L_MEDIA and IRP$L_MEDIA+4).[1] The routine then inserts the IRP 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 $QIO system service the two longwords of status contained in the I/O-status block (if any) specified in the I/O request.

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.

8.3 FDT Routines for VMS 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 Section 8.5.

8.4 FDT Routines for VMS 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 confirm either read or write access to the user's buffer and allocate a buffer in system space.

8.4.1 Checking Accessibility of 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$L_BCNT. EXE$READCHK also sets IRP$V_FUNC in IRP$W_STS to indicate that the function is a read.

[1] EXE$FINISHIOC clears the second longword of the final I/O status.

8.4.2 Allocating the System Buffer

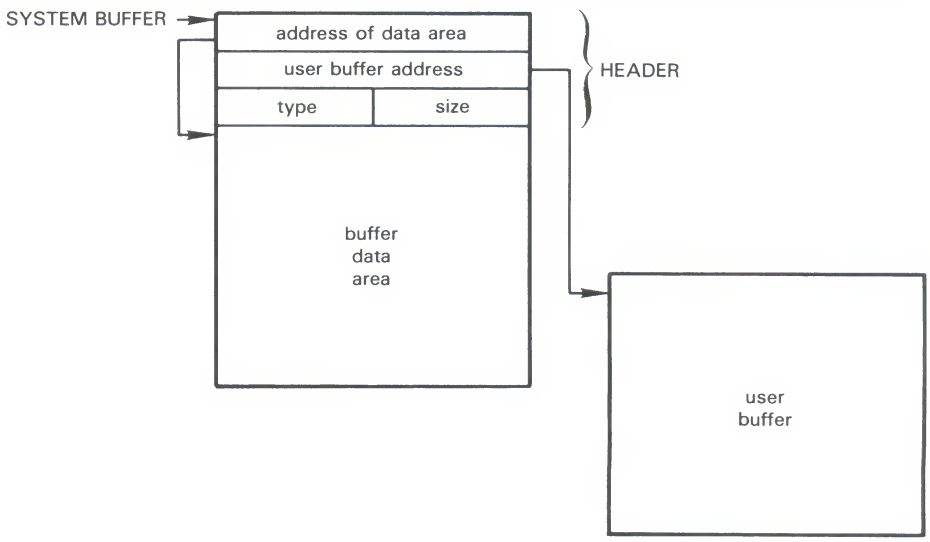
Next, the FDT routine allocates a system buffer. First, it adds 12 bytes for a buffer header to the byte count passed in the **p2** argument of the user's I/O request. This is the total system buffer size. The FDT routine then calls EXE$BUFFRQUOTA to ensure that the user has sufficient remaining resources. If EXE$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:

1. Loads the address of the system buffer into IRP$L_SVAPTE.
2. Loads the total size of the system buffer into IRP$W_BOFF.
3. 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).
4. Stores the starting address of the system buffer data area in the first longword of the buffer header.
5. Stores the user's buffer address in the second longword of the header.
6. 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 a Buffered-I/O Read Function



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

8.4.3 Buffered-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 three steps:

1. Adds the value in IRP$W_BOFF to JIB$L_BYTCNT to update the user's byte count quota
2. If IRP$L_SVAPTE is nonzero, assumes a system buffer was allocated and checks to see whether IRP$V_FUNC is set in IRP$W_STS
3. If IRP$V_FUNC is clear, deallocates the system buffer used for the write operation; if IRP$V_FUNC is set, the special kernel-mode AST copies the data to the user's buffer and then deallocates the buffer in addition to performing other kernel-mode AST functions

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

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

8.5 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 EXE$QIODRVPKT, EXE$FINISHIO, EXE$FINISHIOC, or EXE$ABORTIO. Once a VAX/VMS FDT routine is called, no subsequent FDT processing occurs.

For additional information about VAX/VMS FDT routines, see the pertinent routine descriptions in Appendix C.[2]

[2] For disk drivers, VAX/VMS supplies the FDT routine EXE$LCLDSKVALID, described in Appendix C, that processes an IO$_PACKACK, IO$_AVAILABLE, or IO$_UNLOAD function on a local disk. This routine must be the last FDT routine called for the function, and dispatches to either EXE$FINISHIO or EXE$QIODRVPKT when it completes FDT processing.

8.5.1 EXE$ONEPARM

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

Exit Method

Queues the IRP 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 IRP. Then, it queues the IRP to the driver.

8.5.2 EXE$READ

EXE$READ processes a logical-read or physical-read function for a direct I/O operation. EXE$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 IRP to a driver.

Description

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

EXE$READ writes the fourth device-dependent argument to the I/O request (**p4**), located at 12(AP) into the carriage-control field (IRP$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 IRP.

If argument **p2** (the transfer byte count) of the $QIO system service call is zero, EXE$READ queues the IRP to a device driver. Argument **p2** 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$READCHKR finds that the buffer is accessible, it updates the IRP by writing the size in bytes of the transfer to IRP$L_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 65,535 (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 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 IRP and restarts the request procedure at the $QIO 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 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.

8.5.3 EXE$SENSEMODE

EXE$SENSEMODE processes the sense-device-mode and sense-device-characteristics functions by reading fields of the UCB. No device activity occurs.

Exit Method

Transfers control to EXE$FINISHIO.

Description

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

8.5.4 EXE$SETCHAR

EXE$SETCHAR processes the set-device-mode and set-device-characteristics functions. If setting device characteristics requires no device activity or requires no synchronization with fork processing, the driver's FDT entry can specify EXE$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 argument to the I/O request (**p1**), 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$SETCHAR stores the new characteristics in fields of the device's UCB. If the function is IO$_SETCHAR, the device type and class fields (UCB$B_DEVCLASS and UCB$B_DEVTYPE, respectively) of the UCB receive the first word of data contained in the quadword.

For both the IO$_SETCHAR and IO$_SETMODE functions, the routine writes the second word of data into the UCB's 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 IRP in the I/O postprocessing queue.

8.5.5 EXE$SETMODE

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

Exit Method

Aborts the I/O request if an error occurs; otherwise, queues the IRP 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 argument to the I/O request (**p1**), 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$L_MEDIA+4) of the IRP. The routine then transfers control to the exit routine EXE$QIODRVPKT, which queues the request to the appropriate device driver.

8.5.6 EXE$WRITE

EXE$WRITE processes a logical- or physical-write function for a direct I/O operation. EXE$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 IRP to a driver.

Description

Writes the fourth argument to the I/O request (**p4**), found at 12(AP) into the IRP's carriage control field (IRP$B_CARCON).

EXE$WRITE replaces the logical-function code IO$_WRITELBLK with the physical-function code IO$_WRITEPBLK in the function code field of the IRP (IRP$W_FUNC).

If argument **p2** to the I/O request (the transfer byte count) is zero, EXE$WRITE queues the IRP to the driver. Argument **p2** is found at 4(AP). If the byte count is not zero, EXE$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$WRITECHKR finds that the buffer is accessible, it updates the IRP by writing the size in bytes of the transfer to IRP$L_BCNT. EXE$WRITE can transfer a maximum of 65,535 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 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 IRP and restarts the request procedure at the $QIO 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 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.

8.5.7 EXE$ZEROPARM

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

Exit Method

Queues the IRP to the driver.

Description

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

8.6 VAX/VMS Exit Routines

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.

8.6.1 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 $QIO system service final status code

R3 Address of current IRP

R4 Address of process-control block (PCB) of current process

R5 Address of UCB of device unit assigned to process-I/O channel

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

Description

EXE$ABORTIO clears the address of the I/O-status block in the IRP (IRP$L_IOSB) so that no status will be returned during I/O postprocessing. EXE$ABORTIO also clears the bit in the IRP (ACB$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' use of its AST quota.

Then EXE$ABORTIO inserts the IRP in the I/O postprocessing queue. If no other entries are in the queue, EXE$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 user
- 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.

8.6.2 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 Contents

R0 Value to be placed in the first longword of final I/O status when the $QIO 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 current IRP

R4 Address of process-control block (PCB) of current process

R5 Address of UCB of device unit assigned to process-I/O channel

R3 through R5 always contain the IRP, 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 database and then complete the I/O request in the following steps:

1. Increase the number of I/O operations completed on the current device in the operation count field of the UCB (UCB$L_OPCNT)
2. Store the contents of R0 and R1 in the media fields of the IRP (IRP$L_MEDIA and IRP$L_MEDIA+4)
3. Insert the IRP in the I/O postprocessing queue and, if the queue is empty, request a software interrupt at IPL$_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 three steps:

1. Lower IPL to zero, which is the normal IPL for a process.
2. Change mode back to the original processor access mode.
3. 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 I/O request has completed without device-independent error.

8.6.3 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 UCB. Routines usually access the UCB at driver fork IPL to synchronize modifications, and FDT routines do not execute at this interrupt priority level. Drivers containing FDT routines that access device registers or carelessly modify the UCB risk unpredictable operation or a system failure.

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

Required Register Contents

R3 Address of IRP

R4 Address of process-control block (PCB) of current process

R5 Address of the UCB for device unit assigned to 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$B_FIPL). Driver fork level is, by convention, the interrupt priority level at which device drivers and VAX/VMS read and alter critical portions of the device's UCB. By executing at fork level, EXE$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:

1. Sets the busy bit of the device's UCB (UCB$V_BSY in UCB$L_STS)
2. Stores the address of the current IRP in the UCB field UCB$L_IRP
3. Copies the transfer parameters contained in the IRP into the UCB:
   - Copies the starting address from IRP$L_SVAPTE to UCB$L_SVAPTE
   - Copies the byte offset within the page from IRP$W_BOFF to UCB$W_BOFF
   - Copies the low order word of the byte count from IRP$L_BCNT to UCB$W_BCNT
4. Clears the cancel-I/O and timeout bits in the UCB status word (UCB$V_CANCEL and UCB$V_TIMOUT in UCB$L_STS)
5. 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 to which IRP$L_DIAGBUF points (the $QIO system service having already allocated the buffer)
6. Finds the entry point of the device driver's start-I/O routine using the following chain of pointers:

   UCB → DDT → start-I/O routine

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

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

- The time that the entry is queued; for each IPL, the queue is ordered on a first-in/first-out basis
- The priority of the IRP, which is derived from the requesting process' base priority and stored in the field IRP$B_PRI

After completing one of the operations described above, EXE$INSIOQ reduces the interrupt priority level to IPL$_ASTDEL, the level at which it began executing. EXE$INSIOQ returns control to EXE$QIODRVPKT. Finally, EXE$QIODRVPKT returns from the $QIO system service in the following steps:

1. Loads a success status code (SS$_NORMAL) into R0
2. Reduces the interrupt priority level to 0
3. Changes mode to the access mode of the requesting process at the time of the I/O request by issuing an REI instruction
4. 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 might no longer be the mapped process. Therefore, the driver must assume that all information regarding the I/O request is in the UCB or the IRP and that all buffer addresses in the UCB are either system addresses or page-frame numbers that can be interpreted in any process context.

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

8.6.4 EXE$ALTQUEPKT

You might want special-purpose drivers to use their own internal I/O queues as well as the device unit's 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$ALTQUEPKT permits the driver to ignore synchronization of the I/O queue for the unit. When called by an FDT routine, EXE$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 must not only maintain its internal queues but must also synchronize those queues with the unit's pending I/O queue, which the operating system maintains.

Drivers complete I/O requests by calling the routine COM$POST. This routine places each IRP in a postprocessing queue and returns control to the driver. The driver can then fetch another IRP from an internal queue. For more information about COM$POST, see Appendix C.

If a driver processes more than one IRP 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 IRP

R5 Address of UCB

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:

1. Saves the current interrupt priority level on the stack
2. Raises interrupt priority level to driver fork level (UCB$B_FIPL)
3. Finds the entry point of the alternate start-I/O routine using the following chain of pointers:

   UCB → DDT → alternate start-I/O routine
4. 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:

1. Sets the success status code SS$_NORMAL in R0
2. Lowers the interrupt priority level to zero
3. Returns (with the RET instruction) to the system-service dispatcher

9 Writing a 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. Section 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 the Start-I/O Routine

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-request, 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 8.6.3.) 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 pending I/O queue.

After a device completes an I/O operation, the driver fork process exits by transferring control to IOC$REQCOM. IOC$REQCOM inserts the IRP for the finished transfer into the postprocessing queue. It then dequeues the next IRP from the device unit's pending I/O queue and calls IOC$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 the stack to its previous state before relinquishing control. The stack used depends on whether the I/O startup is the result of a new I/O request or because a previously requested I/O operation has been completed. The choice of stacks must not affect the operation of the start-I/O routine.

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 IRP.
- R5 contains the address of the UCB for the device.

The start-I/O routine must preserve all general registers except R0, R1, R2, and R4.

Before the packet-queuing routines call the start-I/O routine, they copy the following IRP fields into their corresponding slots in the device's UCB:

- IRP$L_BCNT (low-order word) → UCB$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 that perform the following functions:

- Analyze the I/O function
- Transfer the details of a transfer from the IRP into the UCB
- Obtain and initialize the controller and, for DMA transfers, I/O 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 representative DMA device such as a parallel communications link; the details of processing, however, are specific to the particular device. Section 10 describes the UNIBUS- and Q22 bus-related details of DMA transfers.

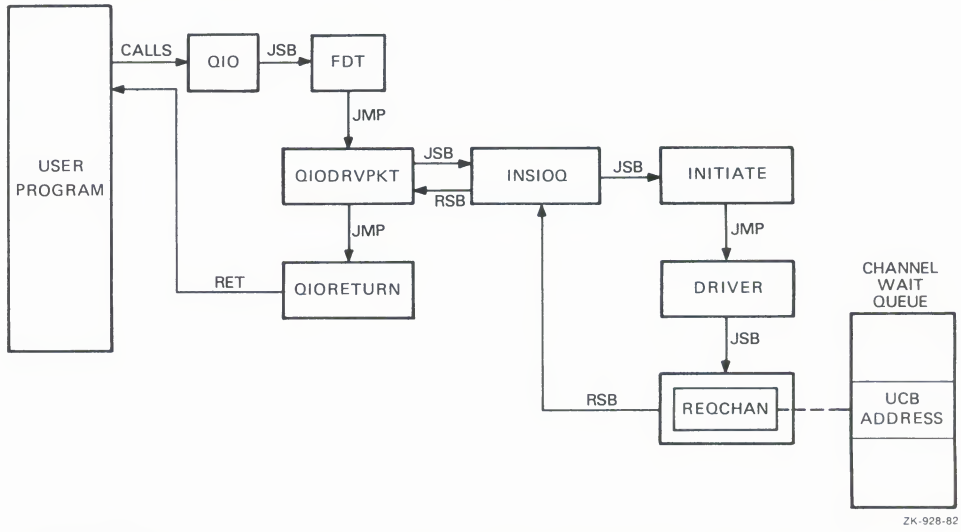
9.3.1 Obtaining Controller Access

If the device is one of several attached to a controller, the start-I/O routine invokes the VAX/VMS macro REQPCHAN to assign the controller's data channel to the device unit. Controllers that control only one device do not require arbitration for the controller's data channel. 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 EXE$INSIOQ, as illustrated in Figure 9–1. This procedure is further discussed in Section 3.3.1.

Figure 9–1 Inserting a UCB into the Channel-Wait Queue



The UCB fork block now represents the entire context of the suspended driver:

- Saved R3 containing the IRP address
- Implicitly saved R5 containing the UCB address
- A return address in the driver

IOC$REQPCHANL does not save R4 because it writes R4 before returning control to the driver.

If the channel is available, IOC$REQPCHANL locates the interrupt-dispatch block (IDB) for the channel with a pointer in the UCB:

UCB → CRB → IDB

The IDB contains the address of the control and status register (CSR) for the channel (IDB$L_CSR). IOC$REQPCHANL returns the CSR address in R4. The driver for a unit attached to a dedicated controller must contain the code needed to load the CSR address into R4.

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

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

At this point, the device driver follows procedures to obtain I/O bus resources, as detailed in Section 10.[1]

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 UCB (UCB$W_BCNT) by 2. The routine loads the computed value into the device's word-count 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.

[1] Because of the unavailability of mapping registers for MicroVAX I Q22 bus devices, coding for MicroVAX I DMA drivers diverges somewhat from the normal method of setting up a DMA transfer. Section 10.7 describes the means by which MicroVAX I DMA transfers are accomplished over the Q22 bus.

9.3.4 Computing the Transfer's Starting Address

The start-I/O routine calculates the address of the transfer using the byte-offset field of the UCB (UCB$W_BOFF) and the number of the starting mapping register (CRB$L_INTD+VEC$W_MAPREG). The result is an 18-bit value representing an address in UNIBUS or MicroVAX II Q22 bus address space.[2] Section 10.4 details the calculation of the starting address for a UNIBUS or MicroVAX II Q22 bus transfer.

The start-I/O routine stores the low-order 16 bits of the computed address in the device's buffer-address register. It stores the two high-order bits of the computed address in the memory-extension bits of the register that contains the bit mask described in Section 9.3.5. This register now contains the information on the device function that is to be placed in the device's CSR and the two high-order bits of the bus address.

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 bit and the go bit in the general register that also contains the high-order bits of the bus address and the device-function bits. At this point, the general register contains a complete command for starting the transfer, also known as the *control mask*.

When the start-I/O routine copies the contents of the register into the device's CSR, the device starts the transfer. Before activating the device, however, the start-I/O routine should perform the steps described in Sections 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's status word (UCB$V_POWER in UCB$L_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 might have occurred between the time that the start-I/O routine wrote the first device register and the time that the start-I/O routine is ready to activate the device. Such a power failure could modify the already-written device registers and cause unpredictable device behavior if the device were to be started.

If the bit UCB$V_POWER is set, the start-I/O routine branches to an error handler in the driver. The driver must clear UCB$V_POWER before error-recovery procedures can be started. Many drivers clear this field and transfer control to the beginning of the start-I/O routine, which restarts the processing of the I/O request.

[2] The MicroVAX II implements only 496 of its 8192 mapping registers; thus, 18 significant bits are adequate to select a Q22 bus address (see Section 4.2 for details).

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's CSR. When the device notices the new contents of the device register, it begins to transfer the requested data.

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 these events occurs:

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

Still executing at IPL$_POWER, the driver's start-I/O routine asks 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 9.3.6.

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

9.4.1 WFIKPCH and WFIRLCH Macro Formats

A start-I/O routine invokes either the WFIKPCH or WFIRLCH macro to wait for a device interrupt.

Formats

```
WFIKPCH  excpt [,time]
WFIRLCH  excpt [,time]
```

Arguments

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 65,536 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 65,536 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 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 resumes execution as a result of an interrupt-servicing routine's JSB instruction.

IOC$WFIKPCH then saves the contents of R3, R4, and the address to which control must be returned to the driver, which it takes from the top of the stack. It saves this information in the first part of the UCB in the UCB fork block.

Note that after an interrupt the interrupt-servicing routine must restore R5 so that it contains the address of the UCB. The interrupt-servicing routine normally obtains the address of the UCB from the field IDB$L_OWNER of the IDB.

The VAX/VMS routine that detects a device timeout calculates the address of the driver's timeout routine by subtracting 2 from the saved PC in the UCB's 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$WFIKPCH sets bits in the UCB (UCB$V_INT and UCB$V_TIM in UCB$L_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 fork level, the IPL at which the driver was executing previously. Then it 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 UCB. 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 (the address of the UCB fork block)
- The address at which to return control to the driver
- The implicit address of a device-timeout routine

By convention, R4 often contains the address of the CSR; it permits the driver's interrupt-servicing routine to examine device registers. When the driver's fork process regains control after an interrupt processing, R5 contains the UCB address. The UCB is the key to that part of the I/O database relevant to the current I/O operation.

When a device interrupts, the driver's interrupt-servicing routine analyzes the interrupt, as detailed in Section 11 and summarized below:

- Identifies the address of the UCB of the device that generated the interrupt
- Obtains device-status or controller-status information from the device registers, if necessary, and stores the status information in the UCB
- Restores the driver's 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.

10 Writing Driver Code for DMA Transfers

A driver performing DMA transfers over the UNIBUS, Q22 bus, or MASSBUS must take I/O bus operation into consideration.[1] The VAX/VMS operating system and the I/O database manage the mapping registers and data path resources of the I/O adapter for device drivers.

The I/O database contains an adapter-control block (ADP) that describes the I/O adapter. This block contains allocation information for the mapping registers; for UNIBUS adapters, the ADP also contains similar information for data paths.

The ADP also contains the virtual address of the adapter's configuration register. All the adapter's other registers are located at fixed offsets from the configuration register. The VAX/VMS adapter-handling routines modify the adapter's mapping registers and data-path register according to requests from the driver's fork process.

In general, drivers' fork processes do not directly access the ADP. Instead, drivers call VAX/VMS routines that perform adapter-related services, such as the following:

- Allocating a buffered data path
- Allocating mapping registers
- Loading mapping registers
- Deallocating mapping registers
- Purging a buffered data path
- Deallocating a buffered data path

The critical responsibility of device drivers that actively compete for such shared I/O adapter resources as mapping registers and data paths is that they all execute at the same fork IPL. This IPL convention synchronizes access to the I/O adapter data structures.

The system creates a driver's fork process by calling the start-I/O routine in a device driver. The fork process takes some or all of the following steps to initiate an I/O transfer to or from a device on a UNIBUS, MicroVAX II Q22 bus, or MicroVAX I Q22 bus.

Operation	Applicable to
Requests buffered data path	UNIBUS
Requests mapping registers	UNIBUS, MicroVAX II
Loads mapping registers	UNIBUS, MicroVAX II
Calculates starting bus address	UNIBUS, MicroVAX II, MicroVAX I
Activates device	UNIBUS, MicroVAX II, MicroVAX I
Waits for interrupt	UNIBUS, MicroVAX II, MicroVAX I

[1] MASSBUS drivers are discussed in Appendix G.

When a hardware interrupt indicates that the I/O transfer is complete, the driver's fork process checks the success or failure of the transfer. The driver then concludes with the following steps:

Operation	Applicable to
Purges the data path	UNIBUS, MicroVAX II, MicroVAX I[1]
Releases the buffered data path	UNIBUS
Releases the mapping registers	UNIBUS, MicroVAX II

[1]Regardless of whether the associated processor provides buffered data paths or not, drivers of all devices should initiate a purge of the data path after a transfer. The purge operation enables the detection of memory parity errors that may have occurred during the transfer, as described in the sections on the PURDPR macro and IOC$PURGDATAP in Appendixes B and C, respectively.

Because of the different requirements of DMA transfers on different VAX processors, a driver must contain some run-time conditional code in order to function for equivalent UNIBUS, MicroVAX II, and MicroVAX I devices. Appendix E contains an example of one driver that supports the RL11 on the UNIBUS and the RLV11 on the MicroVAX I and MicroVAX II Q22 bus.

Regarding the material presented in this section, UNIBUS driver writers should read Sections 10.1 through 10.6.3. MicroVAX II driver writers should read Sections 10.1.3 and 10.2 through 10.6.3. MicroVAX I driver writers should turn directly to 10.7. Because the MicroVAX I provides no scatter-gather map, MicroVAX I device drivers must perform transfers according to the method described therein.

10.1 Selecting and Requesting a Data Path

DMA device drivers for certain VAX processors can elect to request the use of a UNIBUS adapter buffered data path to accelerate data transfers (as described in Section 4.3). Other VAX/VMS processors, such as the MicroVAX II and VAX–11/730, provide no buffered data paths for data transfers. The descriptions of the direct data path in the following sections apply to drivers written for devices on those processors.

10.1.1 Requesting a Buffered Data Path

Some VAX systems allow UNIBUS drivers to request temporary or permanent allocation of a buffered data path (see Table 4-1). 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$REQDATAP that locates the ADP. To do this, IOC$REQDATAP uses a series of pointers that begins in the current unit-control block (UCB), as follows:

UCB → CRB → ADP

IOC$REQDATAP performs the following services:[2]

1. Tests the path-lock bit (VEC$V_PATHLOCK) in the data-path number field of the channel-request block (CRB$L_INTD+VEC$B_DATAPATH). If the device has a permanent data path allocated to it, IOC$REQDATAP simply returns.
2. Determines which data paths are available by examining the data path allocation information in the ADP (ADP$W_DPBITMAP).
3. Allocates the first free data path to the driver by inserting its number in the data path field of the CRB (CRB$L_INTD+VEC$B_DATAPATH) and indicating in the ADP that the data path is in use (by setting the appropriate bit in ADP$W_DPBITMAP).
4. Returns control to the driver fork process.

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 UCB and the content of R5, in the ADP's 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.

When these conditions are met, the VAX/VMS routine IOC$RELDATAP allocates the data path to the suspended driver and reactivates the driver's fork process.

10.1.2 Requesting a Permanent Buffered Data Path

A device driver can permanently allocate a buffered data path with code in a unit-initialization routine. Instead of using the REQDPR macro, however, a unit-initialization routine should perform the following steps:

1. Test the path-lock bit (VEC$V_PATHLOCK) in the data-path-number field of the CRB (CRB$L_INTD+VEC$B_DATAPATH) to ensure that a data path is not already allocated for this device.
2. Call the subroutine IOC$REQDATAPNW to allocate the data path as shown below:

   ```
   JSB  G^IOC$REQDATAPNW
   ```

   If IOC$REQDATAPNW successfully allocates the data path, it stores the number of the data path it obtained in the CRB at VEC$B_DATAPATH and returns with the low-order bit set in R0 (SS$_NORMAL). If it cannot allocate a data path, IOC$REQDATAPNW does *not* create a fork process to wait for one to become available. Instead, it returns to the unit-initialization routine with the low-order bit clear in R0.
3. Set the path-lock bit (VEC$V_PATHLOCK) in the CRB at VEC$B_DATAPATH

[2] When called from a driver running on a processor that does not provide buffered data paths, IOC$REQDATAP and IOC$RELDATAP simply return after examining the data path bit map in the ADP.

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, which is one path for each controller that the driver serves.

Because some VAX processors have few buffered data paths (refer to Table 4-1), device drivers running on these processors must limit their allocation of permanent buffered data paths. For example, if the drivers loaded on a VAX–11/750 permanently allocated all three of the processor's buffered data paths, none would remain for normal system operations. As a result, I/O transfers requiring a buffered data path would wait forever.

10.1.3 Requesting the Direct Data Path

Because the UNIBUS adapter or other I/O interface arbitrates among devices that wish to use the direct data path and the data path field in the CRB (CRB$L_INTD+VEC$B_DATAPATH) 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 processors, such as the VAX–11/780, do not permit byte-offset transfers on the direct data path (see Table 4-1). Because the UNIBUS itself is word-oriented, a processor such as the VAX–11/780 must ensure that the data buffer is aligned on a word boundary for word-aligned devices.

10.1.4 Mixed Use of Direct and Buffered Data Paths

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 is completed, the driver should then purge and release the buffered data path. The release automatically resets the data path number to zero, which signifies a direct data path. When using the direct data path is complete, 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 Mapping Registers

The UNIBUS adapter and MicroVAX II processor logic allow UNIBUS and MicroVAX II Q22 bus drivers, respectively, to allocate mapping registers as needed or to allocate them permanently.

10.2.1 Allocating Mapping Registers

After the driver's fork process gains access to the controller (see Section 9.3.1), it requests a set of adapter mapping registers by invoking the VAX/VMS macro REQMPR. This macro calls the routine IOC$REQMAPREG. IOC$REQMAPREG calculates the number of mapping registers needed for a transfer. The calculation is based on the transfer byte count field and the byte offset fields of the device's UCB (UCB$W_BCNT and UCB$W_BOFF).

The procedure for allocating mapping registers is similar to that used to allocate a buffered data path. First, IOC$REQMAPREG locates the ADP from a series of pointers that begins with the current UCB, as follows:

UCB → CRB → ADP

Then, the routine examines the mapping-register-allocation information to locate the required number of contiguous mapping 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's address (same as UCB address and the contents of R5) in the mapping-register-wait queue.

When the mapping registers are available, IOC$REQMAPREG allocates them and adjusts the appropriate information about the allocation of mapping registers in the adapter-control block. IOC$REQMAPREG then writes the number of the first mapping register and the number of mapping registers allocated into the CRB and returns control to the driver's fork process.

10.2.2 Permanently Allocating Mapping Registers

A device driver can permanently allocate a set of mapping registers with code in the unit-initialization routine. The number of mapping registers permanently allocated must be sufficient for the longest possible transfer. The following steps permanently allocate a set of mapping registers:

1. Test the map-lock bit (VEC$V_MAPLOCK) in the CRB (CRB$L_INTD+VEC$W_MAPREG) to ensure that mapping registers are not already allocated for this device.
2. Load the number of mapping registers required into R3.
3. Call the VAX/VMS routine IOC$ALOUBAMAPN with a JSB instruction:

   ```
   JSB  G^IOC$ALOUBAMAPN
   ```

   If IOC$ALOUBAMAPN successfully allocates the mapping registers, it stores the number of mapping registers allocated and the number of the first of the allocated mapping registers. It stores these items in the CRB at CRB$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.
4. Set the map-lock bit in the CRB (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 routine contains the code described above, it permanently allocates one set of mapping registers for each CRB associated with the driver, which is one set of registers for each device controller that the driver serves.

10.3 Loading Mapping Registers

Once a driver's fork process has assigned a data path and allocated a set of mapping registers, it can request VAX/VMS to load the mapping registers with physical page-frame numbers (PFNs) by invoking the VAX/VMS macro LOADUBA.[3] LOADUBA calls a VAX/VMS routine IOC$LOADUBAMAP that loads each allocated mapping register with five data items:

- A bit setting to indicate whether the mapping 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$W_BOFF is a 1.
- The number of the data path to use for the transfer.
- The page-frame number of a page in memory.
- A bit setting to indicate that the transfer operates in longword-aligned, random-access mode on the buffered data path; this bit is set when VEC$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 mapping register, the page-frame number of the second page of the transfer into the second mapping register, and so forth.

IOC$LOADUBAMAP sets the valid bit in every allocated mapping register except the last. It clears the valid bit in the final mapping register to prevent a prefetch from an invalid page.

To calculate the page-frame number used in the I/O transfer, IOC$LOADUBAMAP uses three fields that VAX/VMS has written into the UCB:

- 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's number, the number of the first mapping register, the address of the first mapping register, and the number of allocated mapping registers from the CRB and the ADP, as follows:

UCB → CRB → number of the data path
UCB → CRB → number of first mapping register
UCB → CRB → ADP → virtual address of first mapping register
UCB → CRB → number of mapping registers

Drivers that handle UNIBUS byte-addressable devices call the routine IOC$LOADUBAMAPA. This routine performs the same function as IOC$LOADUBAMAP, with one exception. When IOC$LOADUBAMAPA loads mapping registers, it clears the byte-offset bit even if the transfer begins on an odd-byte address.

When IOC$LOADUBAMAP has loaded all the mapping registers and marked the last mapping register invalid, it returns control to the driver's fork process.

[3] MicroVAX II DMA driver writers also use the LOADUBA macro.

10.4 Computing the Starting Address of a Transfer

The driver fork process must calculate the starting address of a DMA transfer and load this address into the appropriate device register. MicroVAX I device drivers perform the procedure outlined in Section 10.7. UNIBUS and MicroVAX II Q22 bus drivers take the following five steps to make the calculation:[4]

1. Write the byte-offset-in-page field of the UCB (UCB$W_BOFF) into bits 0 through 8 of a general register.
2. Get the number of the starting mapping register for the transfer from a field in the CRB (CRB$L_INTD+VEC$W_MAPREG). Write bits 0 through 6 of this 9-bit value into bits 9 through 15 of the general register.
3. Write bits 0 through 15 of the general register into the device's buffer address register.
4. Write bits 7 and 8 of the mapping register number, acquired in step 2, into the extended memory bits of the appropriate device register (usually the control and status register (CSR)).[5]

10.5 Activating the Device

Because a driver's fork process can address device registers as though they were any other virtual address, the loading of the device buffer address register and CSR are simple procedures. The driver locates the CSR address of the device in the interrupt-dispatch block (IDB), 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 CSR is the first device register and the device's word-count register is the third device register, the device driver can describe the device register offsets and load the word-count register with the following series of instructions:

```
DEV_CSR =   0
DEV_XREG =  2
DEV_WDCNT = 4
    .
    .
    .
; Compute word count of transfer and store it in user-defined UCB field,
; UCB$W_WDCNT.
    .
    .
    .
        MOVL    UCB$L_CRB(R5),R4                ;Address of CRB
        MOVL    @CRB$L_INTD+VEC$L_IDB(R4),R4    ;Address of CSR
        MOVW    UCB$W_WDCNT,DEV_WDCNT(R4)       ;Move word count to device word
                                                ;count register
```

[4] Although the MicroVAX II processor actually contains 8192 mapping registers, VAX/VMS currently enables only 496 of them. As a result, the upper four bits of the 13-bit MicroVAX II mapping register number should be stored as zero. In other words, VAX/VMS treats any mapping register number as having nine significant bits.

[5] One example of a device that does not treat the extended memory bits in this fashion is the DRV11-WA, the code for which is listed in Appendix F. For the DRV11-WA, code in XADRIVER stores bits 7 and 8 of the mapping register number in a discrete device bus address extension register, then clears the extended address bits of the device's CSR. In contrast, XADRIVER handles the DR11-W according to the method described above.

10.6 Completing a DMA Transfer

After a UNIBUS, MicroVAX II, or MicroVAX I device driver's fork process activates a DMA device, the driver waits for a device interrupt by invoking a VAX/VMS macro that suspends execution of the driver. When the device requests a hardware interrupt, the interrupt dispatcher gains control.

The dispatcher saves R0 through R5 and transfers control to the driver's interrupt-servicing routine. If the interrupt-servicing routine can match the interrupt with a suspended driver's fork process, the interrupt-servicing routine reactivates the driver's fork process at the point where execution was suspended. Most drivers almost immediately invoke 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-servicing routine, which dismisses the interrupt.

When the fork dispatcher reactivates the driver's fork process, the driver performs any necessary clean up operations, such as purging the data path and deallocating adapter resources used in the DMA transfer.

10.6.1 Purging the Data Path

Driver fork processes must purge the data path after the DMA transfer is complete. This is true for devices with buffered data paths, direct data paths, or no data path.

To purge the data path, the driver invokes the macro PURDPR, which in turn calls the VAX/VMS routine IOC$PURGDATAP. This routine takes the following steps to purge the data path:

1. Saves the contents of R4 on the stack.
2. Locates the CRB as follows:

   R5 → UCB → CRB

3. Obtains the starting address of UNIBUS adapter register space and stores it in R2.
4. Extracts the number of the data path to be purged from the CRB and loads it into R1.
5. Stores the address of the data path in R4.
6. Instructs the UNIBUS adapter or Q22 bus interface 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 adapter mapping register.

The address of the CRB 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.

7 Restores the information stored on the stack to R4 and returns to the address in the driver immediately after the invocation of the PURDPR macro.

8 Some machine implementations also check for memory errors that might have occurred during the DMA operation, and, if an error is detected, log it.

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's 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 CRB. IOC$RELDATAP locates it as follows:

UCB → CRB → number of the data path

If the data path is permanently assigned to a device, IOC$RELDATAP does not release the data path. Otherwise, the data path number in the CRB (CRB$L_INTD+VEC$B_DATAPATH) is zeroed. The IOC$RELDATAP routine attempts to dequeue a waiting driver fork process from the data-path-wait queue. It finds the queue as follows:

UCB → CRB → ADP → data-path-wait queue

If another driver is waiting for a buffered data path, IOC$RELDATAP grants that driver fork process the data path, restores its context from its UCB fork block, and transfers control to the saved driver PC. When IOC$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's fork process.

If the data-path-wait queue is empty, IOC$RELDATAP marks the data path as available in the ADP and returns control to the driver.

10.6.3 Releasing Mapping Registers

A driver fork process releases a set of mapping registers by invoking the VAX/VMS macro RELMPR at fork IPL. RELMPR calls the VAX/VMS routine IOC$RELMAPREG, which releases mapping registers in a manner similar to the way in which the RELDPR macro releases data paths. The CRB records the number of mapping registers assigned to the device. The number of the first mapping register and the number of mapping registers are located as follows.

UCB → CRB → number of the first mapping register
UCB → CRB → number of allocated mapping registers

IOC$RELMAPREG releases the mapping registers by adjusting the mapping-register-allocation information in the ADP.

Then, IOC$RELMAPREG attempts to dequeue a driver's fork process from the mapping-register-wait queue. If a suspended driver is found, IOC$RELMAPREG takes the following steps:

1. Dequeues the fork block and restores driver context
2. Satisfies the mapping-register request, if possible
3. Reactivates the driver's fork process at the instruction following the driver's request for mapping registers
4. Repeats Steps 1 through 3

If the mapping-register-wait queue is empty or if IOC$RELMAPREG still does not have enough contiguous mapping registers for any of the waiting fork processes, it returns control to the fork process that released the mapping registers.

10.7 Considerations for MicroVAX I DMA Devices

Because the MicroVAX I does not provide a scatter-gather map, MicroVAX I Q22 bus DMA devices must use a physically contiguous buffer in data transfers. Because there is no guarantee that this is the state of the user's buffer, the driver must allocate an intermediate buffer consisting of contiguous physical pages. The driver never deallocates this buffer unless the driver is being unloaded (by means of SYSGEN's RELOAD command). The best time to allocate such a buffer is during the device's initialization. Memory is most likely contiguous at that time. Later it will be much more difficult to obtain a buffer that contains physically contiguous pages.

To be sure that the buffer you allocate to the driver is contiguous, use the VAX/VMS routine EXE$ALOPHYCNTG, described in Appendix C. The size of the buffer will depend on the device's characteristics and the size of the transfers requested on the device. A buffer of four pages is likely to be large enough for most disk transfers, for example; but if you have enough memory on your system, you might want to make your buffer the size of a disk track in order to reduce disk latency. In any event, large transfers to the device can be segmented into transfers the size of your intermediate buffer.

When a user requests a transfer to a MicroVAX I Q22 bus device, the driver start-I/O routine copies the data from the user's buffer into the intermediate, physically contiguous buffer by means of the routine IOC$MOVFRUSER. The driver must ensure that the buffer is word-aligned because the MicroVAX I has no byte-offset capability.

The driver then sets up the device for the DMA transfer:

1. Determines the 22-bit physical address of the buffer from the system virtual address returned by EXE$ALOPHYCNTG. Presuming that the virtual address has been temporarily stored in CRB$L_AUXSTRUC, the driver can use code similar to the following excerpt from DLDRIVER (in Appendix E).

```
        MOVL    UCB$L_CRB(R5),R1        ;GET CRB ADDRESS
        MOVL    CRB$L_AUXSTRUC(R1),R2   ;MEMORY ALLOC FAILURE DURING CTL INIT?
        BEQL    70$                     ;IF EQL, YES, LEAVE OFFLINE
        MOVL    R2,UCB$A_DL_BUF_VA(R5)  ;SAVE BUFFER'S VIRTUAL ADDRESS
        EXTZV   #VA$V_VPN,#VA$S_VPN,R2,R1;GET VIRTUAL PAGE NUMBER OF BUFFER
        MOVL    G^MMG$GL_SPTBASE,R0     ;GET BASE ADDRESS OF SPTS
        MOVL    (R0)[R1],R0             ;GET THE PTE CONTENTS
        BICL3   #^C<VA$M_BYTE>,R2,R1    ;GET BUFFER OFFSET (BA00-BA08)
        ASSUME  PTE$S_PFN GE 13
        INSV    R0,#9,#13,R1            ;COPY BA09-BA21
        MOVL    R1,UCB$A_DL_BUF_PA(R5)  ;SAVE PHYSICAL ADDRESS OF BUFFER
        .
        .
        .
70$:    RSB
```

2 Moves the low word (bits 0 to 15) of the buffer physical address into the device's buffer address register.

3 Moves the extended address bits of the buffer's physical address into the device's extended address register or the device's CSR, as required by the device.

4 Activates the device as described in Section 10.5.

5 If the transfer size exceeds the size of the buffer, returns to Step 1.

When a user requests a transfer from a MicroVAX I Q22 bus device, the driver moves the data from the device to the intermediate, physically contiguous buffer by means of a DMA transfer, then calls IOC$MOVTOUSER to copy the data into the user's buffer.

A MicroVAX I driver should complete the transfer as described in Section 10.6. The driver should call IOC$PURGDATAP in order to detect and log any memory errors that might have occurred during the transfer.

11 Writing an Interrupt-Servicing Routine

For most device drivers, the driver-prologue table contains, in the reinitialization section established by the DPT_STORE macro, the address of one or more interrupt-servicing routines. Each interrupt-servicing routine corresponds to an interrupt vector on the I/O bus. You specify the address of an I/O bus vector using the SYSGEN command CONNECT, as described in Section 14.2.2.

Most device interrupt-servicing routines perform the following functions:

- Locate the device's UCB
- 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

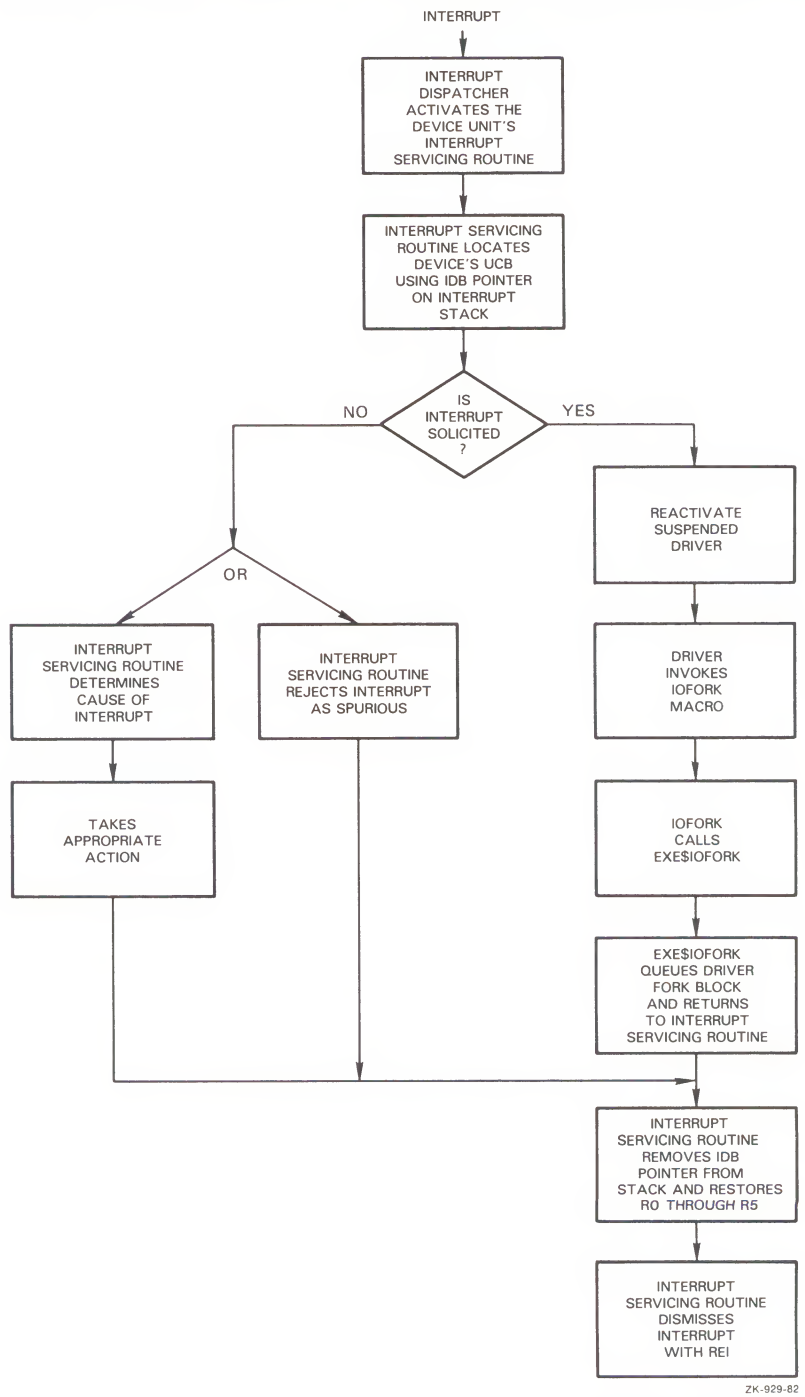
When a device generates a hardware interrupt, the device requests the interrupt at its device IPL. The UNIBUS adapter or MicroVAX Q22 bus interface then requests a processor interrupt at that IPL. When the processor executes at an interrupt priority level below the device IPL, interrupt dispatching begins.

Note: The subsequent discussion applies to UNIBUS and MicroVAX device interrupts exclusively. MASSBUS adapter interrupt dispatching differs substantially from UNIBUS and MicroVAX interrupt dispatching. MASSBUS driver writers should familiarize themselves with the discussion in Sections G.4 and G.6.

On a configuration that uses nondirect vector interrupts—such as the VAX–11/780 and the VAX 8600—the following sequence occurs:

1. The processor saves, on the interrupt stack, the PC and PSL of the currently executing code. It dispatches the interrupt by means of the appropriate vector in the system control block (SCB) to the interrupt-servicing routine for the UNIBUS adapter of the device that requested the interrupt (see Section 3.1.5).
2. The UNIBUS adapter's interrupt-servicing 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's address to the UNIBUS adapter's interrupt-servicing routine.
3. The UNIBUS adapter's interrupt-servicing 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 (CRB).

Figure 11–1 Flow of Interrupt Servicing



4 The CRB's 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—such as the MicroVAX I, MicroVAX II, VAX 8200, VAX 8800, VAX-11/750, and VAX-11/730—the following sequence occurs:

1 The processor saves, on the interrupt stack, the PC and PSL of the currently executing code and acknowledges the device's interrupt.

2 The device supplies its vector address, which the processor uses as an index into a table in the second (or third) page of the SCB (see Section 3.1.5). This table contains a list of addresses in the CRB that point to the interrupt-servicing routines for devices attached to the first UNIBUS or an optional second UNIBUS (for the VAX-11/750).

3 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 CRB.

4 The CRB's interrupt-dispatching field (CRB$L_INTD) contains executable code that the driver-loading procedure has associated with the interrupt 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 address of the interrupt-servicing routine for each interrupt-dispatching field from the reinitialization portion of the driver-prologue table (see Section 7.1). This section of the DPT contains one or more DPT_STORE macros that identify the addresses of the interrupt-servicing routines. The number of DPT_STORE macros that identify interrupt-servicing routines must equal the number of vectors given in the /NUMVEC qualifier to avoid errors in device initialization or interrupt handling.

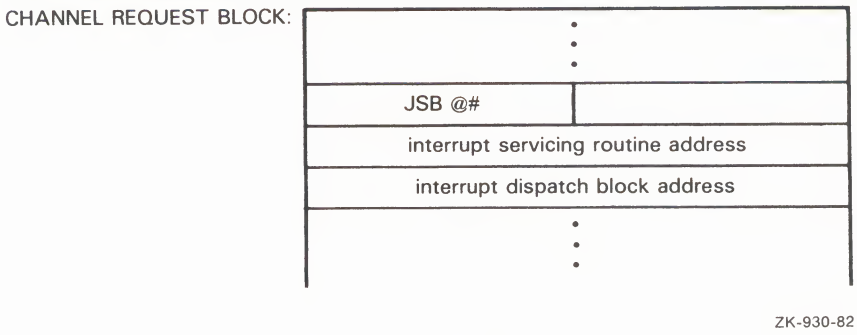
Immediately following the JSB instruction in the CRB is the address of the interrupt-dispatch block (IDB) associated with the CRB. When the JSB instruction executes, a pointer to the address of the IDB is pushed onto the top of the stack as though it were a return address. The driver interrupt-servicing routine can use this IDB address as a pointer into the I/O database. Figure 11–2 illustrates the portion of a CRB that contains the address of the interrupt-servicing routine.

11.2 Interrupt Context

When the interrupt dispatcher calls a driver's interrupt-servicing routine, execution context is as follows:

- R0 through R5 are saved on the stack.
- System address space is mapped. The interrupt-servicing routine can gain access to appropriate data structures in the I/O database.

Figure 11–2 CRB Containing the Address of an Interrupt-Servicing Routine



- 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 IDB
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

11.3 Servicing a Solicited Interrupt

When a driver's fork process activates a device and expects to service a device interrupt as a result, the fork process suspends its execution and waits for an interrupt to occur. The suspended driver is represented only by the contents of the device's UCB, which contains a description of the I/O request and the fork process.

When the driver regains control from the interrupt-servicing routine, only R3, R4, R5, and the PC address are restored to their previous state by the interrupt-servicing routine.

In the sequence below, a driver's interrupt-servicing routine returns control to the waiting driver:

1 The interrupt-servicing routine obtains the address of the device's UCB from the IDB, as follows:

0(SP) → CRB → IDB → IDB$L_OWNER → UCB

2 The interrupt-servicing routine then tests the software-interrupt-expected bit in the UCB status word (UCB$V_INT in UCB$L_STS). If the bit is set, the driver is waiting for an interrupt from this device. The interrupt-servicing routine then clears UCB$V_INT in UCB$L_STS to indicate that it has received the expected interrupt.

Note: Because device timeout processing mostly occurs at fork IPL (see Section 12.2), a driver's interrupt-servicing routine, executing at device IPL, could interrupt the processing of a timeout on the same device unit. For this reason, the driver's interrupt-servicing routine should check the interrupt-expected bit (UCB$V_INT) before handling the interrupt. VAX/VMS clears this bit before it calls the driver's timeout handler.

3 The interrupt-servicing routine restores R5 of the driver's fork process, placing in it the address of the UCB fork block. It restores R3 and R4 of the driver process by placing in them the contents of UCB$L_FR3 and UCB$L_FR4, respectively.

4 The interrupt-servicing routine transfers control to the driver's PC address, which is 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 IPL from device level to fork level. The driver 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 into the appropriate fork queue the UCB fork block that describes the driver process. Then it returns control to the driver's interrupt-servicing routine.

The interrupt-servicing routine then performs the following steps:

1 Removes the IDB pointer from the stack

2 Restores R0 through R5

3 Dismisses the interrupt with an REI instruction

11.4 Servicing an Unsolicited Interrupt

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

Other changes in the device's status occur when the device has not been activated by a device driver. The device reports 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 handled 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 UCB.

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

- Ignore the interrupt as spurious

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

The driver's interrupt-servicing routine determines whether an interrupt is solicited or not by examining the software-interrupt-expected bit in the UCB status word (UCB$V_INT in UCB$L_STS). All UNIBUS and Q22 bus device drivers must use this method to determine whether or not an interrupt is solicited; the address of the unsolicited-interrupt routine, specified in the driver-dispatch table, is used only by MASSBUS drivers (see Sections G.4 and G.6.)

If the interrupt is unsolicited, the driver can reject the interrupt with the following code sequence:

1. Remove the IDB pointer from the stack
2. Restore R0 through R5
3. Dismiss the interrupt with an REI instruction

Rather than rejecting the interrupt, the driver might wish to handle it. For example, the driver can send a message to the operator or the job controller's mailbox when an unsolicited interrupt occurs.

Drivers should always handle unsolicited interrupts from busy devices at device IPL. If a driver must create a fork process to handle such an interrupt, it should use extreme caution. The UCB of a busy device might contain the active fork block of a previously created driver fork process. If a routine servicing an unsolicited interrupt creates a fork process to handle its interrupt, it can destroy the fork context currently stored in that UCB.

Because only one sequence of instructions can use the UCB as a fork block, the interrupt-servicing routine must perform the following steps before it can create the fork process:

- Ensure that no one is using the device, and that no one wants to use it, by determining that the reference count (UCB$W_REFC) is zero.
- Ensure that it is not already using the UCB, to create a fork process in order to lower IPL and to send a message to the job controller, by testing the job-attached bit (UCB$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-servicing routine.

11.4.1 Examples of Unsolicited Interrupts

A card reader requests an unsolicited interrupt when a user turns the reader on line. Once the card-reader driver's interrupt-servicing routine determines that the interrupt is unsolicited, the routine analyzes the interrupt, as in the following code example:

```
CR$INT::
        MOVL    @(SP)+,R3               ;GET ADDRESS OF IDB❶
        MOVQ    IDB$L_CSR(R3),R4        ;GET CONTROLLER CSR AND OWNER UCB ADDRESS❷
        BBCC    #UCB$V_INT,UCB$L_STS(R5),10$ ;IF CLR, INTERRUPT NOT EXPECTED❸
        .
        .
        .
;
; UNSOLICITED INTERRUPT
;
10$:    MOVZWL  CR_CSR(R4),R0           ;GET READER STATUS
        MOVZBW  #CR_CSR_M_IE,CR_CSR(R4) ;CLEAR STATUS, ENABLE INTERRUPTS❹
        BITW    #CR_CSR_M_ONLINE,R0     ;READER TRANSITION TO ONLINE?❺
        BEQL    20$                     ;IF EQL NO
        TSTW    UCB$W_REFC(R5)          ;DEVICE ASSIGNED OR ALLOCATED?❻
        BNEQ    20$                     ;IF NEQ YES
        BBSS    #UCB$V_JOB,UCB$W_DEVSTS(R5),20$ ;IF SET, MESSAGE ALREADY SENT❼
        BSBB    30$                     ;SEND MESSAGE TO JOB CONTROLLER
20$:    MOVQ    (SP)+,R0                ;RESTORE REGISTERS
        MOVQ    (SP)+,R2
        MOVQ    (SP)+,R4
        REI
30$:    JSB     G^EXE$FORK              ;CREATE FORK PROCESS❽
        MOVZBL  #MSG$_CRUNSOLIC,R4      ;SET MESSAGE TYPE❾
        MOVAB   G^SYS$GL_JOBCTLMB,R3    ;SET ADDRESS OF JOB CONTROLLER MAILBOX
        JSB     G^EXE$SNDEVMSG          ;SENT MESSAGE TO JOB CONTROLLER
        BLBS    R0,40$                  ;IF LBS SUCCESSFUL NOTIFICATION❿
        BICW    #UCB$M_JOB,UCB$W_DEVSTS(R5) ;CLEAR MESSAGE SENT BIT⓫
40$:    RSB
```

❶ The interrupt-servicing routine obtains the address of the IDB from the top of the stack.

❷ By means of this address, it obtains the address of the control and status register (CSR).[1]

❸ It checks for an unsolicited interrupt by testing the interrupt enable bit in the UCB status word.

❹ Because the interrupt is unsolicited, the routine clears all CSR bits except for the interrupt enable bit.

❺ It confirms that the reader was just placed on line by examining a saved copy of the CSR.

❻ It examines the reference count field of the device's UCB (UCB$W_REFC) to determine whether a process has allocated the device or assigned a channel to it.

❼ If the reference count is zero, the interrupt-servicing 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 on line. By using the job-attached bit to synchronize message sending, the interrupt-servicing routine protects the send-message-to-job-controller function from the adverse effects of frequent online interrupts.

❽ 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, using the system routine EXE$SNDEVMSG (described in Appendix C). It is necessary to lower IPL from device IPL by forking at this point because EXE$SNDEVMSG expects its caller's IPL to be no greater than IPL$_MAILBOX.

[1] Because the card reader has a dedicated controller, the IDB$L_OWNER field always points to the UCB for the single unit: 0(SP) → CRB → IDB → IDB$L_OWNER → UCB

When the interrupt-servicing routine regains control, it restores R0 through R5 and dismisses the interrupt with an REI instruction. (The interrupt-servicing routine removed the IDB pointer from the stack earlier in its execution in order to obtain CSR and UCB addresses.)

❾ When the fork process created at Step 8 above eventually executes, it writes a message to the job controller's mailbox, indicating that the card reader is on line.

❿ 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's cancel-I/O routine later clears the bit.)

⓫ 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.

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-servicing routine must determine what drive unit requested the interrupt, obtain status information from the drive's CSR, and then decide whether the interrupt was solicited.

If the interrupt is unexpected, the driver's interrupt-servicing routine calls its unsolicited-interrupt-servicing routine. The routine checks the status of the volume, as described in the following steps:

1 It sets a bit in the UCB to indicate that the unit is on line (UCB$V_ONLINE in UCB$L_STS).

2 If the UCB's volume-valid bit is set (UCB$V_VALID in UCB$L_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.

3 The routine returns control to the driver's interrupt-servicing routine.

The driver's interrupt-servicing 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.

12 Completing an I/O Request and Handling Timeouts

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

If the device requests an interrupt, the driver interrupt-servicing 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 the **excpt** argument to the wait-for-interrupt macro.

12.1 I/O Postprocessing

Once the driver interrupt-servicing 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-servicing routine to the context of a driver fork process in the following steps:

1. It disables software timeouts by clearing the timeout enable bit in the UCB status word (UCB$V_TIM in UCB$L_STS).
2. It saves R3 and R4 of the current driver context in the UCB fork block (UCB$L_FR3 and UCB$L_FR4).
3. It saves the current driver PC in the UCB fork block (UCB$L_FPC). (The driver PC is the top longword on the stack, as a *result of the JSB to* EXE$IOFORK.)
4. It obtains the fork IPL of the device from the UCB (UCB$B_FIPL).
5. It inserts the address of the UCB fork block (R5) into the fork queue corresponding to the driver's fork IPL.
6. Finally, if the fork block is the first entry in the fork queue, EXE$IOFORK requests a software interrupt at the driver's fork IPL.

The steps listed above move the fork process' context into the UCB's fork block. They save R3 through R5 and the driver's PC address. The driver's fork process resumes processing when the VAX/VMS fork dispatcher dequeues the UCB fork block from the fork queue and reactivates the driver at the driver's fork IPL.

12.1.2 Completing an I/O Request

When VAX/VMS reactivates a driver's 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's fork process for a DMA device proceeds as follows:

1. Purges the data path
2. Releases the buffered data path (applies only to UNIBUS DMA device drivers)
3. Releases mapping registers (does not apply to MicroVAX I DMA device drivers)
4. Releases the controller (applies only to drivers of devices on multiunit controllers)
5. 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
6. Returns control to the operating system

Section 10 discusses the first three steps listed above because they relate to DMA transfers. The sections that follow describe the last 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's fork process the channel, restores its context from the UCB fork block, and transfers control to the saved PC. When no more drivers are awaiting the channel, IOC$RELCHAN returns control to the fork process that released the channel.

Drivers for devices with dedicated controllers need not release the controller's data channel (as discussed in Sections 9.3.1 and 13.1). By means of code in the unit-initialization routine, these drivers set up the device's UCB so that the device owns the controller permanently.

Drivers must be executing at driver's fork IPL when they 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:

1. Loads a success status code (SS$_NORMAL) into bits 0 through 15 of R0.
2. Loads the number of bytes transferred into the high-order 16 bits of R0 (bits 16 through 31), if the I/O operation performed by the device is a transfer function.
3. Loads device-dependent status information, if any, into R1.[1]

12.1.2.3 Returning Control to the Operating System

Finally, the driver returns control to the system by invoking the REQCOM macro 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 (IRP) corresponding to the I/O operation in the device's UCB (UCB$L_IRP). It then writes the two longwords of completion status contained in R0 and R1 into the media field of the IRP (IRP$L_MEDIA and IRP$L_MEDIA+4).

IOC$REQCOM then inserts the IRP in the I/O-postprocessing queue. If the packet is the only entry in the postprocessing queue, IOC$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 UCB (UCB$V_ERLOGIP in UCB$L_STS), IOC$REQCOM obtains the address of the error message buffer from the UCB (UCB$L_EMB). It then writes the following information into the error buffer:

- Final device status (UCB$W_DEVSTS)
- Final error count (UCB$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 IRPs are waiting for driver processing IOC$REQCOM dequeues an IRP from the head of the queue of packets waiting for the device unit (UCB$L_IOQFL), and transfers control to IOC$INITIATE. IOC$INITIATE proceeds to create a new driver fork process for the device unit and activate the driver's start-I/O routine, as described in Section 5.2.1.

Otherwise, IOC$REQCOM clears the unit-busy bit in the device's UCB status word (UCB$V_BSY in UCB$L_STS) and transfers control to IOC$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.

[1] 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 $QIO system service. If the user specifies no I/O-status block, VAX/VMS does not use R0 and R1.

12.2 Timeout Handling Routines

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 invocation of the wait-for-interrupt macro. Among its other activities, the VAX/VMS IPL$_TIMERFORK interrupt-servicing routine, having raised IPL to IPL$_SYNCH, scans UCBs once every second to determine whether a device has timed out.

When the IPL$_TIMERFORK interrupt-servicing routine locates a device that has timed out, the routine calls the driver's timeout handler by performing the following steps:

1. It disables expected interrupts and timeouts on the device by clearing bits in the status field of the device's UCB (UCB$V_INT and UCB$V_TIM in UCB$L_STS).
2. It sets the device-timeout bit in the UCB status field (UCB$V_TIMOUT in UCB$L_STS).
3. It sets IPL to hardware device interrupt IPL (UCB$B_DIPL).
4. It restores the saved R3 and R4 of the driver's fork process from the UCB fork block (UCB$L_FR3 and UCB$L_FR4).
5. It restores R5 (address of the UCB fork block).
6. It computes the address of the driver's timeout handler from the saved PC in the UCB fork block (UCB$L_FPC).
7. 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.

Because VAX/VMS originally invoked the timeout handler through an interrupt at IPL$_TIMERFORK, the driver can lower IPL from device IPL to the driver's fork IPL to process the timeout.[2]

Note: The driver should lower IPL with SETIPL to preserve the contents of the stack.

When the driver's fork process regains control, R3 and R4 are restored from UCB$L_FR3 and UCB$L_FR4 to their previous state.

[2] Because the device can interrupt device-timeout processing at fork IPL, the driver's interrupt-servicing 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.

During recovery from a power failure, VAX/VMS forces a device timeout by altering the timeout field (UCB$L_DUETIM) of a UCB if that device's UCB records that the unit is waiting for an interrupt or timeout (UCB$V_INT and UCB$V_TIM set in UCB$L_STS). The timeout handler can perceive that recovery from a power failure is occurring by examining the power bit (UCB$V_POWER in UCB$L_STS) in the UCB.

A timeout handler usually performs one of three functions:

- It retries the I/O operation unless a retry count is exhausted.
- It aborts the I/O request.
- It sends a message to an operator mailbox and resumes waiting for a subsequent interrupt or timeout.

12.2.1 Retrying an I/O Operation

Some devices might retry an I/O operation after a timeout. For example, a disk driver might take the following steps after a transfer timeout:

1 Invoke the following VAX/VMS macro to lower IPL to fork level:

```
SETIPL UCB$B_FIPL(R5)
```

The resulting IPL must not drop below IPL$_SYNCH.

2 Release mapping registers, data path, and controller data channel.

3 Perform one of the following actions depending upon the occurrence of a power failure:

- If a power failure occurred, load the address of the IRP into R3, reload the following fields of the IRP into the corresponding UCB fields, and branch to the start-I/O routine:

  IRP$L_BCNT (low-order word)
  IRP$W_BOFF
  IRP$L_SVAPTE

  This results in a retry of the transfer from the beginning.

- If no power failure has occurred and the device driver supports error logging (see Section 7.2), call ERL$DEVICTMO to log the device timeout.

4 Perform one of the following actions according to the error retry count:

- If the retry count is not exhausted, decrease the count, clear the UCB timeout bit in UCB$L_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 an 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$V_CANCEL in UCB$L_STS) is set, a cancel-I/O request was made 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:

1. Clear the device control and status register (CSR), if appropriate to the device and controller.
2. Invoke the following VAX/VMS macro to lower IPL to fork level:

   ```
   SETIPL UCB$B_FIPL(R5)
   ```

   The resulting IPL must not drop below IPL$_SYNCH.
3. Release mapping registers, data path, and controller data channel.
4. Load the abort status code (SS$_ABORT) into the low word of R0.
5. Clear bits 16 through 31 in R0 to indicate that no data was transferred.
6. Invoke the VAX/VMS macro REQCOM described in Section 12.1.2.3 to complete the processing of the I/O request.

12.2.3 Sending a Message to the Operator

The following sequence describes a timeout handler that sends a message to the operator's mailbox and then goes back into a wait-for-interrupt or timeout state:

1. The timeout handler 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$_SYNCH.
2. It checks the cancel-I/O bit in the UCB status word (UCB$V_CANCEL in UCB$L_STS).

   If UCB$V_CANCEL is set, the timeout handler can abort the request. However, if UCB$V_CANCEL is clear, the timeout handler does the following:

   a. Saves R3 and R4 on the stack
   b. Loads an OPCOM message code, such as MSG$_DEVOFFLIN, into R4
   c. Loads the address of the operator's mailbox (SYS$GL_OPRMBX) into R3
   d. Calls a VAX/VMS routine to place the message in the operator's mailbox, as follows:

      ```
      JSB   G^EXE$SNDEVMSG
      ```

   e. Restores R3 and R4

3 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.

4 Finally, it invokes the WFIKPCH macro 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.

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 (by repeatedly invoking the DPT_STORE macro described in Section 7.1.2) specifies the addresses of the unit- and controller-initialization routines.[1] The driver-dispatch table (DDT) contains the addresses of the cancel-I/O and error-logging routines. The type of device determines which of these routines are required in a driver.

13.1 Initialization Routines

Most device controllers and device units require initialization both when the corresponding device driver is loaded and when the operating system is recovering from a power failure.

At these times, the duty of initialization routines is to prepare, according to their characteristics, controllers and device units for operation. Among the actions of initialization routines for typical controllers and devices are:

- Enabling controller interrupts
- Clearing the error-status bits in device registers
- Initiating a device operation, such as clearing a drive or acknowledging a disk pack
- Storing values in UCB fields that the DPT_STORE macro cannot reach The DPT_STORE macro can initialize only the first 256 bytes of a data structure.
- Permanently allocating data paths and mapping registers, as necessary, according to the methods described in Section 10.
- Setting the online bit (UCB$V_ONLINE in UCB$L_STS) in the UCB.
- Filling in the IDB$L_OWNER field for single-unit devices such as line printers.

[1] A MASSBUS device driver must specify the address of its unit-initialization routine in the driver-dispatch table (using the **unitinit** argument to the DDTAB macro as discussed in Section 7.2). UNIBUS and Q22 bus drivers can specify the address in either the driver-prologue table or driver-dispatch table.

13.1.1 Initialization During Driver Loading

The extent of initialization needed during driver loading depends upon whether the driver is being loaded for the first time or is replacing a driver that was previously loaded.

The SYSGEN commands AUTOCONFIGURE, CONNECT, and LOAD add new drivers to the system 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 perform the following steps:

- Create a UCB for the device. If this is the first occurrence of device-name and controller, the commands create a device-data block (DDB), a channel-request block (CRB), and an interrupt-dispatch block (IDB).
- 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 DDT and FDT to system virtual addresses.
- Call the controller-initialization routine specified in the CRB, if the CRB was created.
- Call the unit-initialization routine (if any) specified in the DDT. 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's code into nonpaged system memory. Unlike the other SYSGEN commands for driver loading, RELOAD assumes that the data structures associated with the driver already exist, and thus updates the I/O database 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 FDT and DDT to system virtual addresses
- Calls the controller-initialization routine

Section 14 contains detailed descriptions of all SYSGEN commands related to device drivers.

13.1.2 Initialization During Recovery from a Power Failure

During recovery from a power failure, the operating system locates every UCB in the I/O database, by means of following the chain of pointers to all DDBs in the system (starting at IOC$GL_DEVLIST and chained by DDB$L_LINK) and the chain of pointers to all UCBs of the same device and controller type (starting at DDB$L_UCB and chained by UCB$L_LINK). For each UCB it finds, VAX/VMS performs the following procedure:

1. It locates the CRB associated with the UCB (UCB$L_CRB) and determines whether a controller initialization routine exists for the device's controller by examining CRB$L_INTD+VEC$L_INITIAL. If an invocation of the DPT_STORE macro loaded the address of a controller-initialization routine into this field, VAX/VMS calls that routine.
2. It determines whether a unit-initialization routine exists for the particular device unit by examining the unit-initialization field of the DDT (DDT$L_UNITINIT). If the field does not contain an address, the system checks the CRB (CRB$L_INTD+VEC$L_UNITINIT).[2]

   If either the CRB or the DDT 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 DDT contains an address, the address in the CRB is ignored.

13.1.3 Context of an Initialization Routine

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 controller-initialization routine for a card reader might enable interrupts from the device by setting the interrupt-enable bit in the device's control and status register (CSR). A disk's 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 CSR
R5	Address of IDB that describes the controller
R6	Address of DDB associated with the controller
R8	Address of CRB for the controller

Unit-initialization routines are useful for initializing device-dependent fields in the UCB. For example, unit-initialization routines for disks can also specify disk-drive geometry (such as number of cylinders) in the UCB and wait for online units to spin up to speed. Unit-initialization routines must set the online bit in the UCB (UCB$V_ONLINE) to declare the unit to be on line.

[2] MASSBUS drivers store unit-initialization routines addresses only in the DDT.

If a device needs permanently allocated I/O adapter resources, a unit-initialization routine can call VAX/VMS routines to allocate the resources. Then, the initialization routine can set bits in the CRB's I/O adapter resource-description fields (for example, VEC$V_MAPLOCK in CRB$L_INTD+VEC$W_MAPREG and VEC$V_PATHLOCK in CRB$L_INTD+VEC$B_DATAPATH.)

At the time of a call to a unit-initialization routine, the registers contain the following values:

Register	Value
R3	Address of primary CSR
R4	Address of secondary CSR; R4 is equal to R3 if there is no secondary CSR
R5	Address of the device's UCB

If a driver's 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 a device driver's cancel-I/O routine under the following circumstances:

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

The VAX/VMS routine EXE$CANCEL locates the UCB for the device associated with a process I/O channel from a pointer in the CRB, as follows:

channel index number → CCB → UCB

EXE$CANCEL performs the following steps:

1. Raises IPL to fork IPL (UCB$B_FIPL)
2. Removes from the device's pending I/O queue all IRPs associated with the process
3. Sets the status code SS$_CANCEL in IRP$L_MEDIA
4. For a buffered-I/O read operation, clears the buffered-read function bit (IRP$V_FUNC) in IRP$W_STS
5. Inserts the IRPs removed from the pending I/O queue into the I/O postprocessing queue

6 If the I/O-postprocessing queue is empty, requests a software interrupt at IPL$_IOPOST

7 Calls the cancel-I/O routine specified in the DDT of the associated device driver (argument **cancel** to the DDTAB macro). EXE$CANCEL locates the routine using the following chain of pointers:

UCB → DDT → 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 UCB registers at the time of the call contain the following values:

Register	Value	
R2	Channel index number	
R3	Address of current IRP	
R4	Address of process-control block (PCB) of process for which the $CANCEL system service is being performed	
R5	Address of device's UCB	
R8	Reason for call to cancel the I/O request. Codes that signify the reasons for cancellation 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 might want to base their cancel-I/O operation on whether the cancel-I/O request is the result of a channel deassignment (CAN$C_DASSGN). For example, the terminal driver cancels out-of-band AST requests only if the call to its cancel-I/O routine results from a Deassign-I/O-Channel ($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$RETURN into the appropriate position in the DDT. The routine IOC$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:

1. It confirms that the device is busy by examining the device-busy bit in the UCB status word (UCB$V_BSY in UCB$L_STS).
2. It locates the process-identification field in the IRP currently being processed on the device by 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$L_PID).
3. It confirms that the specified channel-index number is the same as the value stored in the IRP's channel-index field (IRP$W_CHAN).
4. It sets the cancel-I/O bit in the UCB status word (UCB$V_CANCEL in UCB$L_STS). Other driver routines, such as the timeout-handling routine, check the cancel-I/O bit to determine whether to retry the I/O operation or abort it. (See Section 12.2.2 for additional information.)

13.2.4 Device-Dependent Cancel-I/O Routine

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$DEVICERR and ERL$DEVICTMO, respectively. Drivers call either routine at fork IPL; each expects to find the address of the device's UCB in R5.

Note: See the *VAX/VMS System Manager's Reference Manual* and the *VAX/VMS Error Log Utility Reference Manual* for help with producing and reading error log files.

Both ERL$DEVICERR and ERL$DEVICTMO perform the following steps:

1. Allocate an error log buffer of the length specified in the device's DDT (in argument **erlgbf** to the DDTAB macro). The following chain of pointers is used to locate the buffer length:

   UCB → DDT → length of error log buffer
2. Load fields from the UCB, the IRP, and the DDB into the buffer.
3. Load into R0 the address of the location in the buffer in which device-register's contents are to be stored.

4 Call a register-dumping routine in the device driver. The following chain of pointers is used to locate the register-dumping routine:

UCB → DDT → register-dumping routine

Specify the address of a register-dumping routine with the value of the **regdmp** argument to the DDTAB macro.

The register-dumping routine expects the following registers to be loaded as described:

Register	Content
R0	Buffer address
R4	Address of CSR if the driver used the WFIKPCH macro to wait for an interrupt or timeout
R5	Address of the device's UCB

The register-dumping routine should save and restore R3 through R11 if the routine requires their use.

The driver register-dumping routine should fill the buffer as follows:

1 Write a longword value representing the number of device registers to be written into the buffer

2 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 of the UCB.
- 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$V_ELG in UCB$L_DEVCHAR.

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 Preparing a Driver for Loading into the Operating System

To prepare a device driver for loading, take the following steps:

1 Write the device driver in one or more source files. If the driver comprises several 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.

2 Assemble the source file(s) with the system's macro library (SYS$LIBRARY:LIB.MLB). For example:

```
$ MACRO MYDRIVER.MAR+SYS$LIBRARY:LIB.MLB/LIBRARY
```

3 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 several 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 a Driver

Once the driver has been 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 CMKRNL privilege, using the following command:

```
$ RUN SYS$SYSTEM:SYSGEN
```

SYSGEN responds with a prompt and waits for further input:

```
SYSGEN>
```

The *VAX/VMS System Generation Utility Reference Manual* describes the full set of SYSGEN commands. The sections that follow describe those commands SYSGEN uses to load drivers:

SYSGEN command	Privilege required
LOAD	CMKRNL
CONNECT	CMKRNL
RELOAD	CMKRNL
SHOW/ADAPTER	CMEXEC
SHOW/CONFIGURATION	CMEXEC
SHOW/DEVICE	CMEXEC

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.

Note: If the controller has only a single unit attached to it, you can issue the CONNECT command to perform the driver-loading tasks normally performed by the LOAD command, as well as its task of creating the device's I/O database (see Section 14.2.2).

Format

```
LOAD file-spec
```

Parameter

file-spec

Name of a file containing an executable driver image. The driver-loading procedure compares the name field in the driver-prologue table (DPT$T_NAME) of the driver being loaded with the names of the drivers in the current system configuration. If the procedure discovers that a driver with the same name already exists in the configuration, it will not load the new driver. If it does not find a configured driver with the same name, it loads the new driver into contiguous locations in nonpaged pool, and links the DPT into the system's linked list of DPTs (headed by IOC$GL_DEVLIST).

The LOAD command uses SYS$SYSTEM as the default device/directory name, and EXE as the default file type.

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 data structures in the I/O database for a specified device. The device-connecting procedure performs the following general functions:

- If the CONNECT command specifies a new device unit on an *existing* controller, it creates a unit-control block for the new unit and calls the driver's unit-initialization routine.
- If the CONNECT command specifies a device unit on a *new* controller, it creates a device-data block, channel-request block, interrupt-dispatch block, and unit-control block and then calls both the controller-initialization and unit-initialization routine in the driver.

The CONNECT command can also load into system memory a driver that has not been previously loaded. (See the discussion of the /DRIVERNAME qualifier below and the description of the LOAD command in Section 14.2.1 for information on driver loading.)

CAUTION: **The database-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 database is apt to become corrupted and will likely cause a system failure.**

Format

```
CONNECT device
```

Parameter

device

Name of the device to be connected. Specify the device name in the format *ddcu* where:

dd = device code (up to 9 alphabetic characters)

c = controller designation (alphabetic)

u = unit number

For example, LPA0 specifies the line printer (LP) on controller A at unit number 0. 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 data structures for the specified device/controller already exist, the device-connecting procedure does not create any data structures or perform any initialization operations. If the device/controller name does not accurately name a device, the procedure creates spurious data structures.

The device-connecting procedure examines the I/O database for data structures that support the specified device. The procedure creates the following data structures if they do not exist:

- Device-data block (DDB) for the specified device/controller combination (*ddcu*).
- Channel-request block (CRB) and interrupt-dispatch block (IDB) for the specified controller. The device-connecting procedure creates these data structures whenever it creates a DDB for a UNIBUS or Q22 bus device.
- Unit-control block (UCB) for the device unit. The device-connecting procedure creates a UCB whenever it creates a DDB, or when a UCB for the specified device does not exist. If a UCB already exists, the procedure ceases its modifications to the I/O database and continues its other tasks.

After it creates the data structures listed above, the procedure initializes them as follows:

- Performs the initialization operations specified by the DPT_STORE macros in the initialization and reinitialization portions of the driver-prologue table (DPT).
- Relocates all addresses in the driver-dispatch table (DDT) and function-decision table (FDT) to absolute system virtual addresses.
- Raises IPL to IPL$_POWER so that initialization is not interrupted.
- If it created a new CRB, calls the controller-initialization routine (if one exists) specified by CRB$L_INTD+VEC$L_INITIAL.
- Calls the unit-initialization routine (if one exists) specified by DDT$L_UNITINIT. If the DDT does not contain the address of a unit-initialization routine, the procedure calls the unit-initialization routine (if any) specified by CRB$L_INTD+VEC$L_UNITINIT.

Required Qualifiers

/[NO]ADAPTER=nexus

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 as listed by the /ADAPTER qualifier to the SYSGEN command SHOW. (See Section 14.2.4 for a discussion of the SHOW/ADAPTER command.)

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).

The nexuses of the various VAX processors are conventionally assigned as listed in Table 14–1.

Table 14–1 Conventional Nexus Assignments

Adapter	VAX–11/725 VAX–11/730	VAX–11/750	VAX–11/780 VAX–11/782 VAX–11/785 VAX 8600 VAX 8650	MicroVAX I MicroVAX II VAX 8200[1] VAX 8800[1]
UNIBUS				
0	3	8	3	0
1	–	9	4	–
2	–	–	5	–
3	–	–	6	–
MASSBUS				
0	–	4	8	–
1	–	5	9	–
2	–	6	10	–
3	–	–	11	–

[1]The BI-to-UNIBUS adapter on a VAX 8800 or VAX 8200 must be located at Node 0.

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-addr
UNIBUS or Q22 bus address of the device's control and status register (CSR). All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X).

/CSR_OFFSET=value
Offset from the CSR address of a multiple-device board to the CSR address of the device. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X).

/VECTOR=vector-addr
Q22 bus or 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.

/VECTOR_OFFSET=value
Offset from the interrupt vector of a multiple-device board to the interrupt vector of the device being connected. 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=vector-cnt

Number of interrupt vectors for the device. If this qualifier is omitted, the default number of vectors is 1. The number specified by the /VECTOR qualifier is the address of the lowest vector. Vectors must be contiguous.

/DRIVERNAME=driver

Name of the driver for the device to be connected. If the driver for the specified device has not yet been loaded, the CONNECT command will load its driver. First, it will attempt to load the driver whose name is specified in this qualifier, defaulting to a file type of EXE in device/directory SYS$SYSTEM).

If the /DRIVERNAME 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 field DDB$T_DRVNAME.

Consult the SYSGEN device table in Section 14.3.2 for the driver names of the devices supported by VAX/VMS.

/ADPUNIT=unit-number

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's number.

/MAXUNITS=max-unit-cnt

Maximum number of units attached to the controller. This number determines the size of the UCB list appended to the IDB. If specified, this value overrides the maximum number of units designated in the DPT. The maximum number of units is stored in the field IDB$W_UNITS.

Example

```
SYSGEN> CONNECT LPA0 /ADAPTER=UBO/CSR=%O777514/VECTOR=%O200
```

This command loads the driver LPDRIVER, if it is not already loaded, and creates the data structures (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.

CAUTION: **Use the RELOAD command only when all devices supported by the driver are inactive. The checks for activity made by the RELOAD command might not detect all device activity, and changing a driver while an I/O request is being processed will cause a system failure.**

Format

```
RELOAD file-spec
```

Parameter

file-spec

Name of a file containing an executable driver image. The driver-reloading procedure compares the name field in the driver-prologue table (DPT$T_NAME) of the driver being loaded with the names of the drivers in the current system configuration. If no such driver is configured, the driver-reloading procedure loads the driver as described in the discussion of the LOAD command in Section 14.2.1.

If the driver-reloading procedure finds a driver with the specified name in the configuration, it first determines that the current driver can be replaced in the following steps:

- Confirms that the DPT$M_NOUNLOAD flag of the current driver is not set.
- Calls the current driver's driver-unloading routine, if one exists, and confirms that the returned status is a success code.
- Ensures that no devices that use the current driver are busy, as indicated by the UCB$V_BSY bit set in UCB$L_STS.

If these checks succeed, the driver-reloading procedure replaces the current driver with the new driver in the following manner:

1. Loads the new driver into contiguous locations in nonpaged system memory.
2. Searches the I/O database for references to the driver. If any device-data block refers to the driver being reloaded, the driver-reloading procedure must reinitialize data structure fields according to the reinitialization instructions in the new driver-prologue table (see Section 7.1).

   Fields that must be reinitialized when a driver is reloaded include those that contain relative addresses within the driver:

   - Addresses of the driver's interrupt-servicing routines
   - Addresses of the device's unit-initialization and controller-initialization routines
   - Address of the driver-dispatch table
3. Calls the driver's controller-initialization routine. (It does not call the unit-initialization routine.)
4. Removes the newly replaced driver from the system's linked list of DPTs (headed by IOC$GL_DEVLIST). (headed by IOC$GL_DEVLIST) and deallocates the nonpaged system space the old driver occupied.
5. Links the address of the new driver-prologue table to the system's list of DPTs.

14.2.4 SHOW/ADAPTER Command

The SHOW/ADAPTER command displays nexus numbers and generic names of UNIBUS and MASSBUS adapters, memory controllers, and interconnection devices such as the DR32. Use of the SHOW/ADAPTER command requires CMEXEC privilege.

Format

```
SHOW/ADAPTER
```

Example

```
SYSGEN> SHOW/ADAPTER
CPU Type:  11/780
Hardware Revision #96
  Nexus   Generic Name or Description
    1     16K memory, non-interleaved
    4     UBO
    5     UB1
    8     MBO
    9     MB1
```

This example shows a VAX–11/780 that uses one memory controller composed of 16K-bit chips, two UNIBUS adapters, and two MASSBUS adapters.

14.2.5 SHOW/CONFIGURATION Command

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 to or autoconfigured in the system.

Format

```
SHOW/CONFIGURATION
```

Optional Qualifiers

/ADAPTER=nexus

Nexus value of the UNIBUS adapter, MASSBUS adapter, or other interconnect to be displayed. The nexus value can be expressed as an integer or as one of the generic names listed by the SHOW/ADAPTER command.

/COMMAND_FILE

Option by which you instruct SYSGEN to format all device data produced by the SHOW/CONFIGURATION command into CONNECT/ADAPTER=nexus commands and write them to a specified output file. By executing the commands in this file, you can remove a device from floating address space without completely reconnecting the CSR and vector addresses of the remaining devices. See the *VAX/VMS System Generation Utility Reference Manual* for more details.

/OUTPUT=file-spec

Name of a file into which SHOW/CONFIGURATION is to write device configuration information.

Example

```
SYSGEN> SHOW/CONFIGURATION/ADAPTER=UB1

     System CSR and Vectors on  24-JUL-1986 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 Command

The SHOW/DEVICE command displays the following information:

- Name of the driver
- Starting virtual address of the driver (that is, the address of the driver-prologue table)
- Ending virtual address of the driver
- Generic device/controller name associated with the driver
- Addresses of the device-data block, channel-request block, and interrupt-dispatch block for the generic device/controller supported by the driver
- Unit number and UCB address of each device unit associated with the driver

The SHOW/DEVICE command requires CMEXEC privilege.

Format

```
SHOW/DEVICE [=driver-name]
```

Parameter

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.

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 database that describes all supported DIGITAL peripherals in the configuration. The following command requests SYSGEN to prepare a database for all supported DIGITAL devices attached to every UNIBUS, Q22 bus, and MASSBUS:

```
SYSGEN> AUTOCONFIGURE ALL
```

To configure devices attached to the UNIBUS or Q22 bus, SYSGEN goes through the steps described in subsequent sections of this chapter.

DIGITAL-supplied devices are attached to the UNIBUS or Q22 bus 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 (l 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 (l 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_8. The base of floating CSR space is 760010_8.

14.3.1 The SYSGEN Autoconfiguration Facility

The SYSGEN utility automatically configures a UNIBUS or Q22 bus as follows:

- It initializes the base of floating space to 300_8 and 760010_8 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

database for that device and loads the driver's image if necessary. If the device at the CSR is supported by VAX/VMS and SYSGEN cannot locate its associated driver's image, it generates an error message. If the device is unsupported and has no corresponding driver's image, SYSGEN ignores the condition.

14.3.2 SYSGEN Device Table

The SYSGEN device table (see Table 14–2 lists the characteristics of all DIGITAL devices. This table indicates the following information for each device type:

- Device name
- Device controller name
- Interrupt vector
- Number of interrupt vectors per controller
- Vector alignment factor
- Address of the first device register for each controller recognized by SYSGEN (the first register is usually, but not always, the CSR)
- Number of registers per controller
- Device driver name
- Indication of whether the driver is or is not supported

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.

Table 14–2 SYSGEN Device Table

Device Name	Controller Name	Vector	No. of Vectors	Alignment	CSR /Rank	No. of Registers	Driver Name	Support
CR	CR11	230	—	—	777160	—	CRDRIVER	Yes
DM	RK611	210	—	—	777440	—	DMDRIVER	Yes
LP	LP11	200 170 174 270 274	—	—	777514 764004 764014 764024 764034	—	LPDRIVER	Yes
DL	RL11	160	—	—	774400	—	DLDRIVER	Yes
MS	TS11	224	—	—	772520	—	TSDRIVER	Yes
DY	RX211	264	—	—	777170	—	DYDRIVER	Yes

Table 14–2 (Cont.) SYSGEN Device Table

Device Name	Controller Name	Vector	No. of Vectors	Alignment	CSR /Rank	No. of Registers	Driver Name	Support
DY	RB730	250	—	—	775606	—	DQDRIVER	Yes
PU	UDA	154	—	—	772150	—	PUDRIVER	Yes
PT	TU81	260	—	—	774500	—	PUDRIVER	Yes
XE	UNA	120	—	—	774510	—	XEDRIVER	Yes
XQ	QNA	120	—	—	774440	—	XQDRIVER	Yes
OM	DC11	Float	2	8	774000 774010 774020 774030 . . . 32 units maximum	—	OMDRIVER	No
DD	TU58	Float	2	8	776500 776510 776520 776530 . . . 16 units maximum	—	DDDRIVER	Yes
OB	DN11	Float	1	4	775200 775210 775220 775230 . . . 16 units maximum	—	OBDRIVER	No
YM	DM11B	Float	1	4	770500 770510 770520 770530 . . . 16 units maximum	—	YMDRIVER	No

Table 14–2 (Cont.) SYSGEN Device Table

Device Name	Controller Name	Vector	No. of Vectors	Alignment	CSR /Rank	No. of Registers	Driver Name	Support
OA	DR11C	Float	2	8	767600 767570 767520 767550 . . . 16 units maximum	—	OADRIVER	No
PR	PR611	Float	1	8	772600 772604 772610 772614 . . . 8 units maximum	—	PRDRIVER	No
PP	PP611	Float	1	8	772700 772704 772710 772714 . . . 8 units maximum	—	PPDRIVER	No
OC	DT11	Float	2	8	777420 777422 777424 777426 . . . 8 units maximum	—	OCDRIVER	No
OD	DX11	Float	2	8	776200 776240	—	ODDRIVER	No
YL	DL11C	Float	2	8	775610 775620 775630 775640 . . . 31 units maximum	—	YLDRIVER	No
YJ	DJ11	Float	2	8	Float	4	YJDRIVER	No
YH	DH11	Float	2	8	Float	8	YHDRIVER	No

Table 14–2 (Cont.) SYSGEN Device Table

Device Name	Controller Name	Vector	No. of Vectors	Alignment	CSR /Rank	No. of Registers	Driver Name	Support
OE	GT40	Float	4	8	772000 772010	—	OEDRIVER	No
LS	LPS11	Float	6	8	770400	—	LSDRIVER	No
OR	DQ11	Float	2	8	Float	4	ORDRIVER	No
OF	KW11W	Float	2	8	772400	—	OFDRIVER	No
XU	DU11	Float	2	8	Float	4	XUDRIVER	No
XW	DUP11	Float	2	8	Float	4	OODRIVER	No
XV	DV11	Float	3	8	775000 775040 775100 775140	—	XVDRIVER	No
OG	LK11	Float	2	8	Float	4	OGDRIVER	No
XM	DMC11	Float	2	8	Float	4	XMDRIVER	Yes
TT	DZ11	Float	2	8	Float	4	DZDRIVER	Yes
XK	KMC11	Float	2	8	Float	4	XKDRIVER	No
OH	LPS11	Float	2	8	Float	4	OHDRIVER	No
OI	VMV21	Float	2	8	Float	4	OIDRIVER	No
OJ	VMV31	Float	2	8	Float	8	OJDRIVER	No
OK	DWR70	Float	2	8	Float	4	OKDRIVER	No
DL	RL11	Float	1	4	Float	4	DLDRIVER	Yes
MS	TS11	Float	1	4	772524 772530 772534	—	TSDRIVER	Yes
LA	LPA11	Float	2	8	770460	—	LADRIVER	Yes
LA	LPA11	Float	2	8	Float	8	LADRIVER	Yes
OL	KW11C	Float	2	8	Float	4	OLDRIVER	No
RSV	RSV	Float	1	8	Float	4	RSVDRIVER	No
DY	RX211	Float	1	4	Float	4	DYDRIVER	Yes
XA	DR11W	Float	1	4	Float	4	XADRIVER	Yes
XB	DR11B	124	—	—	772410	—	XBDRIVER	No
XB	DR11B	Float	1	4	772430	4	XBDRIVER	No
XB	DR11B	Float	1	4	Float	4	XBDRIVER	No
XD	DMP11	Float	2	8	Float	4	XDDRIVER	Yes
ON	DPV11	Float	2	8	Float	4	ONDRIVER	No
IS	ISB11	Float	2	8	Float	4	ISDRIVER	No
XD	DMV11	Float	2	8	Float	8	XDDRIVER	No
XE	UNA	Float	1	4	Float	4	XEDRIVER	No
PU	UDA	Float	1	4	Float	2	PUDRIVER	Yes
TX	DMF32	Float	8	4	Float	16	YCDRIVER	Yes
XG	—	—	—	—	—	—	XGDRIVER	Yes

Table 14–2 (Cont.) SYSGEN Device Table

Device Name	Controller Name	Vector	No. of Vectors	Alignment	CSR /Rank	No. of Registers	Driver Name	Support
LC	—	—	—	—	—	—	LCDRIVER	Yes
XI	—	—	—	—	—	—	XIDRIVER	No
XS	KMS11	Float	3	8	Float	8	XSDRIVER	No
XP	PCL11	Float	2	8	764200 764240 764300 764340	—	XPDRIVER	No
VB	VS100	Float	1	4	Float	8	VBDRIVER	No
PT	TU81	Float	1	4	Float	2	PUDRIVER	Yes
OQ	KMV11	Float	2	8	Float	8	OQDRIVER	No
UK	KCT32	Float	2	8	764400 764440 764500 764540	—	UKDRIVER	No
IX	IEQ11	Float	2	8	764100	—	IXDRIVER	No
TX	DHV11	Float	2	8	Float	8	YFDRIVER	Yes
TX	DMZ32 CPI32	Float	6	4	Float	16	YCDRIVER	Yes
XG	CPI32	Float	6	4	Float	16	XGDRIVER	Yes
DT	TC11	214	—	—	777340	—	DTDRIVER	No
VC	VC01B	060	—	—	777200	—	VCDRIVER	Yes

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 1.

The **deliver** argument to the DPTAB macro specifies the address of a driver-specific unit-delivery 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 unit-delivery routine once for each unit, the number of which being specified in the **defunits** argument.

The unit-delivery routine prevents the creation of unit-control blocks for devices that do not respond to a test for their presence.

If the unit-delivery 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 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 CSR 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 adapter-control block.

The configuration-control block is described in Figure A-1 and Table A-1.

A driver may or may not specify a unit-delivery routine. For instance, the DZ11's device driver specifies 8 as the default unit number, but no routine to configure eight terminal units automatically for each DZ11's CSR. The RK611 device driver specifies 8 as the default number of units and also specifies the address of a unit-delivery routine that is called once for each of the eight possible devices on the controller.

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 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.

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 because subsequent VAX/VMS maintenance updates might add new devices to the SYSGEN device table.[1]

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

[1] UNIBUS addresses 764100 through 767776 are available for non-DIGITAL-supplied devices.

Controller	Vector(s)	CSR (first register)
3 DR11Bs*	124	772410 (CSR is third register)
	400	772430
	410	760300
1 customer device	420 (or higher)	760320 (or higher)

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. However, 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. Table 15–1 describes the possible values of these flags. Following a boot that includes XDELTA, executing a BPT instruction causes control to transfer to a fault handler located in XDELTA.

Table 15–1 Boot Flags That Control the Loading of XDELTA

Flag Value (f)	Meaning
0	Normal nonstop bootstrap (default)
1	Stop in SYSBOOT (equivalent to @D*xy*GEN on the VAX–11/780)
2	Include XDELTA with the system but do not take the initial breakpoint
6	Include XDELTA with the system and take the initial breakpoint
7	Include XDELTA with the system, stop in SYSBOOT and take the initial breakpoint at system initialization (equivalent to @D*xy*XDT on the VAX–11/780)

The procedures for bootstrapping the system with XDELTA differ depending upon which processor the operating system is running. Some processors that use a console block storage device supply a special boot command file that automatically includes XDELTA in the system and causes the processor to stop in SYSBOOT and take the initial breakpoint at system initialization. When booting other processors, you must specify the appropriate flag value in the BOOT command. Table 15–2 lists some recommended methods for booting with XDELTA. See the *Guide to VAX/VMS Software Installation* for additional information.

Table 15–2 Recommended Methods for Bootstrapping with XDELTA

Boot Commands	Explanation
MicroVAX II, MicroVAX I, and VAX–11/750[1] Processors	

B[/*f*] *devname*	B is the console BOOT command. The flags (*f*) parameter is a 32-bit hexadecimal integer loaded into R5 as input to VMB.EXE, the primary bootstrap program. See Table 15–1 for a list of its possible values.
	
	Using the format *ddcu*, specify the name of the device that contains the volume to be bootstrapped. You must supply both controller (*c*) and unit (*u*) identifiers; there are no defaults. If you omit *devname*, the *f* parameter is ignored.
	
	The following example bootstraps a MicroVAX II system from DUA0.[2]
	
	`>>>B/7 DUA0`
	`SYSBOOT>`
	`SYSBOOT>CONTINUE`

VAX 8800 and VAX 8200 Processors	

B[/R5:*f*] *devname*	B is the console BOOT command. The flags (*f*) parameter is a 32-bit hexadecimal integer loaded into R5 as input to VMB.EXE, the primary bootstrap program. See Table 15–1 for a list of its possible values.
	
	For the VAX 8800, specify *devname* in the format *dddduuu*. The console places the specified unit number (*uuu*) in R3 and executes the procedure *ddd*BOO.COM. If you do not specify *devname*, the console executes DEFBOO.COM. To use the /R5 qualifier, you must have previously removed or commented out the DEPOSIT R5 command in the procedure to be executed.
	
	For the VAX 8200, specify *devname* in the format *ddxu*, where *x* represents the number of the VAXBI node to which the boot device unit is attached. If you do not specify *devname*, the console boots from the default boot device.
	
	The following example bootstraps a VAX 8200 system from the boot disk at VAXBI node 4.[2]
	
	`>>>B/R5:7 DU40`
	`SYSBOOT>`
	`SYSBOOT>CONTINUE`

VAX–11/780, VAX–11/782, and VAX–11/785 Processors	

@DMAXDT
@DBAXDT
	Use either DMAXDT.CMD or DBAXDT.CMD, depending upon the boot device. The following example boots from DMA0, first depositing the value 0 in R3.[2]
	
	`>>>DEPOSIT R3 0`
	`>>>@DMAXDT`
	`SYSBOOT>`
	`SYSBOOT>CONTINUE`

[1]The console TU58 of the VAX–11/750 processor contains command files (DMAXDT.CMD and DBAXDT.CMD) analogous to those supplied for the VAX–11/780. See the *Guide to VAX/VMS Software Installation* for additional information.

[2]At the SYSBOOT prompt enter other required SYSBOOT commands and conclude the boot operation with a CONTINUE command. If you do not set or load system parameters with a USE command, the system uses parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, set system parameter *BUGREBOOT* to 0.

Table 15–2 (Cont.) Recommended Methods for Bootstrapping with XDELTA

Boot Commands	Explanation
VAX–11/730 and VAX–11/725 Processors	

@DQAXDT
@DQ0XDT
	Use either DQAXDT.CMD or DQ0XDT.CMD, depending upon the boot device. The following example boots from DQA1, first depositing the value 1 in R3. When the boot device is DQA0, you can omit this step and execute DQ0XDT.COM.[2]
	`>>>D/G/L 3 1`
	`>>>@DQAXDT`
	`SYSBOOT>`
	`SYSBOOT>CONTINUE`

VAX 8600 and VAX 8650 Processors	

@DU0XDT	Use DU0XDT.COM, if available on the console media, according to the method described for the VAX–11/780. Otherwise, perform a normal bootstrap using the available *ddu*GEN.COM or *ddu*BOO.COM according to the following method:
	Use the */NOSTART* qualifier in the BOOT command to cause the processor to pause and await console commands after it boots. After a variety of progress messages are displayed, the console prompt reappears. First, a value for the flag that controls XDELTA loading (see Table 15–1). Then, examine the current value of R5; if it is nonzero (for instance, it is the system root number), perform an inclusive-OR operation upon it and your selected XDELTA flag value.[2]
	`>>>BOOT/NOSTART`
	`SYSBOOT>EXAMINE R5`
	`SYSBOOT>DEPOSIT R5 7`
	`SYSBOOT>`
	`SYSBOOT>CONTINUE`

[2]At the SYSBOOT prompt enter other required SYSBOOT commands and conclude the boot operation with a CONTINUE command. If you do not set or load system parameters with a USE command, the system uses parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, set system parameter *BUGREBOOT* to 0.

15.1.1 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 and in the *VAX/VMS Delta/XDelta Utility Reference Manual.*

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 0, 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 15–1 provides a sample dialogue.

The first SHOW command in Figure 15–1 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.

Figure 15–1 Loading a Driver

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN>LOAD DMAO:[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
```

15.3 Inserting Breakpoints in Driver 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 one or more calls to the special system routine INI$BRK within the source code of either the controller- or unit-initialization routine. To call INI$BRK, use 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 top of the stack. Another display command (/) reveals the contents of the top of the stack, which should be 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:

1. Run SYSGEN and issue the SHOW/DEVICE command. The resulting display lists the location in nonpaged pool at which SYSGEN loaded the driver.
2. 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 Table 15–3 issue a software interrupt to a single processor at IPL 5 or a multiprocessor at IPL 15. The corresponding interrupt-servicing routine handles the interrupt by calling the routine INI$BRK, which in turn executes the first XDELTA breakpoint. XDELTA then issues this message:

```
1 BRK AT nnnnnnnn
address/NOP
```

Table 15–3 Requesting an XDELTA Software Interrupt

Processor	Boot Commands

MicroVAX II
MicroVAX I[1]
	Press and release the HALT button on the CPU control panel, or press the BREAK key (if enabled) on the console terminal. Then issue these commands:
	`>>>D/I 14 5`
	`>>>C`

VAX 8800[1,2]	`$` CTRL/P
	`>>>HALT`
	`>>>D/I 14 F`
	`>>>C`

VAX 8600
VAX 8650[1]
	`$` CTRL/P
	`>>>HALT`
	`>>>D/I 14 5`
	`>>>C`

VAX 8200
VAX–11/750
VAX–11/730
VAX–11/725[1]
	`$` CTRL/P
	`>>>D/I 14 5`
	`>>>C`

VAX–11/780
VAX–11/785
	`$` CTRL/P
	`>>>HALT`
	`>>>DEPOSIT/I 14 5`
	`>>>CONTINUE`

VAX–11/782[2]	`$` CTRL/P
	`>>>HALT`
	`>>>DEPOSIT/I 14 F`
	`>>>CONTINUE`

[1]These VAX processors accept only one-character console commands.

[2]Deposit *F* in the processor IPL register only if multiprocessing is in effect (for example, if a START/CPU command has been executed); otherwise deposit the value *5* .

15.6 Examining the Vector-Jump Table

To gain familiarity with the I/O database, you might wish to look for the address of the location in the channel-request block that contains a JSB instruction to the driver's interrupt-servicing 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-servicing routine on nondirect and direct vector adapters follow.

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 address of the driver's interrupt-servicing routine at the expected vector does not guarantee that an interrupt from the device will dispatch to the driver's interrupt-servicing routine. If the device's physical vector is set to some other address, an interrupt from the device can dispatch to some other interrupt-servicing routine, or dispatch to an unassigned vector.

See the SYSGEN device table shown in Section 14.3.2 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 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, use 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 or the *VAX/VMS Delta/XDelta Utility Reference Manual* for a detailed discussion of XDELTA commands.

15.8 Destroying Register Contents

Because 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 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$IOFORK saves the calling code's R4 in a fork block, and then writes the device's IPL into R4. Because the normal flow of events is that an interrupt-servicing routine restores a driver with a JSB instruction and the driver then calls EXE$IOFORK which returns to the interrupt-servicing routine, the instructions following the JSB in the interrupt-servicing 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-servicing 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 the UCB, IRP, or PSL

In addition to using XDELTA to debug drivers, you can also 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/
```

The PSL location is stored in the longword following the PC.

15.10 XDELTA Commands

Table 15–4 summarizes XDELTA commands. The sections that follow this table describe the commands.

Table 15–4 XDELTA Command Summary

Command	Function
Set Display Mode	
[B	Set byte mode
[W	Set word mode
[L	Set longword mode
[I	Set instruction mode
"	Set ASCII mode
Set and Proceed from Breakpoint	
;P	Proceed from breakpoint
;B	Set/clear/display breakpoint
Open, Examine, and Close Location	
/	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
Deposit in Location	
'string'	Deposit string at current location, autoincrementing the current location symbol (.). Every RET and LF typed will be stored. A single quote terminates the string.

Table 15–4 (Cont.) XDELTA Command Summary

Command	Function
Step, Set Location, and Execute Code	
S	Execute one instruction, step into subroutine call
O	Execute one instruction, step over subroutine call (on CALL*x*, JSB, or BSB*x* instruction)
;G	Go to location and proceed
;E	Execute command string at location
Special Symbols	
,	Field separator
Q	Last quantity displayed
=	Display value of expression; set Q
Xn	Base register n
;X	Set base register
Rn	Register n
Pn	Processor register n
G	Add ^X80000000 to subsequent or preceding value
H	Add ^X7FFE0000 to subsequent or preceding value
.	Current location
Operators	
+	Add
–	Subtract
space	Add
*	Multiply
@	Shift
%	Divide

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 - .
```

XDELTA evaluates expressions from left to right with no precedence, and ignores trailing operators. To display the value of an expression, use the XDELTA Show Value (=) command, as follows:

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.

15.10.2 Special Symbols

XDELTA defines the following special symbols:

.	Current location; set by slash (/), exclamation point (!) and TAB operations.
Q	Last quantity displayed; you can also change this value by using the Show Value (=) command described in Section 15.10.1.
X0–XF	Base registers; used for remembering values. Set base registers by means of the Set Base Register command (;X) described in Sections 15.7 and 15.10.2.3. XDELTA, by default, stores special values in base registers X4 and X5 that help reference the Process Control Block of the current process (see Section 15.10.2.1). Also, XDELTA initializes XE and XF with special commands that help reference page-frame numbers as described in Section 15.10.2.2.
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.

15.10.2.1 Stored Base Registers

XDELTA defines two base registers useful in system debugging: X4 and X5. Base register X4 corresponds to the global symbol SCH$GL_CURPCB. This symbol contains the address of the current process' software process-control block (PCB). Base register X5 corresponds to the global symbol SCH$GL_PCBVEC, which contains the starting address of the list of PCB slots.

15.10.2.2 Stored Command Strings

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 database 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
```

After executing these commands, you can use the commands stored in XE and XF 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

15.10.2.3 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 (X2) 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_{16} from the base register. If the address falls outside this range, XDELTA displays it as a hexadecimal value.

15.10.3 Set Display Mode

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 an instruction display, see Section 15.7.

You can display contents of memory locations in ASCII characters by typing an address expression followed by a 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. The width remains unchanged.

15.10.4 Open, Examine, and Close Location

XDELTA provides the commands described in the following sections to open, examine, and close the specified memory locations.

15.10.4.1 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, as shown below:

```
address-expression/old-value /old-value /old-value
```

15.10.4.2 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/VMS 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
```

15.10.4.3 Close and Display Next Location Command

Syntax

```
LF
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).

15.10.4.4 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 instructions. Otherwise, XDELTA displays the contents of each location in the current data type (byte, word, or longword).

15.10.4.5 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.

15.10.4.6 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.

15.10.5 Breakpoints

XDELTA uses the following commands to set and clear breakpoints, display a list of set breakpoints, continue from a breakpoint, and set a complex breakpoint.

15.10.5.1 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 can now enter XDELTA commands. You can reset the flag that controls instruction display mode by issuing the open and display value command.

15.10.5.2 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.

15.10.5.3 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 15.10.5.5)
- Command string address (for complex breakpoints)

15.10.5.4 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.

15.10.5.5 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.

15.10.6 Step, Set Location, and Execute Instruction Commands

The following XDELTA commands enable you to step through and execute driver code.

15.10.6.1 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.

15.10.6.2 Execute Instruction and Step 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.

15.10.6.3 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; this command steps over subroutines.

15.10.7 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.

15.11 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 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]
```

Modification 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 processors that use direct vector interrupts, these references cause a fatal machine check exception. On VAX 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 0. 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 Reference Material

A The I/O Database

The I/O database is a collection of data structures that provide the following types of information to the VAX/VMS operating system and drivers to help monitor the status of, and control the functions of, the I/O subsystem:

- Descriptions of each pending and in-progress I/O request
- Characteristics of each device type
- Number and type of each device unit
- Status of current activity on each device unit
- External entry points to all device drivers
- Entry points for controller and device unit initialization routines
- Code that dispatches interrupts to the appropriate servicing routines
- Addresses of device registers
- Bit maps describing the allocation of data paths and mapping registers

Much of the I/O database is created and used only by VAX/VMS routines; other parts are the primary source of data for the device drivers. All of its data structures—with the exception of the channel-control block (CCB)—exist in nonpaged system memory. This appendix identifies all data structures in the I/O database, and describes their fields in the order in which they appear in the structures.

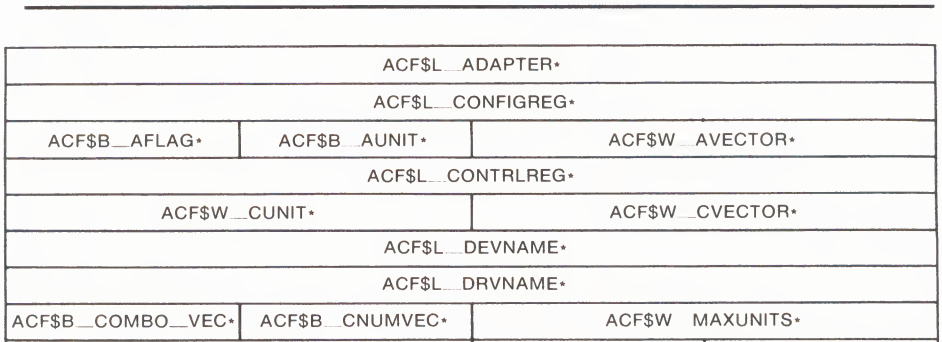
Note: Driver code must consider fields marked by asterisks to be read-only fields. Fields marked by "spare" or "unused" are reserved for future use by DIGITAL unless otherwise specified.

A.1 Configuration-Control Block (ACF)

The configuration-control block (ACF) is used by the SYSGEN autoconfiguration facility to describe the device it is adding to the system. Device drivers can gain access to this data structure only if they have specified a unit-delivery routine in the DPT and only when that routine is executing. Under certain conditions, the information stored in the ACF might 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 (ACF)



ZK-1778-84

Table A–1 Contents of the Configuration-Control Block

Field Name	Contents
ACF$L_ADAPTER*	Address of ADP for adapter currently being configured.
ACF$L_CONFIGREG*	Address of configuration register for adapter currently being configured.
ACF$W_AVECTOR*	Offset from base of SCB to interrupt vector of adapter currently being configured.
ACF$B_AUNIT*	Adapter unit number of device or controller currently being configured.
ACF$B_AFLAG*	Flags associated with autoconfiguration operation. Flags defined in this field include the following:
	ACF$V_RELOAD — Reloading driver code.
	ACF$V_CRBBLT — CRB and IDB already built for device.
	ACF$V_SCBVEC — CVECTOR is offset into SCB.
	ACF$V_NOLOAD_DB — Do not load I/O database, only load driver.
	ACF$V_SUPPORT — VAX/VMS supported device.
	ACF$V_GETDONE — Addresses of data structures in I/O database have been obtained.
ACF$L_CONTRLREG*	Address of CSR for controller currently being configured.
ACF$W_CVECTOR*	Offset into ADP's vector table to longword that contains transfer address of interrupt vector used by controller currently being configured (if ACF$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 device currently being configured.
ACF$L_DEVNAME*	Address of counted ASCII string that gives name of controller currently being configured.
ACF$L_DRVNAME*	Address of counted ASCII string that gives driver name for controller currently being configured.
ACF$W_MAXUNITS*	Maximum number of units that can be connected to controller currently being configured.

Table A–1 (Cont.) Contents of the Configuration-Control Block

Field Name	Contents
ACF$B_CNUMVEC*	Number of interrupt vectors to configure for controller currently being configured.
ACF$B_COMBO_VEC*	Offset to vectors for combo device. (The name of this field is ACF$B_COMBO_VECTOR_OFFSET.)
ACF$B_COMBO_CSR*	Offset to start of control registers of combo device. (The name of this field is ACF$B_COMBO_CSR_OFFSET.)
ACF$B_NUMUNIT*	Number of units to be configured for controller currently being configured.
ACF$L_DLVR_SCRH	Field available for use by unit-delivery routine. SYSGEN never alters this field.

A.2 Adapter-Control Block (ADP)

Each MASSBUS and UNIBUS adapter, as well as each Q22 bus, configured in the system is represented to VAX/VMS and driver routines by an adapter-control block (ADP). The ADP stores adapter-specific static and dynamic data such as the adapter CSR address and mapping-register-wait queues.

The adapter-control block for a UNIBUS adapter and MicroVAX II Q22 bus is illustrated in Figure A–2 and described in Table A–2.

Table A–2 Contents of Adapter-Control Block

Field Name	Contents
ADP$L_CSR*	Virtual address of adapter configuration register. The VAX/VMS initialization routine writes this field. The configuration register marks the base of adapter register space, an area that contains data path registers, mapping registers, or any other registers appropriate to the implementation of the adapter.
ADP$L_LINK*	Address of next ADP. The VAX/VMS adapter initialization routine writes this field. A value of 0 indicates that this is the last ADP.
ADP$W_SIZE*	Size of ADP. The VAX/VMS adapter initialization routine writes this field when the routine creates the ADP. For the UNIBUS and the Q22 bus, this includes the UNIBUS interrupt servicing code and device vector table.
ADP$B_TYPE*	Type of data structure. The VAX/VMS adapter initialization routine writes the symbolic constant DYN$C_ADP into this field when the routine creates the ADP.
ADP$B_NUMBER*	Number of this type of adapter (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$W_TR*	Nexus number of adapter. The VAX/VMS adapter initialization routine writes this field when the routine creates the ADP. The driver-loading procedure compares the nexus number specified in a CONNECT command with this field of each ADP in the system to determine to which adapter a device is attached.
ADP$W_ADPTYPE*	Type of adapter. The CPU initialization routine writes the symbolic constant AT$_UBA into this field when the routine creates an ADP for a UNIBUS adapter or Q22 bus. AT$_MBA is the type code for a MASSBUS adapter.

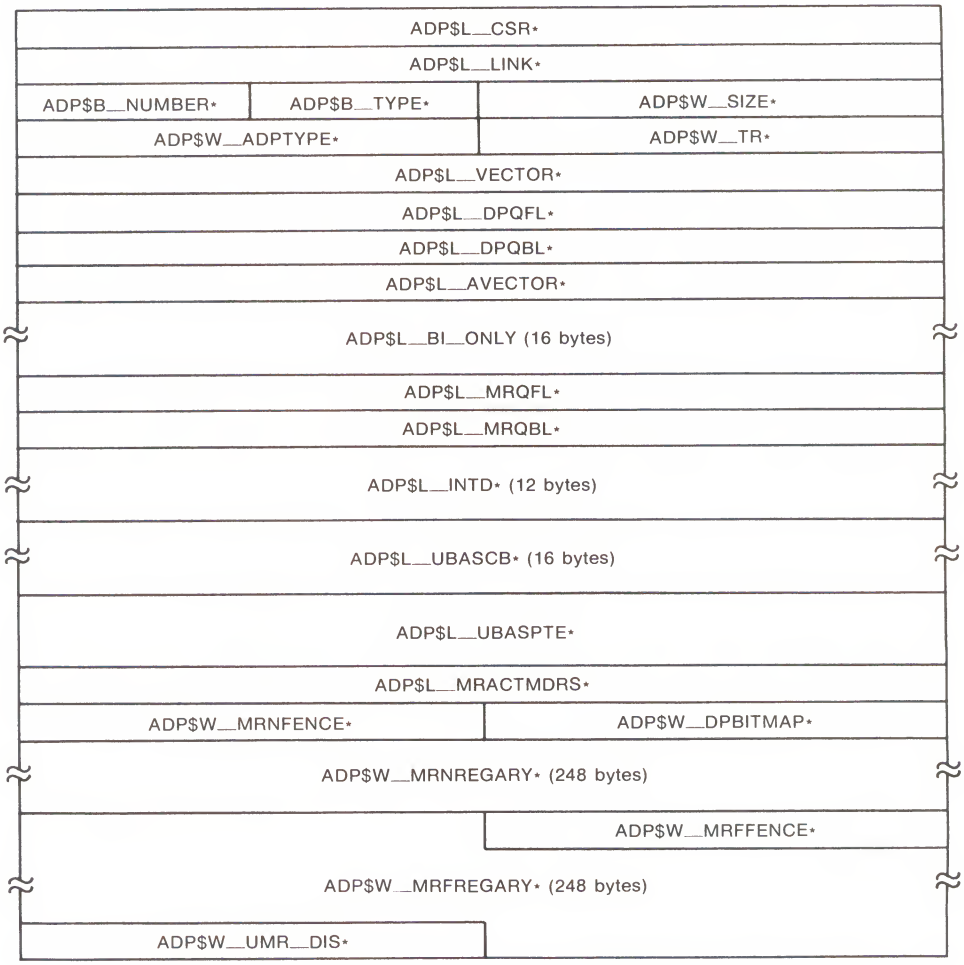
Table A–2 (Cont.) Contents of Adapter-Control Block

Field Name	Contents
ADP$L_VECTOR*	Address of vector table. The table is 512 bytes of longword vectors that correspond to device interrupt vectors (0–%O777). On VAX processors that handle direct vector interrupts, ADP$L_VECTOR points to the second (or third) page of the SCB. The CPU uses this page when it dispatches the device interrupt to the driver interrupt-servicing routine. Each vector entry that corresponds to a vector in use contains the address of the controller's interrupt dispatcher (CRB$L_INTD). On VAX processors that handle nondirect vector interrupts, ADP$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$L_INTD+2). When the UNIBUS adapter interrupts on behalf of a UNIBUS device, the UNIBUS adapter interrupt-servicing routine saves R0 through R5, determines the vector address of the interrupting device, indexes into the vector table, and executes the instruction at CRB$L_INTD+2. For both types of VAX processor, vector table entries that correspond to unused vectors contain the address of the adapter's unexpected-interrupt-servicing routine.
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$L_AVECTOR*	Address of first SCB vector for adapter.
ADP$L_BI_ONLY*	Reserved to DIGITAL.
ADP$L_MRQFL*	Mapping-register-wait queue's forward link. IOC$REQMAPREG and IOC$RELMAPREG read and write these fields. When a driver fork process requests a set of mapping registers and the set is not currently available, IOC$REQMAPREG saves driver fork context in the device's UCB fork block, inserts the fork block address in the mapping-register-wait queue, and suspends the driver fork process. When another driver calls IOC$RELMAPREG to release a set of mapping registers, the routine dequeues a UCB fork block address from the mapping-register-wait queue, allocates the requested set of mapping registers to the driver, and reactivates that driver fork process.
ADP$L_MRQBL*	Mapping-register-wait queue's backward link. IOC$REQMAPREG and IOC$RELMAPREG read and write this field.
ADP$L_INTD*	Interrupt transfer vector. When a device attached to the UNIBUS or Q22 bus requests a hardware interrupt, the processor transfers control to the UNIBUS or Q22-bus ADP$L_INTD field. The field contains code that dispatches the interrupt to the proper driver interrupt-servicing routine. The interrupt transfer vector is only used for UNIBUS adapters that directly generate interrupts and the MicroVAX I and MicroVAX II Q22 bus.

Table A–2 (Cont.) Contents of Adapter-Control Block

Field Name	Contents
ADP$L_UBASCB*	Series of four longwords that contain SCB entry values, one for each bus request (BR) level or interrupt vector. The UNIBUS adapter power failure recovery procedure uses these values.
ADP$L_UBASPTE*	Page-table-entry (PTE) values for base of UNIBUS adapter register space and base of UNIBUS I/O register space. These values are used during UNIBUS adapter power failure recovery.
ADP$L_MRACTMDRS*	Number of active mapping register descriptors in arrays to which ADP$W_MRNREGARY and ADP$W_MRFREGARY point. IOC$REQMAPREG and IOC$RELMAPREG use these fields when allocating and deallocating mapping registers.
ADP$W_DPBITMAP*	Data path allocation bit map. IOC$REQDATAP and IOC$RELDATAP read and write this field. The VAX/VMS adapter 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$W_MRNFENCE*	Boundary marker for array specified by ADP$W_MRNREGARY; contains −1.
ADP$W_MRNREGARY*	Mapping register "number of registers" array of 124 words. The number of words, or cells, that are active in this array is contained in ADP$L_MRACTMDRS. Each active cell gives a number of free mapping registers. For each active cell in this array, there is a corresponding first free mapping register number in the array specified by ADP$W_MRFREGARY. Together, these values give the base mapping register and number of free mapping registers for a block of free mapping registers. This information is used to allocate and deallocate mapping registers.
ADP$W_MRFFENCE*	Boundary marker for array specified by ADP$W_MRFREGARY; contains −1.
ADP$W_MRFREGARY*	Mapping register "first register" array of 124 words. The number of currently active cells in this array is contained in ADP$L_MRACTMDRS. Each active cell gives a number of the first free mapping register within a block of free mapping registers. For each active cell in this array, there is a corresponding cell in the number of registers array that gives a number of free mapping registers. Together, these values give the base mapping register and number of free mapping registers for a block of free mapping registers. This information is used to allocate and deallocate mapping registers.
ADP$W_UMR_DIS*	Number of disabled mapping registers. During system initialization, some mapping registers can be disabled so that their corresponding UNIBUS and Q22 bus addresses can be accessed directly through backplane interconnect physical addresses.

Figure A–2 Adapter-Control Block (ADP)

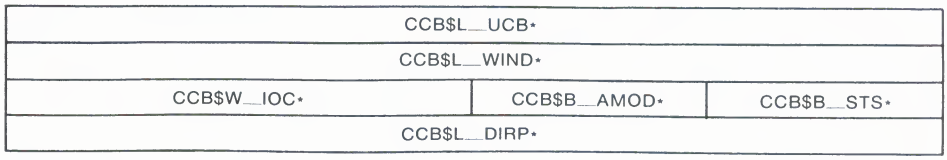


A.3 Channel-Control Block (CCB)

When a process assigns an I/O channel to a device unit with the $ASSIGN system service, EXE$ASSIGN locates a free block among the process' preallocated channel-control blocks (CCBs). EXE$ASSIGN then writes into the CCB a description of the device attached to the CCB's channel.

The channel-control block is illustrated in Figure A–3 and described in Table A–3.

Figure A–3 Channel-Control Block (CCB)



ZK-1780-84

Table A–3 Contents of Channel-Control Block

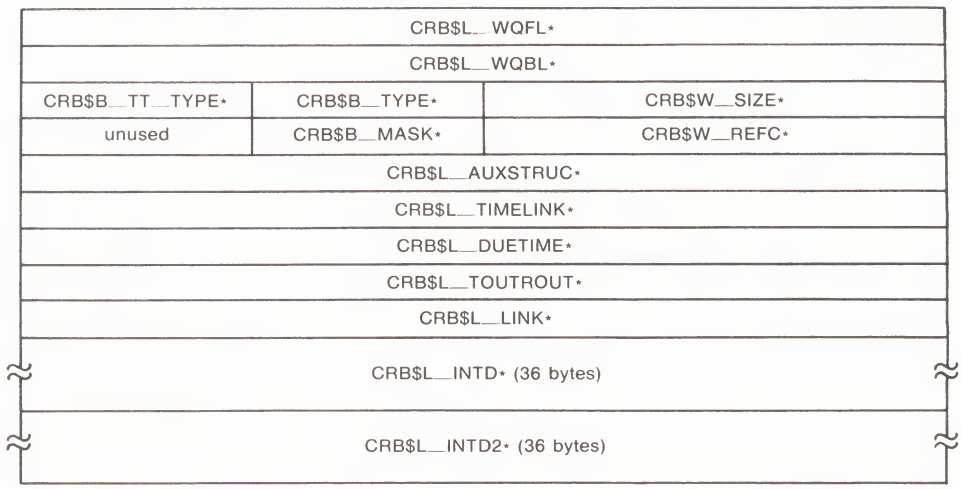
Field Name	Contents
CCB$L_UCB*	Address of UCB of assigned device unit. EXE$ASSIGN writes a value into this field. EXE$QIO reads this field to determine that the I/O request specifies a process I/O channel assigned to a device and to obtain the device's UCB address.
CCB$L_WIND*	Address of window-control block (WCB) for file-structured device assignment. This field is written by an ACP and read by EXE$QIO. A file-structured device's ACP creates a WCB when a process accesses a file on a device assigned to a process I/O channel. The WCB maps the virtual block numbers of the file to a series of physical locations on the device.
CCB$B_STS*	Channel status.
CCB$B_AMOD*	Access mode plus 1 of process at time of channel assignment. EXE$ASSIGN writes the process access mode value into this field.
CCB$W_IOC*	Number of outstanding I/O requests on channel. EXE$QIO increases this field when it begins to process an I/O request that specifies the channel. During I/O postprocessing, the special kernel-mode-AST routine decrements this field. Some FDT routines and EXE$DASSGN read this field.
CCB$L_DIRP*	Address of IRP for requested deaccess. A number of outstanding I/O requests can be pending on the same process I/O channel at one time. If the process that owns the channel issues an I/O request to deaccess the device, EXE$QIO holds the deaccess request until all other outstanding I/O requests are processed.

A.4 Channel-Request Block (CRB)

The activity of each controller in a configuration is described in a channel-request block (CRB). This data structure contains pointers to the wait queue of drivers ready to gain access to a device through the controller. It also stores the entry points to the driver's interrupt-servicing routines and device/controller initialization routines.

The channel-request block is illustrated in Figure A–4 and described in Table A–4.

Figure A–4 Channel-Request Block (CRB)



ZK-1781-84

Table A–4 Contents of Channel-Request Block

Field Name	Contents
CRB$L_WQFL*	Controller data channel wait queue forward link. IOC$REQ*x*CHAN*y* and IOC$REL*x*CHAN insert and remove driver fork block addresses in this field. A channel wait queue contains addresses of driver fork blocks that record the context of suspended drivers waiting to gain control of a controller data channel. If a channel is busy when a driver requests access to the channel, IOC$REQ*x*CHAN*y* 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$REL*x*CHAN dequeues a driver fork block, allocates the channel to the driver, and reactivates the suspended driver fork process. If no drivers are awaiting the channel, IOC$REL*x*CHAN clears the channel busy bit.
CRB$L_WQBL*	Controller channel wait queue backward link. IOC$REQ*x*CHAN*y* and IOC$REL*x*CHAN read and write this field.
CRB$W_SIZE*	Size of CRB. The driver-loading procedure writes this field when the procedure creates the CRB.
CRB$B_TYPE*	Type of data structure. The driver-loading procedure writes the symbolic constant DYN$C_CRB into this field when the procedure creates the CRB.
CRB$B_TT_TYPE*	Type of controller (DZ11 or DZ32) for terminals.
CRB$W_REFC*	UCB 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$B_MASK*	Mask that describes controller status. At present, only one bit, CRB$V_BSY, is defined in this field. IOC$REQ*x*CHAN*y* reads the busy bit to determine whether the controller is free and sets this bit when it allocates the controller data channel to a driver. IOC$REL*x*CHAN clears the busy bit if no driver is waiting to acquire the channel.

Table A–4 (Cont.) Contents of Channel-Request Block

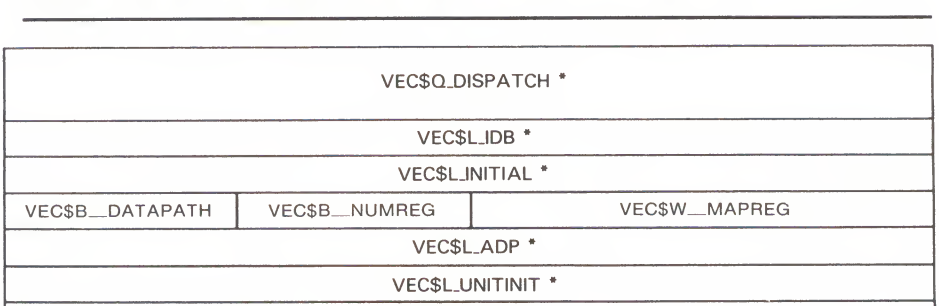
Field Name	Contents
CRB$L_AUXSTRUC*	Address of auxiliary data structure used by device driver to store special controller information. A device driver that wishes to use this field can contain a controller initialization routine that allocates a block of nonpaged dynamic memory and sets this field to point to it.
CRB$L_TIMELINK*	Forward link in queue of CRBs waiting for periodic wakeups. This field points to the CRB$L_TIMELINK field of the next CRB in the list. The CRB$L_TIMELINK field of the last CRB in the list contains zero. The listhead for this queue is IOC$GL_CRBTMOUT. Use of this field is reserved to DIGITAL.
CRB$L_DUETIME*	Time in seconds, relative to EXE$GL_ABSTIM, at which next periodic wakeup associated with CRB is to be delivered. Compute this value by raising IPL to IPL$_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*	Address of routine to be called when periodic wakeup associated with CRB becomes due. The routine must compute and reset the value in CRB$L_DUETIME if another periodic wakeup request is desired. Use of this field is reserved to DIGITAL.
CRB$L_LINK*	Address of secondary CRB (for MASSBUS devices only). This field is written by the driver-loading procedure and read by IOC$REQSCHAN*x* and IOC$RELSCHAN.

CRB$L_INTD*	Interrupt transfer vector. The DPT in every driver for an interrupting device specifies the address of a driver interrupt-servicing routine. The driver-loading procedure writes two instructions in this field:
	`PUSHR  #^M<R0,R1,R2,R3,R4,R5>`
	`JSB  @#^driver-isr-address`
	Direct vector UNIBUS or Q22 bus adapters transfer control to CRB$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-servicing routine.
	On nondirect vector UNIBUS adapters, the UNIBUS adapter interrupt-servicing routine transfers control to CRB$L_INTD+2, which contains the JSB instruction to the driver interrupt-servicing routine. Because the UNIBUS adapter's interrupt-servicing routine has already saved R0 through R5, the PUSHR instruction is bypassed.
	The CRB$L_INTD field is nine longwords long. Figure A–5 and Table A–5 describe the contents of the rest of block.

CRB$L_INTD2*	Second interrupt transfer vector for devices with multiple interrupt vectors. If the DPT in a device driver specifies the address of a second driver interrupt-servicing routine, the driver-loading procedure creates a CRB long enough to contain two INTDx fields of nine longwords each. The first two longwords of the CRB$L_INTD2 field contain a PUSHR instruction and a JSB instruction to the second driver interrupt-servicing 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 I/O adapter information. The block pointed to by CRB$L_INTD is illustrated in Figure A–5 and described in Table A–5.

Figure A–5 Interrupt Transfer Vector Block (VEC)



ZK-1782-84

Table A–5 Fields of CRB$L_INTD

Field Name	Contents
VEC$Q_DISPATCH*	Two interrupt dispatching instructions, written by driver-loading procedure and described above in CRB$L_INTD.
VEC$L_IDB*	Address of IDB for controller. The driver-loading procedure creates an IDB for each CRB and loads the address of the IDB in this field. Device drivers use the IDB address to obtain the virtual addresses of device registers. When a driver interrupt-servicing routine gains control, the top of the stack contains a pointer to this field.
VEC$L_INITIAL*	Address of 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 DPT. 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$W_MAPREG	The following bits are defined within VEC$W_MAPREG: VEC$V_MAPREG — Number of first mapping register allocated to driver that owns controller data channel. IOC$REQMAPREG writes this field when the routine allocates a set of mapping registers to a driver fork process for a DMA transfer. IOC$RELMAPREG reads the field to deallocate a set of mapping registers. Device drivers read this field in calculating the starting address of a UNIBUS or MicroVAX II Q22 bus transfer. VEC$V_MAPLOCK — Mapping register set is permanently allocated (when set).
VEC$B_NUMREG	Number of UNIBUS adapter mapping registers allocated to driver. IOC$REQMAPREG writes this field when the routine allocates a set of mapping registers. IOC$RELMAPREG reads this field to deallocate a set of mapping registers.

Table A–5 (Cont.) Fields of CRB$L_INTD

Field Name	Contents	
VEC$B_DATAPATH	Data path specifier. The bits that make up this field are used as follows:	
	0–4	Number of data path used in DMA transfer. The routine IOC$REQDATAP sets this field when a buffered data path is allocated and clears the field when the data path is released. The routine IOC$LOADUBAMAP copies the contents of this field into the UNIBUS adapter or MicroVAX II mapping registers. These bits also serve as implicit input to the IOC$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 mapping 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 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$RELMAPREG read and write fields in the ADP to allocate and deallocate mapping registers.	
VEC$L_UNITINIT*	Address of 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 DPT. 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 unit initialization field of the DDT (DDT$L_UNITINIT). Other drivers can use either field.	
VEC$L_START*	Reserved to DIGITAL.	
VEC$L_UNITDISC*	Reserved to DIGITAL.	

A.5 Device-Data Block (DDB)

The device-data block (DDB) 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 DDB for each controller during autoconfiguration at system startup and dynamically creates additional DDBs for new controllers as they are added to the system using the SYSGEN command CONNECT. The procedure initializes all fields in the DDB. All the DDBs in the I/O database are linked together in a single-linked list. The contents of IOC$GL_DEVLIST point to the first entry in the list.

VAX/VMS routines and device drivers refer to the DDB.

The device-data block is illustrated in Figure A–6 and described in Table A–6.

Figure A–6 Device-Data Block (DDB)

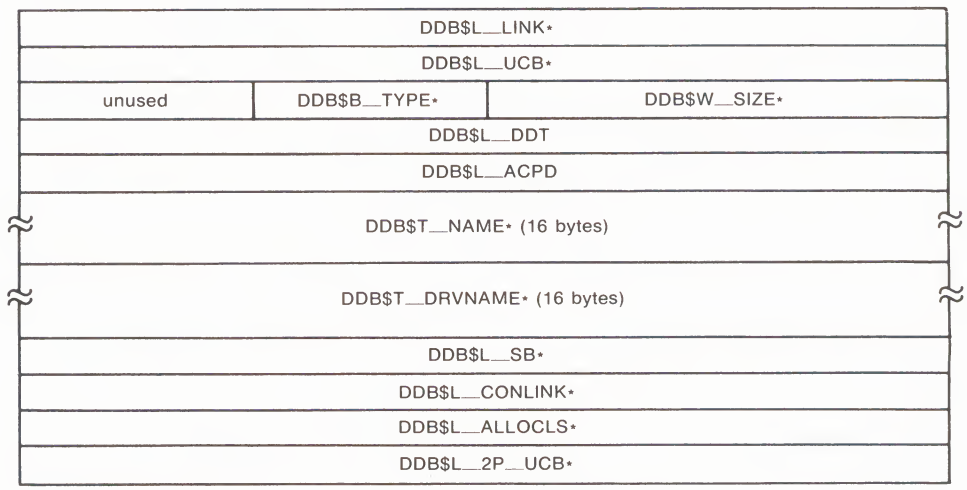


Table A–6 Contents of Device-Data Block

Field Name	Contents
DDB$L_LINK*	Address of next DDB. A zero indicates that this is the last DDB in the DDB chain.
DDB$L_UCB*	Address of UCB for first unit attached to controller.
DDB$W_SIZE*	Size of DDB.
DDB$B_TYPE*	Type of data structure. The driver-loading procedure writes the constant DYN$C_DDB into this field when the procedure creates the DDB.
DDB$L_DDT	Address of DDT. VAX/VMS can transfer control to a device driver only through addresses listed in the DDT, the CRB, and the UCB fork block. The DPT of every device driver must specify a value for this field.
DDB$L_ACPD	Name of default ACP for controller. ACPs that control access to file-structured devices use the high-order byte of this field, DDB$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 listed below. DDB$K_CART Cartridge disk pack DDB$K_PACK Standard disk pack DDB$K_SLOW Floppy disk DDB$K_TAPE Magnetic tape that simulates file-structured device
DDB$T_NAME*	Generic name of devices attached to 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.

Table A–6 (Cont.) Contents of Device-Data Block

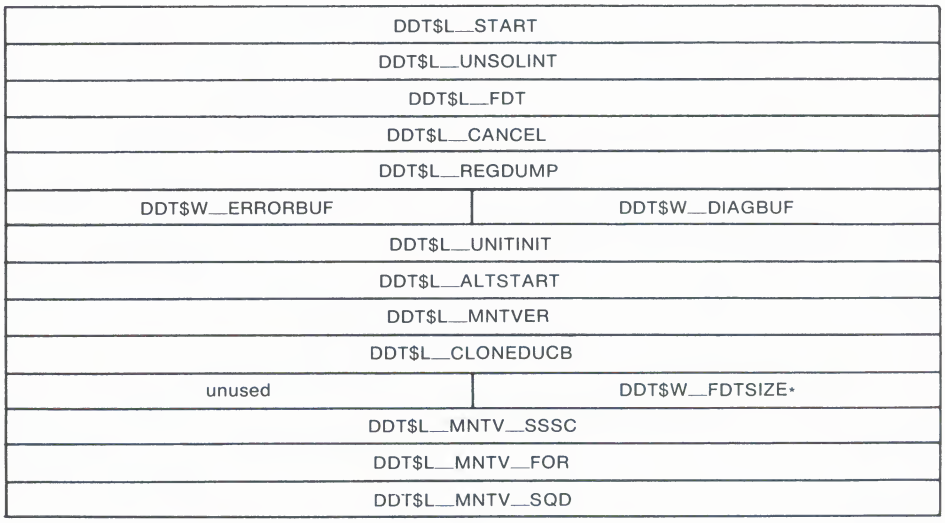
Field Name	Contents
DDB$T_DRVNAME*	Name of device driver for controller. The first byte of this field is the number of characters in the driver name. The remainder of the field contains a string of up to 15 characters in length taken from the DPT in the driver.
DDB$L_SB*	Address of system block
DDB$L_CONLINK*	Address of next DDB in the connection subchain
DDB$L_ALLOCLS*	Allocation class of device
DDB$L_2P_UCB*	Address of first UCB on secondary path. Another name for this field is DDB$L_DP_UCB.

A.6 Driver-Dispatch Table (DDT)

Each device driver contains a driver-dispatch table (DDT). The DDT 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 DDT 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 (DDT)



ZK-1784-84

Table A–7 Contents of Driver-Dispatch Table

Field Name	Contents
DDT$L_START	Entry point to driver start-I/O routine. Every driver must specify this field with the value of the **start** argument to the DDTAB macro. When a device unit is idle and an I/O request is pending for that unit, IOC$INITIATE transfers control to the address contained in this field.
DDT$L_UNSOLINT	Entry point to MASSBUS driver's unsolicited-interrupt-servicing routine. The driver specifies this field with the value of the **unsolic** argument to the DDTAB macro. This field contains the address of a routine that analyzes unexpected interrupts from a device. The standard interrupt-servicing routine, the address of which is stored in the CRB, determines whether an interrupt was solicited by a driver. If the interrupt is unsolicited, the interrupt-servicing routine can call the unsolicited-interrupt-servicing routine.
DDT$L_FDT	Address of driver's FDT. Every driver must specify this field with the value of the **functb** argument to the DDTAB macro. EXE$QIO refers to the FDT to validate I/O-function codes, decide which functions are buffered, and call FDT routines associated with function codes.
DDT$L_CANCEL	Entry point to driver cancel-I/O routine. The driver specifies this field with the value of the **cancel** argument to the DDTAB macro. 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 driver register-dumping routine. The driver specifies this field with the value of the **regdmp** argument to the DDTAB macro. 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 diagnostic buffer. The driver specifies this field with the value of the **diagbf** argument to the DDTAB macro. The value is the size in bytes of a diagnostic buffer for the device. When EXE$QIO preprocesses an I/O request, the routine allocates a system buffer of the size recorded in this field if the user process has diagnostic privileges, specifies a diagnostic buffer in the I/O request, and this field of the DDT contains a nonzero value. IOC$DIAGBUFILL fills the buffer after the I/O operation completes.
DDT$W_ERRORBUF	Size of error-logging buffer. The driver specifies this field as the value of the **erlgbf** argument to the DDTAB macro. The value is the size in bytes of an error-logging buffer for the device. If error logging is enabled and an error occurs during an I/O operation, the driver calls ERL$DEVICERR or ERL$DEVICTMO to allocate and write error-logging data into the error message buffer. IOC$INITIATE and IOC$REQCOM write values into the buffer if an error has occurred.
DDT$L_UNITINIT	Address of 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 and Q22 devices can use either field.
DDT$L_ALTSTART	Address of alternate start-I/O routine. The VAX/VMS routine EXE$ALTQUEPKT initiates the alternate start-I/O routine at this address.

Table A–7 (Cont.) Contents of Driver-Dispatch Table

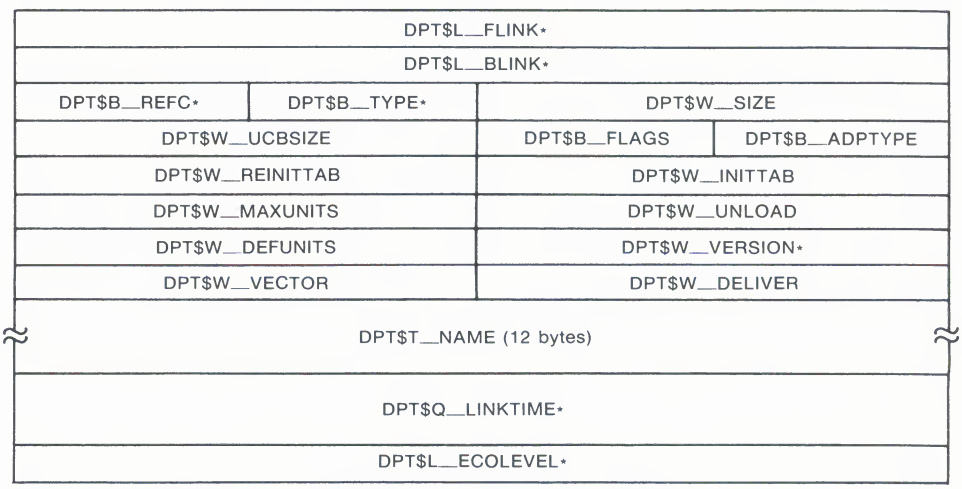
Field Name	Contents
DDT$L_MNTVER	Address of VAX/VMS routine (IOC$MNTVER) called at beginning and end of mount verification operation. The **mntver** argument to the DPTAB macro defaults to this routine. Use of the **mntver** argument to call any routine other than IOC$MNTVER is reserved to DIGITAL.
DDT$L_CLONEDUCB	Address of routine to call when UCB is cloned.
DDT$W_FDTSIZE*	Number of bytes in FDT. The driver-loading procedure uses this field to relocate addresses in the FDT to system virtual addresses.
DDT$L_MNTV_SSSC	Address of routine to call when performing mount verification for a shadow-set-state change.
DDT$L_MNTV_FOR	Address of routine to call when performing mount verification for foreign device.
DDT$L_MNTV_SQD	Address of routine to call when performing mount verification for sequential device.

A.7 Driver-Prologue Table (DPT)

When loading a device driver and its database into virtual memory, the driver-loading procedure finds the basic description of the driver and its device in a driver-prologue table (DPT). The DPT provides the length, name, adapter type, and loading and reloading specifications for the driver.

A device driver creates a DPT by invoking the VAX/VMS macros DPTAB and DPT_STORE. The driver-prologue table is illustrated in Figure A–8 and described in Table A–8.

Figure A–8 Driver-Prologue Table (DPT)



ZK-1785-84

Table A–8 Contents of Driver-Prologue Table

Field Name	Contents
DPT$L_FLINK*	Forward link to next DPT. The driver-loading procedure writes this field. The procedure links all DPTs in the system in a doubly linked list.
DPT$L_BLINK*	Backward link to previous DPT. The driver-loading procedure writes this field.
DPT$W_SIZE	Size in bytes of driver. The DPTAB macro writes this field by subtracting the address of the beginning of the DPT from the address specified as the **end** argument to the DPTAB macro. The driver-loading procedure uses this value to determine the space needed in nonpaged system memory to load the driver.
DPT$B_TYPE*	Type of data structure. The DPTAB macro always writes the symbolic constant, DYN$C_DPT, into this field.
DPT$B_REFC*	Number of DDBs that refer to 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$B_ADPTYPE	Type of adapter used by devices using driver. Every driver must specify the string "UBA" or "MBA" as value of the **adapter** argument to the DPTAB macro. Q22 bus drivers should specify "UBA" as the adapter type. The macro writes the value AT$_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 **flags** argument to the DPTAB macro. The driver-loading procedure modifies the loading and reloading algorithm followed on the basis of 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 to be allocated during driver loading. DPT$M_NOUNLOAD — Driver cannot be reloaded. DPT$V_SCS — SCS code must be loaded with this driver.
DPT$W_UCBSIZE	Size in bytes of UCBs created for device units using driver. Every driver must specify a value for this field as the value of the **ucbsize** argument to the DPTAB macro. The driver-loading procedure allocates blocks of nonpaged system memory of the specified size when creating UCBs for devices associated with the driver.
DPT$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 — Device class UCB$B_DEVTYPE — Device type UCB$W_DEVBUFSIZ — Default buffer size UCB$L_DEVDEPEND — Device-dependent parameters

Table A–8 (Cont.) Contents of Driver-Prologue Table

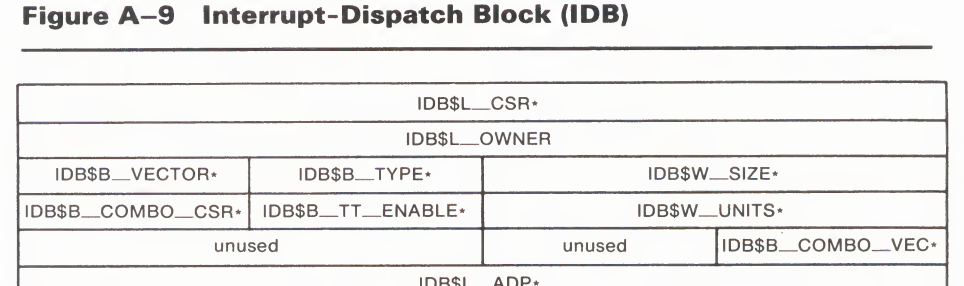
Field Name	Contents
DPT$W_REINITTAB	Offset to driver-reinitialization table. Every driver must specify a list of control-block fields and values to be written into fields at the time that the driver-loading procedure creates the control blocks or loads the driver. The driver invokes the VAX/VMS macro DPT_STORE to specify these fields and their values. Every driver must specify the following field: DDB$L_DDT Driver-dispatch table Other commonly initialized fields are: CRB$L_INTD+4 Interrupt-servicing routine CRB$L_INTD2+4 Second interrupt-servicing routine VEC$L_INITIAL Controller initialization routine VEC$L_UNITINIT Unit initialization routine
DPT$W_UNLOAD	Relative address of driver action routine to be called when driver is reloaded. The driver specifies this field with the value of the **unload** argument to the DPTAB macro. If the driver requires special clean-up processing, such as buffer or mapping register deallocation, before the driver can be reloaded, the driver must specify this field. The driver-loading procedure calls the driver unloading routine before reinitializing all device units associated with the driver.
DPT$W_MAXUNITS	Maximum number of units on 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$W_VERSION*	Version number that identifies format of DPT. The DPTAB macro automatically inserts a value in this field. SYSGEN checks its copy of the version number against the value stored in this field. If the values do not match, an error is generated. To correct the error, reassemble and relink the driver.
DPT$W_DEFUNITS	Number of UCBs that autoconfiguration facility will automatically create. Drivers specify this number with the **defunits** argument to the DPTAB macro. If the driver also gives a value to DPT$W_DELIVER, this field is also the number of times that the autoconfiguration facility calls the unit-delivery routine.
DPT$W_DELIVER	Relative address of unit-delivery routine that autoconfiguration facility calls for number of UCBs specified in DPT$W_DEFUNITS. The driver supplies the address of the unit-delivery routine in the **deliver** argument to the DPTAB macro.
DPT$W_VECTOR	Relative address of driver-specific vector. Use of this field is reserved to DIGITAL.
DPT$T_NAME	Name of device driver. Field is 12 bytes in length. One byte records the length of the name string; the name string can be up to 11 characters in length. Drivers specify this field as the value of the **name** argument to the DPTAB macro. The driver-loading procedure compares the name of a driver to be loaded with the values in this field in all DPTs already loaded into system memory to ensure that it loads only one copy of a driver at a time.
DPT$Q_LINKTIME*	Time and date at which driver was linked, taken from its image header.
DPT$L_ECOLEVEL*	ECO level of driver, taken from its image header.

A.8 Interrupt-Dispatch Block (IDB)

The interrupt-dispatch block (IDB) records controller characteristics. The driver-loading procedure creates and initializes this block when the procedure creates a CRB. The IDB points to the physical controller by storing the virtual address of the CSR. The CSR is the indirect pointer to all device unit registers.

The interrupt-dispatch block is illustrated in Figure A–9 and described in Table A–9.

Figure A–9 Interrupt-Dispatch Block (IDB)



ZK-1786-84

Table A–9 Contents of Interrupt-Dispatch Block

Field Name	Contents
IDB$L_CSR*	Address of CSR. The SYSGEN command CONNECT must specify the address of a device's CSR. The driver-loading procedure writes the system virtual equivalent of this address into the IDB$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$L_OWNER	Address of UCB of device that owns controller data channel. IOC$REQ*x* CHAN*y* writes a UCB address into this field when the routine allocates a controller data channel to a driver. IOC$REL*x* CHAN confirms that the proper driver fork process is releasing a channel by comparing the driver's UCB with the UCB stored in the IDB$L_OWNER field. If the UCB addresses are the same, IOC$REL*x* CHAN allocates the channel to a waiting driver by writing a new UCB address into the field. If no driver fork processes are waiting for the channel, IOC$REL*x*CHAN clears the field. If the controller is a single-unit controller, the unit or controller initialization routine should write the UCB address of the single device into this field.
IDB$W_SIZE*	Size of IDB. The driver-loading procedure writes the constant IDB$K_LENGTH into this field when the procedure creates the IDB.
IDB$B_TYPE*	Type of data structure. The driver-loading procedure writes the symbolic constant DYN$C_IDB into this field when the procedure creates the IDB.

Table A–9 (Cont.) Contents of Interrupt-Dispatch Block

Field Name	Contents
IDB$B_VECTOR*	Interrupt vector number of device, right-shifted by 2 bits. SYSGEN writes a value to this field using either the autoconfiguration database or the value specified in the /VECTOR qualifier to the CONNECT command. Drivers for devices that define the interrupt vector address through a device register must use this field to load that register during unit initialization and reinitialization after a power failure.
IDB$W_UNITS*	Maximum number of units connected to controller. The maximum number of units is specified in the DPT and can be overridden at driver-loading time.
IDB$B_TT_ENABLE*	Reserved for use by VAX/VMS terminal driver.
IDB$B_COMBO_CSR*	Address of start of CSRs for multicontroller device (such as the DMF32). (The name of this field is IDB$B_COMBO_CSR_OFFSET.)
IDB$B_COMBO_VEC*	Address of start of interrupt vectors for multicontroller device. (The name of this field is IDB$B_COMBO_VECTOR_OFFSET.)
IDB$L_ADP*	Address of UNIBUS's ADP. The SYSGEN CONNECT command must specify the nexus number of the I/O adapter used by a device. The driver-loading procedure writes the address of the ADP for the specified I/O adapter into the IDB$L_ADP field.
IDB$L_UCBLST*	List of UCB addresses. The size of this field is the maximum number of units supported by the controller, as defined in the DPT. The maximum specified in the DPT can be overridden at driver load time. 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 $QIO or $QIOW system service, the service creates an I/O-request packet (IRP). The IRP contains a description of the request and receives the status of the I/O processing as it proceeds.

The I/O-request packet is illustrated in Figure A–10 and described in Table A–10.

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 IRPs into a pending I/O queue. IOC$REQCOM reads and writes this field when the routine dequeues IRPs from a pending I/O queue in order to send an IRP to a device driver.
IRP$L_IOQBL	I/O queue backward link. EXE$INSERTIRP and IOC$REQCOM read and write these fields.
IRP$W_SIZE*	Size of IRP. EXE$QIO writes the symbolic constant, IRP$C_LENGTH, into this field when the routine allocates and fills an IRP.
IRP$B_TYPE*	Type of data structure. EXE$QIO writes the symbolic constant DYN$C_IRP into this field when the routine allocates and fills an IRP.

Table A–10 (Cont.) Contents of an I/O-Request Packet

Field Name	Contents
IRP$B_RMOD*	Access mode of process at time of I/O request. EXE$QIO obtains the processor access mode from the PSL and writes the value into this field.
IRP$L_PID*	Process identification of process that issued I/O request. EXE$QIO obtains the process identification from the PCB and writes the value into this field.
IRP$L_AST*	Address of AST routine specified by user in 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 special 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	Parameter sent as argument to AST routine specified by user in I/O request. If the process specifies an AST routine and a parameter to that AST routine in the $QIO call, EXE$QIO writes the parameter in this field. During I/O postprocessing, the special kernel-mode-AST routine queues a 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$L_WIND	Address of window-control block (WCB) that describes file being accessed in I/O request. EXE$QIO writes this field if the I/O request refers to a file-structured device. The ACP reads this field. When a process gains access to a file on a file-structured device or creates a logical link between a file and a process I/O channel, the device ACP creates a WCB that describes the virtual-to-logical mapping of the file data on the disk. EXE$QIO stores the address of this WCB in the IRP$L_WIND field.
IRP$L_UCB*	Address of UCB for device assigned to process I/O channel. EXE$QIO copies this value from the CCB.
IRP$W_FUNC	I/O-function code that identifies function to be performed for I/O request. The I/O request call specifies an I/O-function code; EXE$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 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 process when I/O request was issued. EXE$QIO obtains a value for this field from the PCB. EXE$INSERTIRP reads this field to insert an IRP into a priority-ordered pending I/O queue.
IRP$L_IOSB	Virtual address of process' I/O-status block (IOSB) that receives final status of I/O request at I/O completion. EXE$QIO writes a value into this field if the I/O request call specifies an IOSB address. The I/O postprocessing special kernel-mode-AST routine writes two longwords of I/O status into the IOSB after the I/O operation is complete. When an FDT routine aborts an I/O request by calling EXE$ABORTIO, EXE$ABORTIO fills the IRP$L_IOSB field with zeros so that I/O postprocessing does not write status into the IOSB.
IRP$W_CHAN*	Index number of process I/O channel for request. EXE$QIO writes this field.

Table A–10 (Cont.) Contents of an I/O-Request Packet

Field Name	Contents
IRP$W_STS	Status of I/O request. EXE$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$V_SWAPIO — Swapping-I/O function IRP$V_DIAGBUF — Diagnostic buffer is present IRP$V_PHYSIO — Physical-I/O function IRP$V_TERMIO — Terminal I/O (for priority increment calculation) IRP$V_MBXIO — Mailbox-I/O function IRP$V_EXTEND — An extended IRP is linked to this IRP IRP$V_FILACP — File ACP I/O IRP$V_MVIRP — Mount-verification-I/O function IRP$V_JNL_REMREQ — Remote-I/O (slave) function IRP$V_KEY — IRP$L_KEYDESC contains the address of a key used for encryption.
IRP$L_SVAPTE	For *direct-I/O* transfer, virtual address of first page-table entry (PTE) of I/O-transfer buffer, written here by FDT routine locking process pages; for *buffered-I/O* transfer, address of buffer in system address space, written here by FDT routine allocating buffer. IOC$INITIATE copies this field into the device UCB field UCB$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 transfer or to unlock pages locked for a direct-I/O transfer.
IRP$W_BOFF	Byte offset into first page of direct-I/O transfer. FDT routines calculate this offset and write the field. For buffered-I/O transfers, FDT routines must write the number of bytes to be charged to the process in this field because these bytes are being used for a system buffer. IOC$INITIATE copies this field into the device UCB field UCB$W_BOFF before calling a device driver start-I/O routine. I/O postprocessing uses IRP$W_BOFF in conjunction with IRP$L_BCNT and IRP$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.

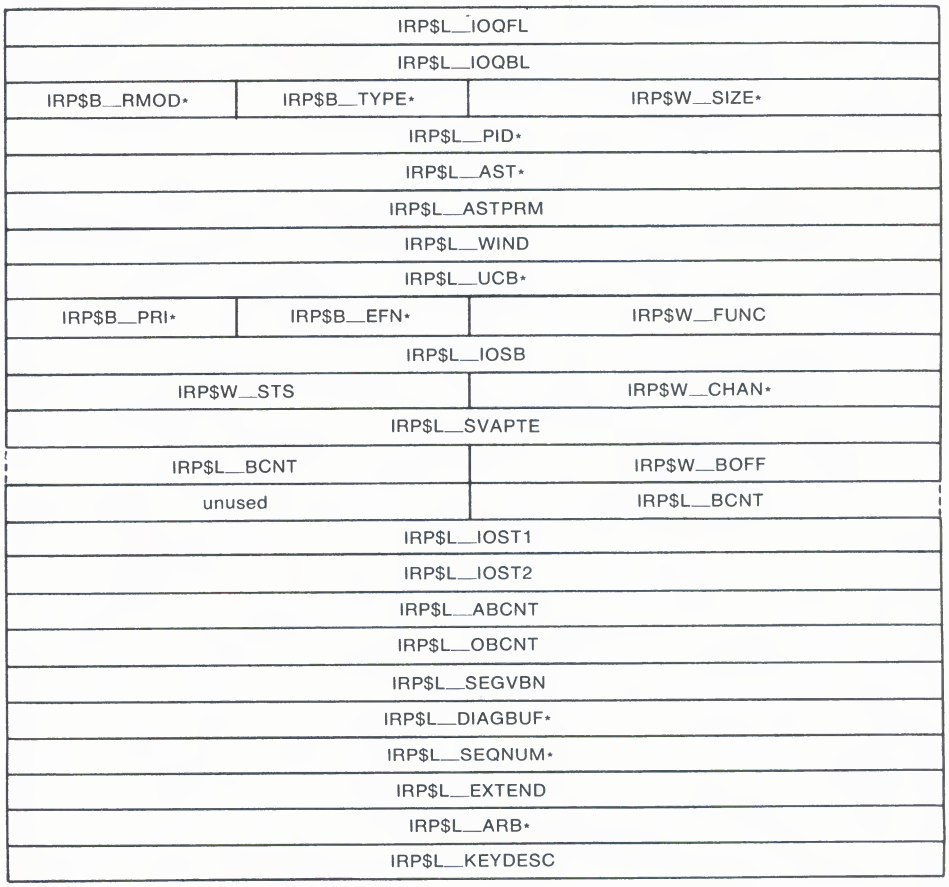
Table A–10 (Cont.) Contents of an I/O-Request Packet

Field Name	Contents
IRP$L_BCNT	Byte count of I/O transfer. FDT routines calculate the count value and write the field. IOC$INITIATE copies the low-order word of this field into UCB$W_BCNT before calling a device driver's start-I/O routine. For a buffered-I/O-read function, I/O postprocessing uses IRP$L_BCNT to determine how many bytes of data to write to the user's buffer. The field IRP$W_BCNT points to the low-order word of this field to provide compatibility with previous versions of VAX/VMS.
IRP$L_IOST1	First I/O status longword. IOC$REQCOM and EXE$FINISHIO(C) write the contents of R0 into this field. The I/O postprocessing routine copies the contents of this field into the user's IOSB. EXE$ZEROPARM copies a 0 and EXE$ONEPARM copies **p1** into this field. This field is a good place to put a $QIO request argument (**p1** through **p6**) or a computed value. This field is also called IRP$L_MEDIA
IRP$L_IOST2	Second I/O status longword. IOC$REQCOM and EXE$FINISHIO(C) write the contents of R1 into this field. The I/O postprocessing routine copies the contents of this field into the user's IOSB. This field is also known as IRP$B_CARCON. IRP$B_CARCON contains carriage control instructions to the driver. EXE$READ and EXE$WRITE copy the contents of P4 of the user's I/O request into this field.
IRP$L_ABCNT	Accumulated bytes transferred in virtual I/O transfer. Read and written by IOC$IOPOST after a partial virtual transfer. The symbol IRP$W_ABCNT points to the low-order word of this field to provide compatibility with previous versions of VAX/VMS.
IRP$L_OBCNT	Original transfer byte count in a virtual I/O transfer. Read by IOC$IOPOST to determine whether a virtual transfer is complete, or whether another I/O request is necessary to transfer the remaining bytes. The symbol IRP$W_OBCNT points to the low-order word of this field to provide compatibility with previous versions of VAX/VMS.
IRP$L_SEGVBN	Virtual block number of current segment of virtual I/O transfer. Written by IOC$IOPOST after a partial virtual transfer.
IRP$L_DIAGBUF*	Address of 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 DDT, and if the process has diagnostic privilege, EXE$QIO copies the buffer address into this field. EXE$QIO allocates a diagnostic buffer in system address space to be filled by IOC$DIAGBUFILL during I/O processing. During I/O postprocessing, the special kernel-mode-AST routine copies diagnostic data from the system buffer into the process diagnostic buffer.
IRP$L_SEQNUM*	I/O transaction sequence number. If an error is logged for the request, this field contains the universal error log sequence number.
IRP$L_EXTEND	Address of IRPE linked to packet. FDT routines write an extension address to this field when a device requires more context than the IRP can accommodate. This field is read by IOC$IOPOST. IRP$V_EXTEND in IRP$W_STS is set if this extension address is used.

Table A–10 (Cont.) Contents of an I/O-Request Packet

Field Name	Contents
IRP$L_ARB*	Address of access-rights block (ARB). This block is located in the PCB and contains the process privilege mask and UIC, which are set up as follows: ARB$Q_PRIV Quadword containing process privilege mask SPARE$L Unused longword ARB$L_UIC Longword containing process UIC
IRP$L_KEYDESC	Address of encryption key.

Figure A–10 I/O-Request Packet (IRP)



ZK-1787-84

A.10 I/O-Request-Packet Extension (IRPE)

I/O-request-packet extensions (IRPEs) hold additional I/O-request information for devices that require more context than the standard IRP can accommodate. IRP extensions are also used when more than one buffer (region) must be locked into memory for a direct-I/O operation, or when a transfer requires a buffer that is larger than 64K. An IRPE provides space for two buffer regions, each with a 32-bit byte count.

FDT routines allocate IRPEs by calling EXE$ALLOCIRP. Driver routines link the IRPE to the IRP, store the IRPE's address in IRP$L_EXTEND and set the bit field IRP$V_EXTEND in IRP$W_STS to show that an IRPE exists for the IRP. The FDT routine initializes the contents of the IRPE. Any fields within the extension not described in Table 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$MODIFYLOCKR). If an error occurs during the locking procedure, the driver must unlock all previously locked regions using MMG$UNLOCK and deallocate the IRPE before returning to the buffer locking routine.

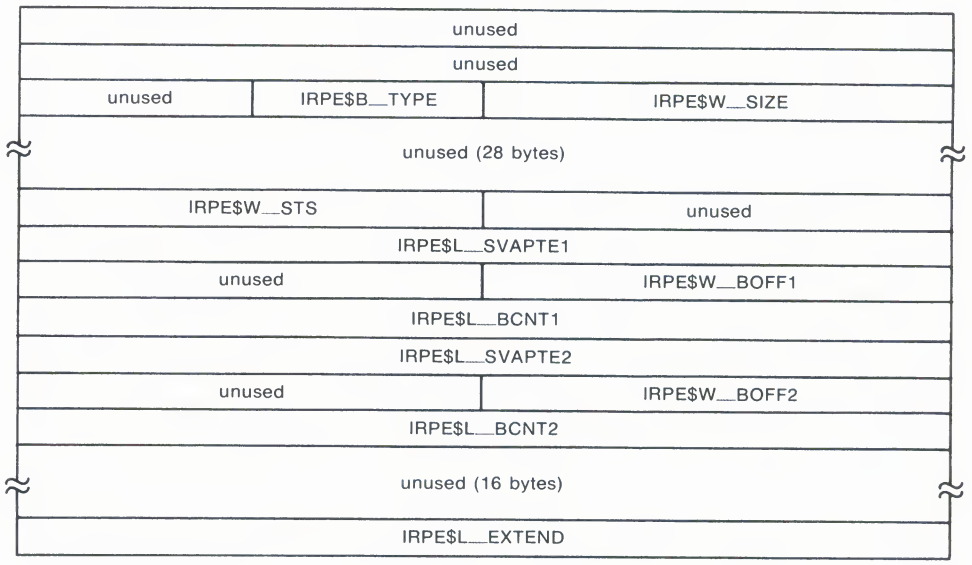
IOC$IOPOST automatically unlocks the pages in region 1 (if defined) and region 2 (if defined) for all the IRPEs linked to the IRP undergoing completion processing. IOC$IOPOST also deallocates all the IRPEs.

The I/O-request-packet extension is illustrated in Figure A–11 and described in Table A–11.

Table A–11 Contents of the I/O-Request-Packet Extension

Field Name	Contents
IRPE$W_SIZE	Size of IRPE. EXE$ALLOCIRP writes the constant IRP$C_LENGTH to this field.
IRPE$B_TYPE	Type of data structure. EXE$ALLOCIRP writes the constant DYN$C_IRP to this field.
IRPE$W_STS	IRPE status field. Bits in the status field describe the following condition: IRPE$V_EXTEND Another IRPE is linked to this one.
IRPE$L_SVAPTE1	System virtual address of page-table entry (PTE) that maps start of region 1. FDT routines write this field. If the region is not defined, this field is zero.
IRPE$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 PTE that maps 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.
IRPE$L_EXTEND	Address of next IRPE for this IRP, if any.

Figure A–11 I/O-Request-Packet Extension (IRPE)



A.11 Object-Rights Block (ORB)

The object-rights block (ORB) is a data structure that describes the rights a process must have in order to access the object with which the ORB is associated.

The ORB is usually allocated when the device is connected by means of SYSGEN's CONNECT command. SYSGEN also sets the address of the ORB in UCB$L_ORB at that time.

When initializing the ORB, device drivers must first zero the ORB, then use the DPT_STORE macro to initialize the fields in the ORB. The object-rights block is illustrated in Figure A–12 and described in Table A–12.

Figure A–12 Object-Rights Block (ORB)

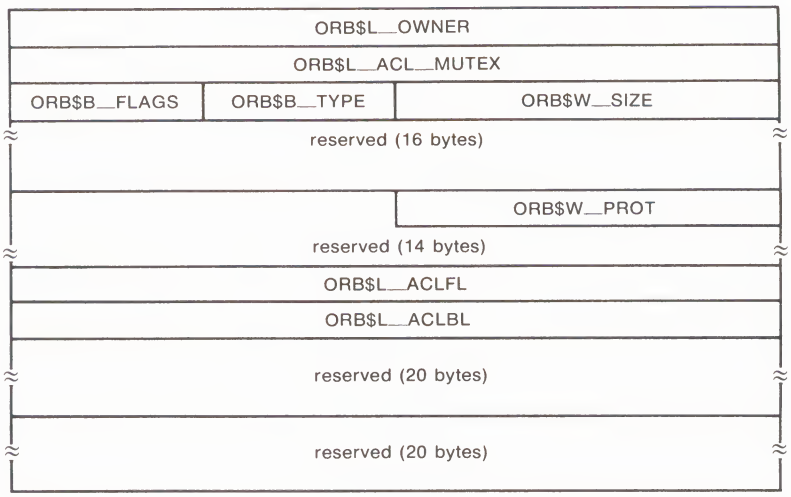


Table A–12 Contents of Object-Rights Block

Field	Use	
ORB$L_OWNER	UIC of object's owner.	
ORB$L_ACL_MUTEX	Mutex for object's ACL, used to control access to ACL for reading and writing.	
ORB$W_SIZE	Size in bytes of ORB (ORB$K_LENGTH).	
ORB$B_TYPE	Type of data structure (DYN$C_ORB).	
ORB$B_FLAGS	Flags needed for interpreting portions of ORB that can have alternate meanings. The following fields are defined within ORB$B_FLAGS:	
	ORB$V_PROT_16	This flag must be set to 1.
	ORB$V_ACL_QUEUE	This flag represents the existence of an ACL queue. The driver should initially set this bit to 0.
	ORB$V_NOACL	This object cannot have an ACL.
ORB$W_PROT	Standard SOGW protection.	
ORB$L_ACLFL	ACL queue forward link. If ORB$V_ACL_QUEUE is 0, this field should contain 0.	
ORB$L_ACLBL	ACL queue backward link. If ORB$V_ACL_QUEUE is 0, this field should contain 0.	

A.12 Unit-Control Block (UCB)

The unit-control block (UCB) is a variable-length block that describes a single device unit. Each device unit on the system has its own UCB. The UCB describes or provides pointers to the device type, controller, driver, device status, and current I/O activity.

During autoconfiguration, the driver-loading procedure creates one UCB for each device unit in the system. A privileged system user can request the driver-loading procedure to create UCBs for additional devices with the SYSGEN command CONNECT as described in Section 14. The procedure creates UCBs of the length specified in the DPT. The driver uses UCB storage located beyond the standard UCB fields for device-specific data and temporary driver storage.

The driver-loading procedure initializes some static UCB fields when it creates the block. VAX/VMS and device drivers can read and modify all nonstatic fields of the UCB. The fields UCB that are present for all devices are illustrated in Figure A–13 and described in Table A–13. The length of the basic UCB is defined by the symbol UCB$K_LENGTH.

UCBs are variable in length depending on the type of device and whether the driver performs error-logging for the device. A number of UCB extensions define symbols employed by the drivers of these devices.

The error-log UCB extension, if present, appears at the end of the standard UCB. The fields in the UCB error-log extension are illustrated in Figure A–14 and described in Table A–14. The symbol UCB$K_ERL_LENGTH defines the end of the extended UCB in this case.

Another extension of the UCB 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 or tape must allow space in the UCB for both the error-log and disk extensions.

For tape devices, the base of the device-dependent UCB must be defined using UCB$K_LCL_TAPE_LENGTH.

The fields are illustrated in Figure A–15 and described in Table A–15.

Another extension to the UCB is a local-disk extension, used by disks that are local to a processor as opposed to disks that are in a cluster with the processor. This UCB extension, if present, appears directly after the UCB's disk extension. For disk devices, the base of the device-dependent UCB must be defined using the symbol UCB$K_LCL_DISK_LENGTH.

The fields in the UCB local-disk extension are illustrated in Figure A–16 and described in Table A–16.

Table A–13 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$W_SIZE*	Size of UCB. The DPT of every driver must specify a value for this field. The driver-loading procedure uses the value to allocate space for a UCB and stores the value in each UCB created. Extra space beyond the standard bytes in a UCB (UCB$K_LENGTH) is for device-specific data and temporary storage.
UCB$B_TYPE*	Type of data structure. The driver-loading procedure writes the constant DYN$C_UCB into this field when the procedure creates the UCB.

Table A–13 (Cont.) Contents of Unit-Control Block

Field Name	Contents
UCB$B_FIPL*	Fork interrupt priority level (IPL) at which device driver usually executes. The DPT of every driver must specify a value for this field. The driver-loading procedure writes the value in the UCB when the procedure creates the UCB. VAX/VMS creates a driver fork process that gains control in a driver start-I/O routine at this IPL. When the driver creates a fork process after an interrupt, VAX/VMS inserts the fork block into a fork queue based on this IPL. A VAX/VMS fork dispatcher executing at UCB$B_FIPL dequeues the fork block and restores control to the suspended driver fork process. All devices that are attached to one I/O adapter and actively compete for shared 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$REQ*x*CHAN*y*, IOC$REQMAPREG, IOC$REQDATAP, and IOC$WFIKPCH. Routines that reactivate suspended drivers include IOC$RELCHAN, IOC$RELMAPREG, IOC$RELDATAP, EXE$FORKDSPTH, and driver interrupt-servicing routines. When a driver interrupt-servicing routine determines that a device is expecting an interrupt, the routine restores control to the saved PC address in the device's UCB.
UCB$L_FR3	Value of R3 at time that VAX/VMS routine suspends driver fork process. The value of R3 is restored just before a suspended driver regains control.
UCB$L_FR4	Value of R4 at time that VAX/VMS routine suspends driver fork process. The value of R4 is restored just before a suspended driver regains control.
UCB$W_BUFQUO*	Buffered-I/O quota if UCB represents mailbox.
UCB$W_SRCADDR*	Local connection number for DECnet.
UCB$L_ORB*	Address of ORB associated with UCB. SYSGEN places the address in this field when you use SYSGEN's CONNECT command.
UCB$L_LOCKID*	ID of lock on device.
UCB$L_CRB*	Address of primary CRB associated with the device. The driver-loading procedure writes this field after it creates the associated CRB. Driver fork processes read this field to gain access to device registers. VAX/VMS routines use UCB$L_CRB to locate interrupt-dispatching code and initialization-routine addresses.
UCB$L_DDB*	Address of DDB associated with device. The driver-loading procedure writes this field when the procedure creates the associated UCB. VAX/VMS routines generally read the DDB field in order to locate device driver entry points, the address of a driver FDT, or the ACP associated with a given device.
UCB$L_PID*	Process identification code of process that has allocated device. Written by the $ALLOC system service.
UCB$L_LINK*	Address of next UCB in chain of UCBs attached to a single controller and associated with a DDB. The driver-loading procedure writes this field when the procedure adds the next UCB. Any 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.

Table A–13 (Cont.) Contents of Unit-Control Block

Field Name	Contents
UCB$L_VCB*	Address of volume-control block (VCB) that describes volume mounted on device. This field is written by the device's ACP and read by EXE$QIOACPPKT and ACPs.
UCB$L_DEVCHAR	First longword of device characteristics bits. The DPT 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 $QIO system service reads the field to determine whether a device is spooled, file-structured, shared, has a volume mounted, and so on. The system defines the following device characteristics: DEV$V_REC Record-oriented device DEV$V_CCL Carriage control device DEV$V_TRM Terminal device DEV$V_DIR Directory-structured device DEV$V_SDI Single directory-structured device DEV$V_SQD Sequential block-oriented device (magnetic tape, for example) DEV$V_SPL Device spooled DEV$V_OPR Operator device DEV$V_RCT Device contains RCT DEV$V_NET Network device DEV$V_FOD File-oriented device (disk and magnetic tape, for example) DEV$V_DUA Dual-ported device DEV$V_SHR Shareable device (used by more than one program simultaneously) DEV$V_GEN Generic device DEV$V_AVL Device available for use DEV$V_MNT Device mounted DEV$V_MBX Mailbox device DEV$V_DMT Device marked for dismount DEV$V_ELG Error logging enabled DEV$V_ALL Device allocated DEV$V_FOR Device mounted as foreign (not file-structured) DEV$V_SWL Device software write-locked DEV$V_IDV Device capable of providing input DEV$V_ODV Device capable of providing output DEV$V_RND Device allowing random access DEV$V_RTM Real-time device DEV$V_RCK Read-checking enabled DEV$V_WCK Write-checking enabled

Table A–13 (Cont.) Contents of Unit-Control Block

Field Name	Contents
UCB$B_DEVCHAR2	Second longword of device characteristics. The DPT of every driver should specify symbolic constant values (defined by the $DEVDEF macro) for this field. The driver-loading procedure writes the field when the procedure creates the UCB. The system defines the following device characteristics: DEV$V_CLU Device available cluster-wide DEV$V_DET Detached terminal DEV$V_RTT Remote-terminal UCB extension DEV$V_CDP Dual-path device with two UCBs DEV$V_2P Two paths known to device DEV$V_MSCP Disk or tape accessed using MSCP DEV$V_SSM Shadow set member DEV$V_SRV Served by MSCP server DEV$V_RED Redirected terminal DEV$V_NNM Name of device (up to 16 characters total) consisting of prefix of node name and dollar sign ($) and string (up to eight characters) consisting of device designation, controller designation, and largest possible unit number
UCB$B_DEVCLASS	Device class. The DPT of every driver should specify a symbolic constant (defined by the $DCDEF macro) for this field. The driver-loading procedure writes this field when the UCB is created. Drivers with set mode and device characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request. The VAX/VMS system defines the following device classes: DC$_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 DPT 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 can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request.

Table A–13 (Cont.) Contents of Unit-Control Block

Field Name	Contents
UCB$W_DEVBUFSIZ	Default buffer size. The DPT can specify a value for this field if relevant. The driver-loading procedure writes the field when the procedure creates the UCB. Drivers with set mode and device characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request. This field is used by RMS for record I/O on nonfile devices.
UCB$L_DEVDEPEND	Contains device-descriptive data that only device driver can interpret. The DPT 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 can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request.
UCB$L_DEVDEPND2	Second longword for device-dependent status. This field is an extension of UCB$L_DEVDEPEND.
UCB$L_IOQFL*	I/O queue listhead forward link. The queue contains the addresses of IRPs waiting for processing on a device. EXE$INSERTIRP inserts IRPs into the pending I/O queue when a device is busy. IOC$REQCOM dequeues IRPs when the device is idle. The queue is a priority queue that has the highest priority IRPs at the front of the queue. Priority is determined by the base priority of the requesting process. IRPs with the same priority are processed first-in/first-out.
UCB$L_IOQBL*	I/O queue listhead backward link. EXE$INSERTIRP and IOC$REQCOM modify the pending I/O queue.
UCB$W_UNIT*	Number of physical device unit. Stored as a binary value. The driver-loading procedure writes a value into this field when the UCB is created. Drivers for multiunit controllers read this field during unit initialization to identify a unit to the controller.
UCB$W_CHARGE*	Mailbox byte count quota charge, if the device is a mailbox.
UCB$L_IRP	Address of IRP currently being processed on device unit by driver fork process. IOC$INITIATE writes the address of an IRP into this field before the routine creates a driver fork process to handle an I/O request. From this field, a driver fork process obtains the address of the IRP being processed. The value contained in this field is valid if the UCB$V_BSY bit in UCB$L_STS is set.
UCB$W_REFC*	Reference count of processes that currently have process I/O channels assigned to 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 device requests hardware interrupts. The DPT 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*	Access mode at which allocation occurred, if device is allocated. Written by the $ALLOC and $DALLOC system services.
UCB$L_AMB*	Associated mailbox UCB pointer. A spooled device uses this field for the address of its associated device. Devices that are nonshareable and not file-oriented can use this field for the address of an associated mailbox.

Table A–13 (Cont.) Contents of Unit-Control Block

Field Name	Contents	
UCB$L_STS	Device unit status (formerly UCB$L_STS). Written by drivers, IOC$REQCOM, IOC$CANCELIO, IOC$INITIATE, IOC$WFIKPCH, IOC$WFIRLCH, EXE$INSIOQ, and EXE$TIMEOUT. This field is read by drivers, the $QIO system service routines, IOC$REQCOM, IOC$INITIATE, and EXE$TIMEOUT.	
	This longword includes the following bits:	
	UCB$V_TIM	Timeout enabled.
	UCB$V_INT	Interrupts expected.
	UCB$V_ERLOGIP	Error log in progress.
	UCB$V_CANCEL	Cancel I/O on unit.
	UCB$V_ONLINE	Device is on line.
	UCB$V_POWER	Power has failed while unit was busy.
	UCB$V_TIMOUT	Unit is timed out.
	UCB$V_INTTYPE	Receiver interrupt.
	UCB$V_BSY	Unit is busy.
	UCB$V_MOUNTING	Device is being mounted.
	UCB$V_DEADMO	Deallocate device at dismount.
	UCB$V_VALID	Software believes volume is valid.
	UCB$V_UNLOAD	Unload volume at dismount.
	UCB$V_TEMPLATE	Template UCB from which other UCBs for this device are made. The $ASSIGN system service checks this bit in the requested UCB and, if the bit is set, creates a UCB from the template. The new UCB is assigned instead.
	UCB$V_MNTVERIP	Mount verification in progress.
	UCB$V_WRONGVOL	Volume name does not match name in the VCB.
	UCB$V_DELETEUCB	Delete this UCB when the value in UCB$W_REFC becomes zero.
	UCB$V_LCL_VALID	The volume on this device is valid on the local node.
	UCB$V_SUPMVMSG	Suppress mount-verification messages if they indicate success.
	UCB$V_MNTVERPND	Mount verification is pending on the device and the device is busy.

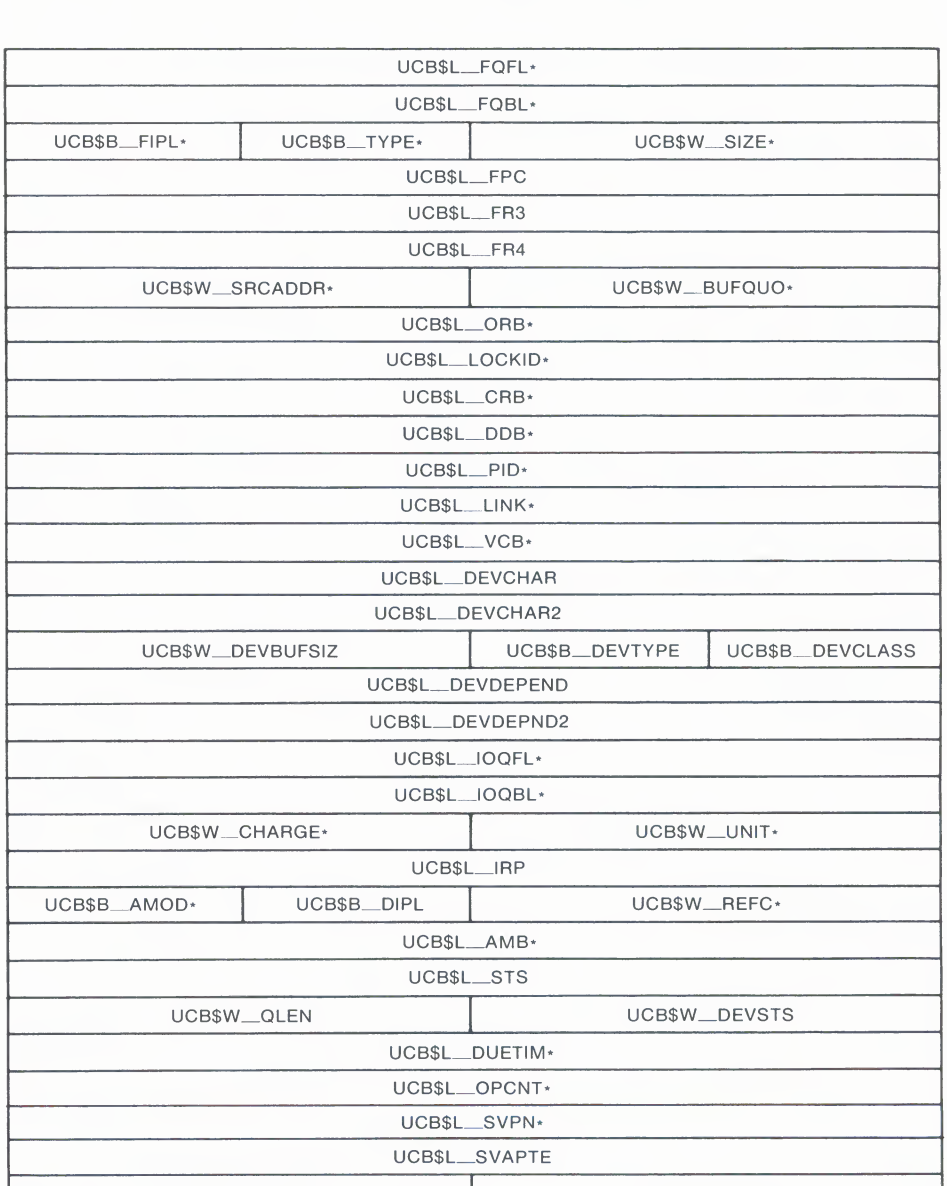
Table A–13 (Cont.) Contents of Unit-Control Block

Field Name	Contents
UCB$W_DEVSTS	Device-dependent status. Read and written by device drivers. The system defines the following status bits: UCB$V_JOB — Job-controller has been notified. UCB$V_TEMPL_BSY — Template UCB is busy. UCB$V_PRMMBX — Device is a permanent mailbox. UCB$V_DELMBX — Mailbox is marked for deletion. UCB$V_SHMMBS — Device is shared-memory mailbox. Disk drivers use three bits in UCB$W_DEVSTS as follows: UCB$V_ECC — ECC correction made. UCB$V_DIAGBUF — Diagnostic buffer is specified. UCB$V_NOCNVRT — No logical block number to media address conversion.
UCB$W_QLEN	Length of queue of IRPs to which UCB$L_IOQFL points.
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 database once per second. If the timeout has occurred and timeouts are enabled for the device, EXE$TIMEOUT calls the device driver timeout handler.
UCB$L_OPCNT*	Count of operations completed on device unit since VAX/VMS booted. IOC$REQCOM writes this field every time the routine inserts an IRP into the I/O postprocessing queue.
UCB$L_SVPN*	Index to virtual address of system PTE permanently allocated to device by driver-loading procedure. The system virtual address of the page described by this index can be calculated by the formula: $(index * 200_{16}) + 80000000_{16}$ If a DPT specifies DPT$M_SVP in the **flags** argument to the DPTAB macro, the driver-loading procedure allocates a page of nonpaged system memory to the device. The procedure writes the system PTE's index into UCB$L_SVPN when the procedure creates the UCB. This field is used for ECC error correction by disk drivers.
UCB$L_SVAPTE	For *direct-I/O* transfer, virtual address of system PTE for first page to be used in transfer; for *buffered-I/O* transfer, address of system buffer used in transfer. IOC$INITIATE writes this field from IRP$L_SVAPTE before calling a driver start-I/O routine. Drivers read this value to compute the starting address of a transfer.
UCB$W_BOFF	For *direct-I/O* transfer, byte offset in first page of transfer buffer; for *buffered-I/O* transfer, number of bytes charged to process for transfer. IOC$INITIATE copies this field from the IRP. Drivers read the field in calculating the starting address of a DMA transfer. If only part of a DMA transfer succeeds, the driver adjusts the value in this field to be the byte offset in the first page of the data that was not transferred.

Table A–13 (Cont.) Contents of Unit-Control Block

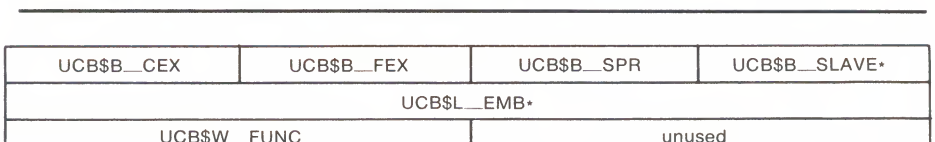
Field Name	Contents
UCB$W_BCNT	Count of bytes in I/O transfer. IOC$INITIATE copies this field from the IRP. Drivers read this field to determine how many bytes to transfer in an I/O operation.
UCB$B_ERTCNT	Error retry count of current I/O transfer. The driver sets this field to the maximum retry count each time it begins I/O processing. Before each retry, the driver decreases the value in this field. If error-logging is occurring, IOC$REQCOM copies the value into the error message buffer.
UCB$B_ERTMAX	Maximum error retry count allowed for single I/O transfer. The DPT of some drivers specifies a value for this field. The driver-loading procedure writes the field when the procedure creates the UCB. If error-logging is occurring, IOC$REQCOM copies the value into the error message buffer.
UCB$W_ERRCNT	Number of errors that have occurred on device since VAX/VMS booted. The driver-loading procedure initializes the field to 0 when the procedure creates the UCB. ERL$DEVICERR and ERL$DEVICTMO increment the value in the field and copy the value into an error message buffer. The DCL command SHOW DEVICE displays in its error count column the value contained in this field.
UCB$L_PDT*	Address of port-descriptor table (PDT). This field is reserved for VAX/VMS port drivers.
UCB$L_DDT*	Address of DDT for unit. The driver load procedure writes the contents of DDB$L_DDT for the device controller to this field when it creates the UCB.

Figure A–13 Unit-Control Block (UCB)



ZK-1789-84

Figure A–14 UCB Error-Log Extension

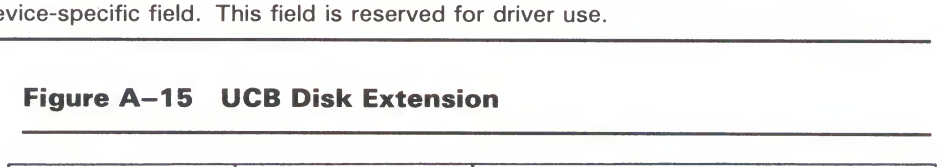


ZK-1790-84

Table A–14 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$L_EMB*	Address of error message buffer. If error logging is enabled and a device/controller error or timeout occurs, the driver calls ERL$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$W_FUNC	I/O-function modifiers. This field is read and written by drivers that log errors.
UCB$L_DPC	Device-specific field. This field is reserved for driver use.

Figure A–15 UCB Disk Extension

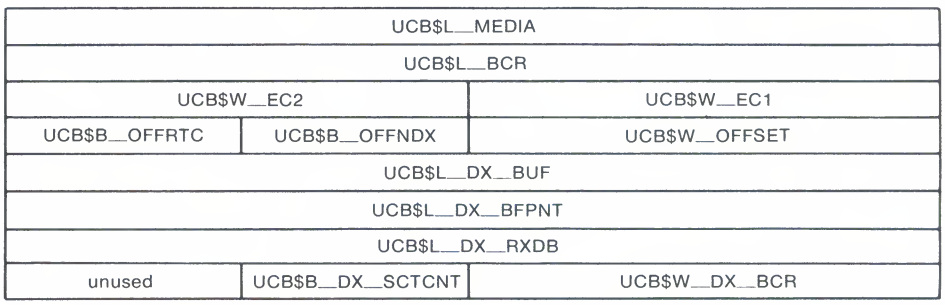


ZK-1791-84

Table A–15 UCB Disk Extension

Field Name	Contents
UCB$W_DIRSEQ	Directory sequence number. If the high-order bit of this word, UCB$V_AST_ARMED, is set, it indicates that the requesting process is blocking ASTs.
UCB$B_ONLCNT	Number of times device has been placed on line since VAX/VMS booted.
UCB$L_MAXBLOCK	Maximum number of logical blocks on random-access device. This field is written by a disk driver during unit initialization and power recovery. This field is also known to tape drivers as UCB$L_RECORD and, as such, contains the number of records between the beginning of the tape and the current position of the tape.
UCB$L_DCCB	Pointer to cache-control block.

Figure A–16 UCB Local Disk Extension



ZK-1792-84

Table A–16 UCB Local Disk Extension

Field Name	Contents
UCB$L_MEDIA	Media address.
UCB$L_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. The symbol UCB$W_BCR points to the low-order word of this field.
UCB$W_EC1	ECC position register. This field records the starting bit number of an error burst. Disk driver register-dumping routines copy the contents of this field into an error-logging or diagnostic buffer. 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$APPLYECC reads the contents of this field to correct disk data.
UCB$W_OFFSET	Current offset register contents.
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.

Table A–16 (Cont.) UCB Local Disk Extension

Field Name	Contents
UCB$B_OFFRTC	Current offset retry count. This field records the number of times to try a particular offset setting in a disk transfer retry.
UCB$L_DX_BUF	Address of sector buffer (used by floppy-disk drivers).
UCB$L_DX_BFPNT	Pointer to current sector (used by floppy-disk drivers).
UCB$L_DX_RXDB	Address of saved receiver-data buffer (used by floppy-disk drivers).
UCB$W_BCR	Current floppy byte count (used by floppy-disk drivers).
UCB$B_DX_SCTCNT	Current sector byte count (used by floppy-disk drivers).

B VAX/VMS Macros Invoked by Drivers

This chapter describes the VAX/VMS macros that drivers can use. Optional arguments are enclosed in brackets. If an argument has a default value, that value is shown, separated from the argument name by an equal sign (=).

CASE

The CASE macro generates a CASE instruction and its associated table.

CASE *src ,displist [,type=W] [,limit=0] - [,nmode=S^#]*

src
Source of the index value to be used with the CASE instruction.

displist
List of destinations to which control is to be dispatched, depending on the value of the index.

[type=W]
Data type of **src** (B, W, or L).

[limit=0]
Lower limit of the value of **src**.

[nmode=S^#]
Addressing mode used to reference the case-table entries; the default, short-literal mode, is good for up to 63 entries.

CPUDISP

The CPUDISP macro provides a means for indirectly distinguishing between I/O bus structures based on the type of the CPU that currently uses that bus structure, and transferring control to a specified destination depending on the CPU-type. CPUDISP builds a case table, first forming the appropriate symbolic constants (PR$_SID_TYPE*xxx*), where *xxx* is the CPU-type, as displacement values and branching to a transfer address according to the contents of global symbol EXE$GB_CPUTYPE.

CPUDISP *addrlist ,[environ=VMS] ,continue=NO*

addrlist

A list containing one or more specifications of the following format:

<**CPU-type, destination**>

The CPUDISP macro accepts only the following values for the **CPU-type**: 8NN (for VAX 8800), 790 (for VAX 8600 and VAX 8650), 8SS (for VAX 8200), 780, 750, 730, UV1, and UV2. The parameter **destination** contains the address to which the code generated by the invocation of the CPUDISP macro passes control to continue with CPU-specific processing.

[environ=VMS]

The environment in which the CPUDISP macro has been invoked. There is no need for driver code to alter the default value of this argument.

continue=NO

Specifies whether execution should continue at the line immediately after the CPUDISP macro if the value at EXE$GB_CPUTYPE does not correspond to any of the values specified as the **CPU-type** in the **addrlist** argument. A fatal bugcheck of UNSUPRTCPU occurs if the dispatching code does not find the executing processor identified in the **addrlist** and the value of **continue** is NO.

Caution: **CPUDISP exists as a temporary means of dispatching to code conditional to the type of the executing processor. Although, it currently functions to distinguish between the I/O bus configurations used by each processor, it most likely will not continue to do so as processors migrate to the various I/O bus configurations.**

DDTAB

The DDTAB macro generates a driver-dispatch table (DDT) labeled devnam$DDT.

DDTAB *devnam ,[start=IOC$RETURN] -*
,[unsolic=IOC$RETURN] -
,functb [,cancel=IOC$RETURN] -
[,regdmp=IOC$RETURN] -
[,diagbf=0] [,erlgbf=0] -
[,unitinit=IOC$RETURN] -
[,altstart=IOC$RETURN] -
[,mntver=IOC$MNTVER] -
[,cloneducb=IOC$RETURN]

devnam
Generic name of the device.

[start=IOC$RETURN]
Address of start-I/O routine.

[unsolic=IOC$RETURN]
Address of unsolicited-interrupt-servicing routine.

functb
Address of FDT.

[cancel=IOC$RETURN]
Address of cancel-I/O routine.

[regdmp=IOC$RETURN]
Address of register-dumping routine.

[diagbf=0]
Length in bytes of the diagnostic buffer.

[erlgbf=0]
Length in bytes of the error-logging buffer.

[unitinit=IOC$RETURN]
Address of unit-initialization routine.

[altstart=IOC$RETURN]
Address of alternate start-I/O routine.

[mntver=IOC$MNTVER]
Address of mount-verification routine; the default is suitable for all single-stream disk drives.

[cloneducb=IOC$RETURN]
Address of routine called when a UCB is cloned by the $ASSIGN system service.

$DEF

Drivers use the $DEF macro to define a data-structure field. $DEF can only be used within the scope of a $DEFINI macro.

$DEF *sym [,alloc] [,siz]*

sym
Name of the symbol by which the field is to be accessed.

[alloc]
Block-storage-allocation directives, one of the following: .BLKB, .BLKW, .BLKL, .BLKQ, or .BLKO

You can define a second symbolic name for the same field by using the $DEF macro a second time immediately following the first definition, leaving this argument blank in the second invocation.

[siz]
Number of block-storage units to allocate.

$DEFEND

The $DEFEND macro ends the scope of the $DEFINI macro, thereby ending the definition of fields within the data structure.

$DEFEND *struc*

struc
Name of the structure that is being defined.

$DEFINI

The $DEFINI macro initiates the definition of a data structure. The $DEF macro is used to define fields within this structure, and the $DEFEND macro ends the definition of this structure.

$DEFINI *struc [,gbl=LOCAL] [,dot=0]*

struc
Name of the data structure to be defined.

[gbl=LOCAL]
Specifies whether the symbols defined for this data structure are to be local or global symbols. The default is to make them local.

To make the symbols' definitions global, you must specify GLOBAL for the value of the **gbl** argument.

[dot=0]
Offset from the beginning of the data structure of the first field to be defined. The default is to make the offset zero.

DPTAB

The DPTAB macro generates a driver-prologue table (DPT) in a program section called $$$105_PROLOGUE.

DPTAB *end ,[adapter ,[flags=0] ,[ucbsize] - ,[unload] ,[maxunits=8] ,[defunits=1] - ,[deliver] ,[vector] ,name*

end
Address of end of the driver.

adapter
Type of adapter (UBA, MBA, DR, or NULL).

[flags=0]
Flags used in loading the driver. The following flags are used:

DPT$M_SVP	Indicates that the device requires a permanently allocated system page. This flag causes the driver-loading procedure to allocate a system page-table entry (PTE) for the device permanently. The system's driver-loading procedure writes the virtual address of the system PTE into the system-page field of the UCB (UCB$L_SVPN) during creation of the UCB. Disk drivers use this page table entry during ECC error correction.
DPT$M_NOUNLOAD	Indicates that the driver cannot be reloaded. A driver with this bit set can be unloaded only by bootstrapping the system.

ucbsize
Size in bytes of UCB required by each device that this driver will drive; this field allows drivers to use extensions to the UCB for storage of device-dependent data that describes an I/O operation.

The amount that the UCB is extended varies for each type of driver. Driver routines and VAX/VMS ECC routines interpret fields in the extended part of the UCB.

[unload]
Address of routine in the driver that SYSGEN is to call before unloading the driver and loading a new version of the driver. SYSGEN calls this routine when you use the RELOAD command.

[maxunits=8]
Maximum number of device units that can be connected to the controller. This field affects the size of the IDB created by SYSGEN's CONNECT command.

If this field is omitted, the default is 8 units. You can override the contents of this field by using the /MAXUNITS qualifier with the CONNECT command.

[defunits=1]
Maximum number of UCBs to be created by SYSGEN's AUTOCONFIGURE command (one for each device unit to be configured). The unit numbers assigned are zero through **defunits**–1.

If the **deliver** argument is present, it names a routine that the AUTOCONFIGURE command calls to determine whether to create each unit's data structures automatically.

[deliver]
Address of routine in the driver that determines whether a unit should be configured automatically, the unit-delivery routine.

If this argument is omitted, the AUTOCONFIGURE command creates the number of units specified by the **defunits** argument.

[vector]
Reserved to DIGITAL; address of a driver-specific transfer vector.

name
Name of the device driver.

DPT_STORE

A driver uses the DPT_STORE macro to instruct the system's driver-loading procedure to store values in a table or data structure.

DPT_STORE *type ,offset ,oper ,exp [,pos] [,size]*

type
Type of control block into which the data is to be stored (DDB, UCB, ORB, CRB, or IDB), or a table marker (INIT, REINIT, or END). If this argument is a table marker, no other argument is allowed and the table affected is the DPT.

offset
Offset from the beginning of the data structure at which the data is to be stored. This cannot be more than 255 bytes.

oper
Type of storage operation, one of the following:

Type	Meaning
B	Byte
W	Word
L	Longword
D	Address relative to the beginning of the driver
V	Bit field

If an at-sign character (@) precedes the **oper** argument, then the **exp** argument describes the address of the data with which to initialize the field.

exp
Value with which to initialize the field; if the value specified for the **oper** argument is preceded by an at-sign character (@), then address at which to find the value with which to initialize the field.

[pos]
Position of the bit affected; used only if **oper**=V.

[size]
Size of the bit-field affected; used only if **oper**=V.

DSBINT

The DSBINT macro disables interrupts occurring at or below the specified IPL and saves the current IPL in the specified longword.

DSBINT *[ipl=31][,dst=-(SP)]*

[ipl=31]
IPL at which to block interrupts. If no IPL is specified, the default is IPL 31, which blocks all interrupts.

[dst=-(SP)]
Location in which to save the current IPL. If no destination is specified, the current IPL is pushed on the stack.

ENBINT

The ENBINT macro enables interrupts at a specified IPL or at the IPL stored on the stack.

ENBINT *[src=(SP)+]*

[src=(SP)+]
Address of IPL at which to enable interrupts. If no **src** is specified, the IPL is popped from the top of the current stack.

$EQULST

The $EQULST macro defines a list of symbols and assigns values to the symbols.

$EQULST *prefix ,[gbl=LOCAL] ,init ,[incr=1] ,list*

prefix
Prefix to be used in forming the names of the symbols, and VALUE is the value assigned to the symbol.

[gbl=LOCAL]
Scope of the definition of the symbol, either LOCAL, the default, or GLOBAL.

init
Value to be assigned to the first symbol in the list.

[incr=1]
Increment by which to increase the value of each succeeding symbol in the list. The default is 1.

list
List of symbols to be defined. Each element in the list can have one of the following forms:

<**symbol**> — where **symbol** is the string appended to the prefix, forming the name of the symbol; the value of the symbol is assigned based on the values of **init** and **incr**.
<**symbol,value**> — where **symbol** is the string that is appended to the prefix, forming the name of the symbol, and **value** specifies the value of the symbol.

FORK

Drivers use the FORK macro to create, by calling EXE$FORK, a fork process, in which context the code that follows the macro invocation executes. Unlike the IOFORK macro, the FORK macro does not clear the UCB$V_TIM bit in the field UCB$L_STS.

FORK

When the FORK macro is invoked, the following registers must contain the values listed below:

Register	Contents
R3	Contents of fork's R3
R4	Contents of fork's R4
R5	Address of fork block
(SP)	Address of caller's caller

FUNCTAB

The FUNCTAB macro generates an entry for a function-decision table (FDT).

FUNCTAB *[action],codes*

[action]
Routine to call when the function code specified in the I/O request matches the **codes** argument to the FUNCTAB macro; if this is to be the first or second entry in the table, this argument must not be supplied.

codes
Code or codes for which the routine specified in the **action** argument to the FUNCTAB macro is to be called; the codes are specified as the I/O-function codes of the form IO$_*xxx*, but without the IO$ prefix.

IFNORD

The IFNORD macro uses the PROBER instruction to check the accessibility of the specified range of memory by checking the accessibility of the first and last bytes in that range.

If both of the specified bytes can be read in the specified access mode, the IFNORD macro dispatches control to the destination specified in the **dest** argument. Otherwise, IFNORD passes control to the next in-line instruction.

IFNORD *siz ,adr ,dest [,mode=#0]*

siz
Offset of the last byte to check from the first byte to check, a number less than or equal to 512.

adr
Address of first byte to check.

dest
Address to which IFNORD passes control if both bytes can be read.

[mode=#0]
Mode in which access is to be checked; zero, the default, causes the check to be performed in the mode contained in the previous-mode field of the current PSL.

IFNOWRT

The IFNOWRT macro uses the PROBEW instruction to check the accessibility of the specified range of memory by checking the accessibility of the first and last bytes in that range.

If both of the specified bytes can be written in the specified access mode, IFNOWRT passes control to the address specified in the **dest** argument. Otherwise, it passes control to the next in-line instruction.

IFNOWRT *siz ,adr ,dest [,mode=#0]*

siz
Offset from the first byte to check to the second byte to check; this number must be less than or equal to 512.

adr
Address of first byte to check.

dest
Address to which IFNOWRT passes control if both bytes can be written in the specified access mode.

[mode=#0]
Mode in which to check access to the bytes; zero, the default, causes the check to be made in the mode contained in the previous-mode field of the current PSL.

IFRD

The IFRD macro checks the accessibility of the specified range of memory by checking the accessibility of the first and last bytes in that range.

If either byte cannot be read in the specified access mode, the IFRD macro passes control to the specified destination. Otherwise it passes control to the next in-line instruction.

IFRD *siz ,adr ,dest [,mode=#0]*

siz
Offset from the first byte to check of the second byte to check; only the first and last bytes in the range are checked.

adr
Address of first byte to check.

dest
Address to which IFRD passes control if either byte cannot be read in the specified access mode.

[mode=#0]
Mode in which to check read access; zero, the default, causes the check to be made in the mode contained in the previous-mode field of the current PSL.

IFWRT

The IFWRT macro checks the accessibility of the specified range of memory by checking the accessibility of the first and last bytes in that range.

If either byte cannot be written in the specified access mode, the IFWRT macro passes control to destination. Otherwise, it passes control to the next in-line instruction.

IFWRT *siz ,adr ,dest [,mode=#0]*

siz
Offset from the first byte to check of the second byte to check; only the first and last bytes in the specified range are checked.

adr
Address of first byte to check.

dest
Address to which IFWRT passes control if either byte cannot be written in the specified access mode.

mode=#0
Mode in which access is to be checked; zero, the default, causes the check to be performed in the mode contained in the previous-mode field of the current PSL.

IOFORK

The IOFORK macro calls EXE$IOFORK to create a fork process for a device driver. IOFORK clears the bit UCB$V_TIM in the field UCB$L_STS, whereas the FORK macro does not.

IOFORK

When the IOFORK macro is invoked, the following registers must contain the values listed below:

Register	Contents
R3	Contents of fork's R3
R4	Contents of fork's R4
R5	Address of UCB that will be used as a fork block for the fork process to be created
(SP)	Address of caller's caller

LOADMBA

The LOADMBA macro calls IOC$LOADMBAMAP to load MASSBUS mapping registers. The driver must own the MASSBUS adapter, and thus the mapping registers, before it can invoke LOADMBA.

LOADMBA

When the LOADMBA macro is invoked, the following registers must contain the values listed below:

Register	Contents
R4	Address of MBA's CSR
R5	Address of UCB

Note that the LOADMBA macro destroys the contents of registers R0 through R2.

LOADUBA

The LOADUBA macro calls IOC$LOADUBAMAP to load the UNIBUS adapter's registers. The registers must already be allocated before the LOADUBA macro can be invoked.

LOADUBA

When the LOADUBA macro is invoked, register R5 must contain the address of the UCB. LOADUBA destroys the contents of registers R0 through R2.

PURDPR

The PURDPR macro calls IOC$PURGDATAP to purge a data path.

PURDPR

When the PURDPR macro is invoked, register R5 must contain the address of the UCB.

When PURDPR returns control to its caller, the following registers must contain the values listed below:

Register	Contents
R0	Status of the purge (success or failure)
R1	Contents of data-path register, provided for the use of the driver's register-dumping routine
R2	Address of first mapping register, provided for the use of the driver's register-dumping routine

The PURDPR macro destroys the contents of R3.

RELCHAN

The RELCHAN macro calls IOC$RELCHAN to release all data channels (controllers) allocated to the device.

RELCHAN

When the RELCHAN macro is invoked, R5 must contain the address of the UCB. RELCHAN destroys the contents of registers R0 through R2.

RELDPR

The RELDPR macro calls IOC$RELDATAP to release a UNIBUS data path register allocated to the driver.

RELDPR

When the RELDPR macro is invoked, R5 must contain the address of the UCB. RELDPR destroys the contents of registers R0 through R2.

RELMPR

The RELMPR macro calls IOC$RELMAPREG to release a set of UNIBUS or MicroVAX II Q22 bus mapping registers allocated by the driver.

RELMPR

When the RELMPR macro is invoked, R5 must contain the address of the UCB. RELMPR destroys the contents of R0 through R2.

RELSCHAN

The RELSCHAN macro calls IOC$RELSCHAN to release all secondary data channels allocated by the driver.

RELSCHAN

When the RELSCHAN macro is invoked, R5 must contain the address of the UCB. RELSCHAN destroys the contents of R0 through R2.

REQCOM

The REQCOM macro calls IOC$REQCOM to complete the processing of an I/O request after the driver has finished its portion of the processing.

REQCOM

When the REQCOM macro is invoked, R5 must contain the address of the UCB. REQCOM destroys the contents of R0 through R2.

REQDPR

The REQDPR macro calls IOC$REQDATAP to request a data path in a UNIBUS adapter.

REQDPR

When the REQDPR macro is invoked, the following registers must contain the values listed below:

Register	Contents
R5	Address of UCB
(SP)	Address of caller's caller

The REQDPR macro destroys the contents of R0 through R2.

REQMPR

The REQMPR macro calls IOC$REQMAPREG to obtain UNIBUS or MicroVAX II Q22 bus mapping registers.

REQMPR

When the REQMPR macro is invoked, the following registers must contain the values listed below:

Register	Contents
R5	Address of UCB
(SP)	Address of caller's caller

The REQMPR macro destroys the contents of R0 through R2.

REQPCHAN

The REQPCHAN macro calls IOC$REQPCHANH or IOC$REQPCHANL, depending on the priority specified, to obtain a controller data channel.

REQPCHAN *[pri]*

[pri]
Priority of request. If the priority is HIGH, REQPCHAN calls IOC$REQPCHANH; otherwise it calls IOC$REQPCHANL.

When the REQPCHAN macro is invoked, the following registers must contain the values listed below:

Register	Contents
R5	Address of UCB
(SP)	Address of caller's caller

The REQPCHAN macro destroys the contents of R0 through R2.

REQSCHAN

The REQSCHAN macro calls IOC$REQSCHANH or IOC$REQSCHANL, depending on the priority specified, to obtain a secondary MASSBUS data channel.

REQSCHAN *[pri]*

[pri]
Priority of request. If the priority is HIGH, REQSCHAN calls IOC$REQSCHANH; otherwise it calls IOC$REQSCHANL.

When the REQSCHAN macro is invoked, the following registers must contain the values listed below:

Register	Contents
R5	Address of UCB
(SP)	Address of caller's caller

The REQSCHAN macro destroys the contents of R0 through R2.

SAVIPL

The SAVIPL macro saves the current IPL, as recorded in the processor IPL register (PR$_IPL), in the specified location or on the stack.

SAVIPL *[dest=–(SP)]*

[dest=–(SP)]
Address of longword in which to save the current IPL; the default is to push the IPL on the stack.

SETIPL

The SETIPL macro sets the current IPL by moving the specified value into the processor IPL register (PR$_IPL).

SETIPL *[ipl=31]*

[ipl=31]
Level at which to set the current IPL; the default is IPL 31.

SOFTINT

The SOFTINT macro moves the specified IPL into the processor software-interrupt-request register (PR$_SIRR), thus requesting a software interrupt at that IPL.

SOFTINT *ipl*

ipl
IPL at which interrupt is to occur.

TIMEWAIT

The TIMEWAIT macro checks for a specific state by testing bits for a specified length of time. Use of the TIMEDWAIT macro instead of the TIMEWAIT macro is recommended.

If the state comes into existence during the specified interval, the TIMEWAIT macro places a success code in R0 and returns control to its caller.

If the state does not occur during the specified period, the TIMEWAIT macro places a failure code in R0 and returns control to its caller.

TIMEWAIT *time ,bitval ,source ,context - [,sense=.TRUE.]*

time
Number of 10-microsecond intervals to wait.

bitval
Mask that determines which bits to test.

source
Address of bits to test.

context
Context in which the bits are to be tested (B, W, or L).

[sense=.TRUE.]
If .TRUE., test for one or more of the specified bits set; otherwise test for all bits cleared.

TIMEDWAIT

The TIMEDWAIT macro waits for a period of time for an event or condition to occur. You can specify up to six instructions for this macro to execute in a loop to determine whether the event has occurred.

TIMEDWAIT macro does not read the processor's clock. The interval it waits is approximate and depends upon the processor and the set of instructions you choose for testing to see if the condition exists.

TIMEDWAIT *time [,ins1] [,ins2] [,ins3] [,ins4] - [,ins5] [,ins6] [,donelbl] [,imbedlbl] - [,ublbl]*

time
Number of 10-microsecond intervals by which to multiply the processor-specific value in order to calculate the interval to wait. The processor-specific value is inversely proportional to the speed of the processor, but is never less than 1.

If you do not specify any embedded instructions, increase the value of **time** by 25 percent.

If you specify embedded instructions that take longer to execute than the average, such as the POLYD instruction, they will cause TIMEDWAIT to wait proportionally longer.

[ins1]
First instruction in the loop.

[ins2]
Second instruction in the loop.

[ins3]
Third instruction in the loop.

[ins4]
Fourth instruction in the loop.

[ins5]
Fifth instruction in the loop.

[ins6]
Sixth instruction in the loop.

[donelbl]
Label placed at the address of the instruction at the end of the TIMEDWAIT loop; embedded instructions can pass control to this label in order to pass control to the instruction following the invocation of the TIMEDWAIT macro.

[imbedlbl]
Label placed at the first of the embedded instructions; after executing a processor-specific delay, the TIMEDWAIT macro passes control here to retest for the condition.

[ublbl]
Label placed at the instruction that performs the processor-specific delay after each execution of the loop of embedded instructions; embedded instructions can pass control here in order to skip the execution of the rest of the embedded instructions in a given execution of the embedded loop.

The TIMEDWAIT macro returns a status code (success or failure) in R0. It destroys the contents of R1, and preserves the contents of all other registers.

$VIELD

The $VIELD macro defines bit-fields whose names have the form mod$*x*_**sym**, where *x* can be V, S, or M and **sym** is a value supplied in the **fields** argument to the macro as described below.

$VIELD *mod ,inibit ,fields*

mod

Module in which this bit field is defined; the prefix portion of the name of the symbols to be defined.

inibit

Bit within the field on which the positions of the bits to be defined are based.

fields

One or more fields of the form: <**sym**,[**size**=1],[**mask**]> , where these arguments are defined as follows:

Argument	Meaning
sym	String appended to the string "mod$" to form the name of this bit-field
[size=1]	Size in bits of this bit-field
[mask]	Character "m" if the value of the symbol is to be a bit mask, blank otherwise

_VIELD

The _VIELD macro defines bit-fields whose names have the form mod_*x*_**sym**, where *x* can be V, S, or M and **sym** is a value supplied in the **fields** argument to the macro as described below.

_VIELD *mod ,inibit ,fields*

mod

Module in which this bit field is defined; the prefix portion of the name of the symbols to be defined.

inibit

Bit within the field on which the positions of the bits to be defined are based.

fields

One or more fields of the form: <**sym**,[**size**=1],[**mask**]> , where these arguments are defined as follows:

Argument	Meaning
sym	String appended to the string "mod_" to form the name of this bit-field
[size=1]	Size in bits of this bit-field
[mask]	Character "m" if the value of the symbol is to be a bit mask, blank otherwise

WFIKPCH

The WFIKPCH macro causes a process to wait for an interrupt from a device by calling IOC$WFIKPCH. The process retains ownership of the channel (the controller) while waiting.

The waiting can be ended by the successful completion of a device operation, a device failure, or a timeout. When the interrupt occurs, control returns to the instruction following the WFIKPCH macro.

WFIKPCH *excpt [,time=65536]*

excpt
Name of a device timeout-handling routine; the address of this routine must be within 65,536 bytes of the address at which the WFIKPCH macro is invoked.

[time=65536]
Number of seconds to wait for an interrupt before a device timeout is considered to exist.

When the WFIKPCH macro is invoked, the following registers must contain the values listed below:

Register	Contents
R5	Address of UCB
(SP)	IPL at which control is passed to the caller's caller (generally placed on the stack by a prior invocation of the DSBINT macro)
4(SP)	Address (in the caller's caller) at which to return control

The WFIKPCH macro destroys the contents of registers R0 through R2.

WFIRLCH

The WFIRLCH macro causes a process to wait for an interrupt from a device by calling IOC$WFIRLCH. The process releases ownership of the channel (the controller) while waiting.

The waiting can be ended by the successful completion of a device operation, a device failure, or a timeout. When the interrupt occurs, control returns to the instruction following the WFIRLCH.

WFIRLCH *excpt [,time=65536]*

excpt
Name of a device timeout-handling routine; the address of this routine must be within 65,536 bytes of the address at which the WFIRLCH macro is invoked.

[time=65536]
Number of seconds to wait for an interrupt before a device timeout is considered to exist.

When the WFIRLCH macro is invoked, the following registers must contain the values listed below:

Register	Contents
R5	Address of UCB
(SP)	IPL at which control is passed to the caller's caller
4(SP)	Address (in the caller's caller) at which to return control

The WFIRLCH macro destroys the contents of registers R0 through R2.

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 IPL at which the routine executes, not the IPL at which it is called.

These routines generally return a status value in R0 (for instance, SS$_NORMAL). The low-order bit of this value indicates successful (1) or unsuccessful (0) completion of the routine. Additional information on returned status values appears in the *VAX/VMS System Services Reference Manual* and the *VAX/VMS System Messages and Recovery Procedures Reference Manual*.

COM$DELATTNAST

Module: COMDRVSUB

Driver fork processes call this routine to deliver all the AST-control blocks (ACBs) linked to the specified AST list. COM$DELATTNAST removes all AST control blocks from the specified list and schedules a fork process at IPL$_QUEUEAST to queue each AST to its process.

input

Registers	Contents
R4	Address of specified listhead
R5	Address of UCB
Fields	**Contents**
—	—

IPL at execution: caller's IPL

output

Registers	Contents
—	—
Fields	**Contents**
Specified listhead	0

IPL at exit: caller's IPL

COM$DRVDEALMEM

Module: COMDRVSUB

Drivers can call this routine from any interrupt priority level to deallocate system dynamic memory.

Because the deallocation of nonpaged pool frees a systemwide resource, the deallocation routine (EXE$DEANONPAGED) eventually calls SCH$RAVAIL to notify the scheduler of the availability of the freed memory.[1] Because the scheduler database is synchronized at IPL$_SYNCH, COM$DRVDEALMEM ensures that the interrupt level upon entry to EXE$DEANONPAGED is less than IPL$_SYNCH.

input

Registers	Contents
R0	Address of block to be deallocate
Fields	**Contents**
IRP$W_SIZE	Size of block in bytes

IPL at execution: caller's IPL or IPL$_QUEUEAST (if caller's IPL is greater than IPL$_SYNCH)

output

IPL at exit: caller's IPL

[1] If the size of the block of memory is less than than 24 bytes, or if the block is not properly aligned, a system bugcheck occurs.

COM$FLUSHATTNS

Module: COMDRVSUB

Driver FDT and fork routines call COM$FLUSHATTNS during cancel-I/O operations to flush an attention AST list.

COM$FLUSHATTNS locates all control blocks whose channel number and process identification match those specified as input to the routine, removes them from the specified list, and deallocates them. COM$FLUSHATTNS exits by returning to its caller.

input

Registers	Contents
R4	Address of current PCB
R5	Address of UCB
R6	Number of the assigned channel
R7	Address of AST-control block listhead
Fields	**Contents**
UCB$B_DIPL	Device IPL
PCB$L_PID	Process' ID
PCB$W_ASTCNT	ASTs remaining in quota

IPL at execution: device IPL (UCB$B_DIPL)

output

Registers	Contents
R0	SS$_NORMAL
R1	Destroyed
R2	Destroyed
R7	Destroyed
Fields	**Contents**
PCB$W_ASTCNT	Number of AST control blocks flushed (added to previous contents)
Specified listhead	Updated

IPL at exit: caller's IPL

COM$POST

Module: COMDRVSUB

Drivers call COM$POST after they have completed all device-dependent I/O postprocessing for an I/O request. Drivers generally use this routine to complete the processing of IRPs initiated by the routine EXE$ALTQUEPKT.

COM$POST inserts the IRP into the I/O postprocessing queue headed by IOC$GL_PSBL and returns to the driver fork process. COM$POST operates independently of the device unit: that is, it does not attempt to dequeue another packet nor does it change the busy status of the device.

input

Registers	Contents
R3	Address of IRP
R5	Address of UCB
Fields	**Contents**
IRP$L_MEDIA	Data to be copied to the I/O-status block
IRP$L_MEDIA+4	Data to be copied to the I/O-status block

IPL at execution: caller's IPL (driver fork level or above)

output

Registers	Contents
R0–R1	Destroyed
Fields	**Contents**
UCB$L_OPCNT	Increased by 1

IPL at exit: caller's IPL

COM$SETATTNAST

Module: COMDRVSUB

Driver FDT routines call COM$SETATTNAST to enable or disable attention ASTs, depending on the contents of the **p1** argument to the $QIO system service.

If **p1** contains the address of an AST routine, COM$SETATTNAST allocates a control block that can double as an AST-control block when the AST is delivered. This control block contains the following information:

- Address of the specified AST routine
- Specified AST parameter
- Specified access mode
- Channel number
- 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's extension area. The driver defines this extension area by using the $DEFINI, $DEF, and $DEFEND macros (see Appendix B).

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 which disables ASTs by searching through this linked list, extracting each entry, and deallocating the identified AST-control block.

COM$SETATTNAST exits by returning to its caller.

input

Registers	Contents
R3	Address of IRP
R4	Address of current PCB
R5	Address of UCB
R6	Address of assigned channel's CCB
R7	Address of specified AST-control block listhead
AP	Address of $QIO system service argument list
Fields	**Contents**
IRP$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

output

Registers	Contents
R0	SS$_NORMAL, SS$_EXQUOTA, or SS$_INSFMEM
R1–R2	Destroyed
R3	Address of IRP
R5	Address of UCB
R6–R8	Destroyed
Fields	**Contents**
PCB$W_ASTCNT	Decreased by 1
Specified listhead	Updated

IPL at exit: caller's IPL

ERL$DEVICERR

Module: ERRORLOG

ERL$DEVICERR logs a controller and/or device error by allocating an error message buffer and filling it with data from IRP and UCB. ERL$DEVICERR sets the error type code to device error.

If the driver specifies the address of a register-dumping routine in the **regdmp** argument to the DDTAB macro, ERL$DEVICERR calculates its address from the DDT and calls it. Otherwise, the DDTAB macro supplies the address of IOC$RETURN.

input

Registers	Contents
R5	Address of UCB

output

Registers	Contents
—	—
Fields	**Contents**
UCB$L_EMB	Address of error message buffer
UCB$L_STS	Bit UCB$V_ERLOGIP (error log in progress) is set

ERL$DEVICTMO

Module: ERRORLOG

ERL$DEVICTMO logs a device timeout. This routine performs the same functions and uses the same input and output as ERL$DEVICERR with one exception: the error type code is device timeout.

EXE$ABORTIO

Module: SYSQIOREQ

FDT routines jump to EXE$ABORTIO to finish an I/O operation without returning final I/O status in the IOSB.

EXE$ABORTIO clears IRP$L_IOSB in IRP, clears a bit (ACB$V_QUOTA in IRP$B_RMOD) to prevent a user mode AST, and inserts the IRP in the I/O postprocessing queue headed by IOC$GL_PSBL.

input

Registers	Contents
R0	First longword of status for I/O-status block
R3	Address of IRP
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

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	Increased by 1 if ACB$V_QUOTA was set

IPL at exit: 0 (normal process IPL)

EXE$ALLOCBUF

Module: MEMORYALC

FDT routines call EXE$ALLOCBUF to allocate a buffer from nonpaged pool for a buffered-I/O operation. EXE$ALLOCBUF performs the necessary operations to synchronize access to the system database from the FDT routine, and then calls EXE$ALONONPAGED to attempt to allocate the buffer.

If the process requesting the I/O operation has resource wait mode enabled, EXE$ALLOCBUF can place the process in a resource wait state if sufficient nonpaged pool is unavailable.

The caller must adjust process quotas, generally subtracting the value returned in R1 from JIB$L_BYTCNT. The normal buffered I/O postprocessing routine, initiated by the REQCOM macro, readjusts the quota and also deallocates the buffer. Note that the value returned in R1 and placed at IRP$W_SIZE in the allocated buffer is the size of the requested buffer. The actual size of the allocated buffer is determined according to the algorithms used by EXE$ALONONPAGED and the size of the lookaside list packets. The nonpaged pool deallocation routine (EXE$DEANONPAGED), called in buffered I/O postprocessing, uses similar algorithms when returning memory to nonpaged pool.

input

Registers	Contents
R1	Size of requested buffer in bytes. This value should include the 12 bytes required to store header information.
R4	Address of current PCB
Fields	**Contents**
PCB$V_SSRWAIT	Clear if the process should wait if no memory is available for requested buffer; set if resource wait mode is disabled.

IPL at execution: caller's IPL, IPL$_SYNCH, and the IPL that synchronizes the allocation of nonpaged pool (EXE$GL_NONPAGED). Generally, EXE$GL_NONPAGED contains 11.

output

Registers	Contents
R0	SS$_NORMAL or SS$_INSFMEM
R1	Size of requested buffer in bytes.
R2	Address of allocated buffer
R3	Destroyed
Fields	**Contents**
IRP$W_SIZE (in allocated buffer)	Size of requested buffer in bytes
IRP$B_TYPE (in allocated buffer)	DYN$C_BUFIO

IPL at exit: IPL$_ASTDEL

EXE$ALLOCIRP

Module: MEMORYALC

EXE$ALLOCIRP allocates an IRP from nonpaged dynamic memory. It performs the same functions and has the same input and output as EXE$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 EXE$ALONONPAGED to allocate a block of memory from nonpaged pool. Depending upon the size of the requested block, EXE$ALONONPAGED allocates nonpaged pool either from one of the lookaside lists or from general nonpaged pool.

EXE$ALONONPAGED cannot be called from an IPL above that specified in EXE$GL_NONPAGED (usually 11). EXE$ALONONPAGED does not initialize the header of the allocated block of memory.

input

Registers	Contents
R1	Requested block size in bytes
Fields	**Contents**
None	—

IPL at execution: caller's IPL and the IPL that synchronizes the allocation of nonpaged pool (EXE$GL_NONPAGED). Generally, EXE$GL_NONPAGED contains 11.

output

Registers	Contents
R0	Status code (0 or 1)
R1	If the allocation succeeds from one of the lookaside lists, the value returned in R1 remains the size of the requested block. If the allocated block is from general nonpaged pool, the value in R1 is the requested size, rounded up to a 16-byte multiple.
R2	Address of allocated block
R3	Destroyed
Fields	**Contents**
—	—

IPL at exit: caller's IPL

EXE$ALONPAGVAR

Module: MEMORYALC

Driver fork processes use EXE$ALONPAGVAR, as an alternative to EXE$ALONONPAGED, to allocate pool from general nonpaged pool. EXE$ALONPAGVAR, unlike EXE$ALONONPAGED, makes no attempt to allocate nonpaged pool from the lookaside lists, which makes it suitable for driver fork processes that may afterwards return the allocated block to nonpaged pool in pieces.

EXE$ALONPAGVAR cannot be called from an IPL above that specified in EXE$GL_NONPAGED (usually 11). EXE$ALONPAGVAR does not initialize the header of the allocated block of memory.

input

Registers	Contents
R1	Requested block size in bytes
Fields	**Contents**
None	—

IPL at execution: caller's IPL and the IPL that synchronizes the allocation of nonpaged pool (EXE$GL_NONPAGED). Generally, EXE$GL_NONPAGED contains 11.

output

Registers	Contents
R0	Status code (0 or 1)
R1	Size of requested buffer, rounded up to a 16-byte multiple
R2	Address of allocated block
R3	Destroyed
Fields	**Contents**
—	—

IPL at exit: caller's IPL

EXE$ALOPHYCNTG

Module: MEMORYALC

Driver fork processes use EXE$ALOPHYCNTG to allocate a physically contiguous block of memory. Note that the number of SPT slots available depends on the value of system parameter *SPTREQ*.

Memory allocated by EXE$ALOPHYCNTG must not be deallocated.

input

Registers	Contents
R1	The number of physically contiguous pages to allocate
Fields	**Contents**
None	—

IPL at execution: caller's IPL (must be IPL$_SYNCH)

output

Registers	Contents
R0	SS$_NORMAL, SS$_INSFMEM, or SS$_INSFSPTS
R2	System virtual address of allocated block, if the allocation succeeds
-	All other registers are preserved.
Fields	**Contents**
—	—

IPL at exit: caller's IPL

EXE$ALTQUEPKT

Module: SYSQIOREQ

Driver FDT routines and fork processes call EXE$ALTQUEPKT to send an IRP to a driver's alternate start-I/O routine, and bypass the synchronization usually afforded by the pending I/O queue for the device's UCB.

EXE$ALTQUEPKT passes the address of the IRP to the driver's alternate start-I/O routine without regard for the status of the device unit and returns to its caller.

input

Registers	Contents
R3	Address of IRP
R5	Address of UCB
Fields	**Contents**
DDT$L_ALTSTART	Address of alternate start-I/O routine
UCB$B_FIPL	Driver fork IPL
UCB$L_DDB	Address of unit's DDB
DDB$L_DDT	Address of DDT

IPL at execution: UCB$B_FIPL

output

Registers	Contents
R0–R5	Destroyed
Fields	**Contents**
—	—

IPL at exit: caller's IPL

EXE$BUFFRQUOTA

Module: EXSUBROUT

FDT routines call EXE$BUFFRQUOTA to determine whether a process' buffered byte count quota usage permits the process to be granted additional buffered I/O. EXE$BUFFRQUOTA places the process in a resource wait state if quota usage is too large and the process has resource wait mode enabled.

input

Registers	Contents
R1	Number of requested bytes
R4	Address of current 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$L_BYTLM	Process' byte count limit
JIB$L_BYTCNT	Process' byte count usage quota

IPL at execution: caller's IPL and IPL$_SYNCH

output

Registers	Contents
R0	SS$_NORMAL or SS$_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

EXE$DEANONPAGED deallocates a block of memory and returns it to nonpaged pool. EXE$DEANONPAGED 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 EXE$FINISHIO to finish an I/O operation and return a quadword of final I/O status to the requesting process.

EXE$FINISHIO writes final I/O status into the IRP and inserts the IRP into the I/O postprocessing queue headed by IOC$GL_PSBL.

input

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 IRP
R4	Address of current PCB
R5	Address of UCB

output

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	Increased by 1

EXE$FINISHIOC

Module: SYSQIOREQ

EXE$FINISHIOC performs the same functions and has the same input and output as EXE$FINISHIO with the following exception: EXE$FINISHIOC clears the contents of R1 before storing R0 and R1 in the IRP.

EXE$FORK

Module: FORKCNTRL

EXE$FORK performs the same functions as EXE$IOFORK except that it does not disable timeouts by clearing UCB$V_TIM in the UCB$L_STS field of the UCB.

EXE$INSERTIRP

Module: SYSQIOREQ

EXE$INSERTIRP inserts an IRP into the pending I/O queue of a device's UCB according to the base priority of process that originated the I/O request. It also 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.

input

Registers	Contents
R2	Address of I/O queue listhead for the device
R3	Address of IRP
Fields	**Contents**
—	—

IPL at execution: caller's IPL (fork level or higher)

output

Registers	Contents
R1	Destroyed

IPL at exit: caller's IPL

EXE$INSIOQ

Module: SYSQIOREQ

EXE$INSIOQ examines UCB$V_BSY in UCB$W_STS. If the device is idle (UCB$V_BSY is clear), EXE$INSIOQ calls IOC$INITIATE; if the device is busy, it calls EXE$INSERTIRP.

input

Registers	Contents
R3	Address of IRP
R5	Address of UCB
Fields	**Contents**
UCB$B_FIPL	Driver fork IPL
UCB$V_BSY (in UCB$L_STS)	Determines whether device is busy
UCB$L_IOQFL	Address of device I/O queue listhead

IPL at execution: driver fork level

output

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$V_BSY (in UCB$L_STS)	Set to 1

IPL at exit: original IPL

EXE$INSTIMQ

Module: EXSUBROUT

EXE$INSTIMQ inserts a timer queue element (TQE) into the timer queue. Elements are ordered according to expiration time with those elements closest to due time taking priority.

input

Registers	Contents
R0, R1	Quadword expiration time for new element
R5	Address of timer element to be queued

IPL at execution: IPL$_TIMER

output

Registers	Contents
R2–R3	Destroyed

IPL at exit: IPL$_TIMER

EXE$IOFORK

Module: FORKCNTRL

EXE$IOFORK saves the contents of R3 and R4—and the return PC value from the top of the stack—in the fork block specified by R5. It then inserts the fork block address into a fork queue, headed by SWI$GL_FQFL, corresponding to the IPL stored in the fork block. If the queue is empty, EXE$IOFORK requests a software interrupt at fork IPL.

Unlike, EXE$IOFORK also disables timeouts by clearing UCB$V_TIM in the UCB$L_STS field.

input

Registers	Contents
R5	Address of 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

Registers	Contents
R3	Destroyed
R4	FKB$B_FIPL
Fields	**Contents**
UCB$V_TIM (in UCB$L_STS)	0
FKB$L_FR3 (in UCB)	R3
FKB$L_FR4 (in UCB)	R4
FKB$L_FPC (in UCB)	0(SP)

IPL at exit: caller's IPL

EXE$LCLDSKVALID

Module: SYSQIOFDT

A disk driver's FDT routines call EXE$LCLDSKVALID to process a request for an IO$_PACKACK, IO$_AVAILABLE, or IO$_UNLOAD on a local disk, and to queue the IRP to the device's UCB for driver processing, if needed. This must be the last FDT routine called during preprocessing of these requests.

For an IO$_PACKACK function, EXE$LCLDSKVALID and the driver proceed as follows:

- If UCB$V_LCL_VALID is *not* set, this routine sets UCB$V_LCL_VALID, increments UCB$B_ONLCNT, and queues the IRP to the UCB for driver processing by branching to EXE$QIODRVPKT. The driver's start-I/O routine must subsequently *set* the UCB$V_VALID bit in the field UCB$L_STS.
- If UCB$V_LCL_VALID *is* set, this routine calls EXE$FINISHIO.

For an IO$_UNLOAD or IO$_AVAILABLE function, EXE$LCLDSKVALID and the driver proceed as follows:

- If UCB$V_LCL_VALID *is* set, this routine clears UCB$V_LCL_VALID, decrements the field UCB$B_ONLCNT, and queues the IRP to the UCB for driver processing. The driver's start-I/O routine must subsequently *clear* the UCB$V_VALID bit in the field UCB$L_STS.
- If UCB$V_LCL_VALID is *not* set, this routine calls EXE$FINISHIO.

Note: Because EXE$LCLDSKVALID passes control to EXE$QIODRVPKT if processing is required by the driver, or to EXE$FINISHIO if no further processing is required, its outputs are not returned to its caller.

input

Registers	Contents
R3	Address of IRP
R5	Address of UCB
R7	The number of the bit that the I/O-function code represents
Fields	**Contents**
UCB$V_LCL_VALID (in UCB$L_STS)	If set, the volume is already valid. If not set, the drive is already unloaded or available

IPL at execution: caller's IPL (should be IPL$_ASTDEL)

output

Registers	Contents
R3	Destroyed
R4	FKB$B_FIPL
Fields	**Contents**
UCB$V_LCL_VALID	If the requested function is IO$_PACKACK, this bit is set (in UCB$L_STS); if the requested function is IO$_UNLOAD or IO$_AVAILABLE, this bit is cleared
UCB$B_ONLCNT	If the function is IO$_PACKACK and, on entry to this routine, UCB$V_LCL_VALID was not set, this field is increased by 1; if the function is IO$_UNLOAD or IO$_AVAILABLE and, on entry to this routine, UCB$V_LCL_VALID was set, this field is decreased by 1

IPL at exit: IPL$_SYNCH

EXE$MODIFY

Module: SYSQIOFDT

FDT routines transfer control to this device-independent routine to validate and prepare a user buffer for a DMA read/write operation. Use EXE$MODIFY instead of EXE$READ when you wish your driver to both read from and write to a buffer.

EXE$MODIFY performs the following functions:

- Translates read-logical functions to read-physical functions
- Transfers $QIO system service arguments to the IRP
- 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 $QIO system service, which repeats the request. EXE$MODIFY disables a paging mechanism used during write-only operations.

If EXE$MODIFY completes successfully, it transfers control to EXE$QIODRVPKT. If it 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, Q22 bus, or MASSBUS space, because the attempted mapping causes a system failure.

input

Registers	Contents
R3	Address of IRP
R4	Address of current PCB
R5	Address of UCB assigned to the device unit
R6	Address of 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 first function-dependent $QIO argument (**p1**)
Fields	**Contents**
0(AP)	Virtual address of buffer (**p1**)
4(AP)	Number of bytes in transfer (**p2**)
12(AP)	Carriage control byte (**p4**)
IRP$W_FUNC	I/O-function code

IPL at execution: caller's IPL (IPL$_ASTDEL)

output

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 PTE that maps the first page of the buffer
IRP$L_BCNT	Size of transfer in bytes

IPL at exit: caller's IPL

EXE$MODIFYLOCK

Module: SYSQIOFDT

FDT routines call EXE$MODIFYLOCK to perform buffer processing for a DMA transfer. Use EXE$MODIFYLOCK instead of EXE$READLOCK when you expect your driver to both read from and write to a buffer.

EXE$MODIFYLOCK performs the following functions:

- 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 $QIO system service, which resubmits the request. EXE$MODIFYLOCK disables a paging mechanism used in write-only operations.

If EXE$MODIFYLOCK completes successfully, it returns control to its caller. If it fails, it transfers control to EXE$ABORTIO.

input

Registers	Contents
R0	Starting address of buffer
R1	Size of transfer in bytes
R3	Address of IRP
R4	Address of current PCB
R6	Address of CCB
Fields	**Contents**
—	—

IPL at execution: caller's IPL (IPL$_ASTDEL)

output

Registers	Contents
R0	SS$_NORMAL
R1	Address of PTE that maps the first page of the buffer
R2	Destroyed
R3	Address of IRP
Fields	**Contents**
IRP$L_SVAPTE	Address of PTE that maps the first page of the buffer
IRP$L_BCNT	Size of transfer in bytes
IRP$V_FUNC (in IRP$W_STS)	A value of 1 (indicating a read function)

IPL at exit: caller's IPL

EXE$MODIFYLOCKR

Module: SYSQIOFDT

Drivers typically use EXE$MODIFYLOCKR when they must lock multiple areas into memory for a single I/O request and, if the request is aborted, must unlock these areas.

EXE$MODIFYLOCKR determines whether a process has write access to the buffer pages it requested. If the process *does* have write access, EXE$MODIFYLOCKR then locks the buffer's pages into memory. If it completes successfully, it returns control to its caller.

If EXE$MODIFYLOCKR fails, it calls back the driver as a coroutine, returning an appropriate error status in R0 and preserving all other registers. The driver then performs any necessary procedures not performed by the system as part of its normal queue-I/O request abortion processing, taking care to preserve all registers, including R0 and R1.

When the driver returns to EXE$MODIFYLOCKR with an RSB instruction, the routine 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 processing this I/O request
;
```

EXE$MODIFYLOCKR can fail for any of the following reasons:

- The buffer-access check fails. In this case, the routine returns SS$_ACCVIO to the driver in R0.
- The calling process has an insufficient working set limit to lock all the buffer pages into memory. The routine returns SS$_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.

input

Registers	Contents
R0	Starting address of buffer
R1	Length of the buffer in bytes
R3	Address of IRP
R4	Address of current PCB
R6	Address of CCB

Fields	Contents
—	—

output

Registers	Contents
R0	SS$_NORMAL
R1	Address of PTE that maps the first page of the buffer
R2	Function indicator (set to 1)
R3	Address of IRP
Fields	**Contents**
IRP$L_SVAPTE	Address of PTE that maps the first page of the buffer
IRP$L_BCNT	Size of transfer in bytes
IRP$M_FUNC (in IRP$W_FUNC)	Set to 1

IPL at exit: caller's IPL

EXE$ONEPARM

Module: SYSQIOFDT

This device-independent FDT routine copies a single $QIO parameter into the IRP and calls EXE$QIODRVPKT. (See Section 8.6 for more information about this routine.)

input

Registers	Contents
R3	Address of IRP for the current I/O request
R4	Address of current PCB
R5	Address of UCB
Fields	**Contents**
UCB$B_FIPL	Driver fork IPL
UCB$V_BSY (in UCB$L_STS)	Unit busy flag
UCB$L_IOQFL	Address of unit I/O queue listhead

EXE$QIORETURN

Module: SYSQIOREQ

EXE$QIORETURN sets a success status code in R0, lowers IPL to 0, and returns to the system service dispatcher.

output

Registers	Contents
R0	SS$_NORMAL

IPL at exit: 0

This routine returns by issuing a RET instruction.

EXE$READ

Module: SYSQIOFDT

This device-independent FDT routine validates and prepares a user buffer for a DMA read operation. This routine performs the same functions and has the same input and output as EXE$MODIFY, with the single exception noted in the description of EXE$MODIFY.

EXE$READCHK

Module: SYSQIOFDT

EXE$READCHK checks that a process has write access to the pages in the specified buffer.

If EXE$READCHK completes successfully, it writes the total byte count of the transfer into the IRP (IRP$L_BCNT) and returns control to its caller. If it determines that the process does not have write access to the buffer, it transfers control to EXE$ABORTIO, which terminates the request with access violation status.

input

Registers	Contents
R0	Address of buffer
R1	Size of transfer in bytes
R3	Address of IRP
Fields	**Contents**
—	—

IPL at execution: caller's IPL

output

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$L_BCNT	Size of transfer in bytes
IRP$V_FUNC (in IRP$W_STS)	Value of 1 (indicates a read function)

IPL at exit: caller's IPL

EXE$READCHKR

Module: SYSQIOFDT

EXE$READCHKR performs the same function as EXE$READCHK, except that, on 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 EXE$READLOCK to check buffer accessibility and lock the user buffer in memory for a DMA read transfer. This routine performs the same functions and has the same input and output as EXE$MODIFYLOCK, except that it is used when the driver performs only a read function.

EXE$READLOCKR

Module: SYSQIOFDT

EXE$READLOCKR determines whether a process has write access to the requested buffer pages and, if access is permitted, locks those pages into memory. EXE$READLOCKR performs the same functions and has the same input and output as EXE$MODIFYLOCKR.

EXE$SENSEMODE

Module: SYSQIOFDT

This device-independent FDT routine 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. (See Section 8.5 for additional information.)

input

Registers	Contents
R3	Address of IRP for the current I/O request
R4	Address of current PCB
R5	Address of UCB of the device assigned to the user-specified process I/O channel
R6	Address of CCB that describes the user-specified process I/O channel
R7	Bit number of the user-specified I/O-function code
R8	Address of FDT dispatcher
AP	Address of 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

Registers	Contents
R0	SS$_NORMAL
R1	Device-dependent characteristics copied from UCB$L_DEVDEPEND
Fields	**Contents**
—	—

IPL at exit: caller's IPL

EXE$SETCHAR

Module: SYSQIOFDT

This device-independent FDT routine writes into the device's unit-control block a quadword of information, the address of which is supplied by the **p1** argument to the $QIO request.

If EXE$SETCHAR completes successfully, it places a success code into R0 and transfers control to EXE$FINISHIO. If it fails because the user lacks read access to the characteristics quadword, it transfers control to EXE$ABORTIO with access violation status. (For additional information on EXE$SETCHAR, see Section 8.5.)

input

Registers	Contents
R3	Address of IRP for the current I/O request
R4	Address of current PCB
R5	Address of UCB of the assigned device unit
R6	Address of CCB that describes the specified process I/O channel
R7	Bit number of the I/O-function code
R8	Address of FDT dispatcher
AP	Address of first function-dependent $QIO parameter
Fields	**Contents**
0(AP)	Address of new device characteristics (**p1**)

IPL at execution: caller's IPL

output

Registers	Contents
R0	SS$_NORMAL or SS$_ACCVIO
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

EXE$SETMODE

Module: SYSQIOFDT

This device-independent FDT routine writes into the device's unit-control block a quadword of information, the address of which is supplied by the **p1** argument to the $QIO request.

If EXE$SETMODE completes successfully, it places a success code into R0 and transfers control to EXE$QIODRVPKT. If it fails because the user lacks read access to the characteristics quadword, it transfers control to EXE$ABORTIO with access violation status. (For additional information on EXE$SETMODE, see Section 8.5.)

input

Registers	Contents
R3	Address of IRP for the current I/O request
R4	Address of current PCB
R5	Address of UCB of the device assigned to the user-specified process I/O channel
R6	Address of CCB that describes the user-specified process I/O channel
R7	Bit number of the I/O-function code
R8	Address of FDT entry
AP	Address of first function-dependent $QIO parameter
Fields	**Contents**
p0(AP)	Address of a quadword of device characteristics

IPL at execution: caller's IPL

output

Registers	Contents
R0	SS$_NORMAL or SS$_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

EXE$SNDEVMSG

Module: MBDRIVER

Driver fork processes call EXE$SNDEVMSG to send messages to system processes such as OPCOM.

EXE$SNDEVMSG builds a 32-byte message on the stack that includes the following information:

Bytes	Contents
0–1	Low word of R4 (message type)
2–3	UCB$W_UNIT (device unit number)
4–31	Counted string of device controller name, formatted as *node$controller* for clusterwide devices

EXE$SNDEVMSG then calls EXE$WRTMAILBOX to send the message to a mailbox.

If EXE$SNDEVMSG completes successfully, it exits with an RSB instruction. If it fails, it returns error status to its caller.

EXE$SNDEVMSG can fail for any 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.

input

Registers	Contents
R3	Address of mailbox UCB
R4	Message type
R5	Address of UCB
Fields	**Contents**
UCB$W_UNIT	Device unit number
UCB$L_DDB	Address of device DDB
DDB$T_NAME and mailbox UCB fields	Device controller name

IPL at execution: caller's IPL (must be at or below IPL$_MAILBOX)

output

Registers	Contents
R0	SS$_NORMAL, SS$_MBTOOSML (message too large for mailbox), SS$_MBFULL (mailbox full of messages), SS$_INSFMEM (memory allocation problem), or SS$_NOPRIV (no owner write access)
R1-R4	Destroyed

Fields	Contents
—	—

IPL at exit: caller's IPL

EXE$WRITE

Module: SYSQIOFDT

This device-independent FDT routine validates and prepares a user buffer for a DMA write operation. EXE$WRITE performs the same actions as EXE$MODIFY, and has the same input and output.

EXE$WRITECHK

Module: SYSQIOFDT

EXE$WRITECHK checks that a process has read access to the pages in the specified buffer.

If EXE$WRITECHK completes successfully, it writes the total byte count of the transfer into the IRP (IRP$L_BCNT) and returns control to its caller. If it determines that the process does not have read access to the buffer, it transfers control to EXE$ABORTIO, which terminates the request with access violation status.

input

Registers	Contents
R0	Address of buffer
R1	Size of transfer in bytes
R3	Address of IRP

IPL at execution: caller's IPL

output

Registers	Contents
R0	Buffer address (success)
R1	Size of transfer in bytes
R2	Cleared (indicates a write function)
R3	Address of IRP
Fields	**Contents**
IRP$L_BCNT	Contains transfer size in bytes

IPL at exit: caller's IPL

EXE$WRITECHKR

Module: SYSQIOFDT

EXE$WRITECHKR performs the same functions as EXE$WRITECHK, except that, if it fails, 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 EXE$WRITELOCK to perform buffer processing for a DMA write transfer.

EXE$WRITELOCK calls EXE$WRITECHK and MMG$IOLOCK, and performs the following operations:

- Determines whether the caller has read 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 $QIO system service, which resubmits the request.

If EXE$WRITELOCK completes successfully, it returns control to its caller. If it fails, it transfers control to EXE$ABORTIO.

input

Registers	Contents
R0	Starting address of I/O buffer
R1	Length of transfer in bytes
R3	Address of IRP
R4	Address of current PCB
R6	Address of CCB
Fields	**Contents**
—	—

IPL at execution: caller's IPL (IPL$_ASTDEL)

output

Registers	Contents
R0	SS$_NORMAL
R1	Address of PTE that maps the first page of the buffer
R2	Destroyed
R3	Address of IRP
Fields	**Contents**
IRP$L_SVAPTE	Address of PTE that maps the first page of the buffer
IRP$L_BCNT	Size of transfer in bytes
IRP$V_FUNC (in IRP$W_STS)	A value of 0 (indicating a write function)

IPL at exit: caller's IPL

EXE$WRITELOCKR

Module: SYSQIOFDT

EXE$WRITELOCKR determines whether the process has read access to the requested buffer pages and, if access is permitted, 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), indicating a write function.

EXE$WRTMAILBOX

Module: MBDRIVER

Driver fork processes call EXE$WRTMAILBOX to send messages to mailboxes.

If it completes successfully, EXE$WRTMAILBOX returns success status to its caller. If it fails, it returns an appropriate error status to its caller.

EXE$WRTMAILBOX can fail for any 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.

input

Registers	Contents
R3	Size of message
R4	Message address
R5	Address of mailbox UCB
Fields	**Contents**
Mailbox UCB fields	

IPL at execution: caller's IPL (must be at or below IPL$_MAILBOX)

output

Registers	Contents
R0	SS$_NORMAL, SS$_MBTOOSML (message too large for mailbox), SS$_MBFULL (mailbox full of messages), SS$_INSFMEM (memory allocation problem), or SS$_NOPRIV (no owner write access)
R1–R2	Destroyed

EXE$ZEROPARM

Module: SYSQIOFDT

This device-independent FDT routine clears the parameter field of the IRP and calls EXE$QIODRVPKT. (For additional information, see Section 8.5.)

input

Registers	Contents
R3	Address of IRP for the current I/O request
R4	Address of current PCB
R5	Address of UCB of the device assigned to the user-specified process I/O channel
R6	Address of CCB 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 first function-dependent parameter specified in the user's request

Fields	Contents
—	—

IPL at execution: caller's IPL

output

Registers	Contents
—	—

Fields	Contents
IRP$L_MEDIA	0

IPL at exit: caller's IPL

IOC$ALOUBAMAP(N)

Module: IOSUBNPAG

IOC$ALOUBAMAP and IOC$ALOUBAMAPN both search the mapping register bit map in the ADP to allocate a contiguous set of mapping registers to a driver fork process.

If mapping registers are already permanently allocated to the controller, these routines exit successfully without allocating any mapping registers. Otherwise, they search the mapping register bit map for the required number of contiguous mapping registers, call IOC$ALTUBAMAP, and exit with an RSB instruction.

IOC$ALOUBAMAP calculates the number of needed mapping registers by using the values contained in UCB$W_BCNT and UCB$W_BOFF; it automatically allocates an extra mapping register to be set invalid by IOC$LOADUBAMAP to prevent a wild block transfer. If you use IOC$ALOUBAMAPN, you must specify the number of mapping registers you wish to allocate in R3. Be sure to include the extra mapping register in this value.

input

Registers	Contents
R3	Number of mapping registers to allocate (if the called routine is IOC$ALOUBAMAPN)
R5	Address of UCB
Fields	**Contents**
UCB$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 CRB
CRB$L_INTD+ VEC$L_ADP	Address of device's ADP
VEC$V_MAPLOCK (in CRB$L_INTD+ VEC$W_MAPREG)	Bit that indicates whether mapping registers are permanently allocated to this controller
ADP$W_MRNREGARY, ADP$W_MRFREGARY	Determine which mapping registers are available

IPL at execution: caller's IPL

output

Registers	Contents
R0	1 (success) or 0 (insufficient contiguous mapping registers)
R1–R2	Destroyed

Fields	Contents
CRB$L_INTD+ VEC$B_NUMREG	Number of mapping registers allocated
CRB$L_INTD+ VEC$W_MAPREG	Starting mapping register number
ADP$W_MRNREGARY, ADP$W_MRFREGARY	Bits for allocated mapping registers set to zero.

IPL at exit: caller's IPL

IOC$APPLYECC

Module: IOSUBRAMS

Disk drivers call IOC$APPLYECC to apply an ECC correction to data transferred from a device into memory. IOC$APPLYECC corrects the data by performing an exclusive-OR operation on the data and a correction pattern from the UCB. IOC$APPLYECC also sets a UCB bit (UCB$V_ECC in UCB$W_DEVSTS) to indicate that it has made an ECC correction.

input

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 UCB
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 system PTE for a page that is available for use by driver
UCB$L_SVAPTE	System virtual address of PTE that maps the transfer

IPL at execution: caller's IPL

output

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

This device-independent cancel-I/O routine sets a cancel-I/O bit in the UCB (UCB$V_CANCEL in UCB$L_STS) if the IRP in process on the device originates from the current process on the specified channel and the unit is busy.

input

Registers	Contents
R2	Channel index number
R3	Address of IRP
R4	Address of current PCB
R5	Address of UCB
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	Device busy flag (in UCB$L_STS)

IPL at execution: caller's IPL

output

Registers	Contents
—	—
Fields	**Contents**
UCB$V_CANCEL (in UCB$L_STS)	Set if I/O request should be canceled

IPL at exit: caller's IPL

IOC$DIAGBUFILL

Module: IOSUBNPAG

Driver fork processes call IOC$DIAGBUFILL to fill a diagnostic buffer, if the $QIO request specifies such a buffer.

IOC$DIAGBUFILL saves the system time and final error count in the diagnostic buffer. It then calls the driver register-dumping routine which fills the remainder of the buffer, and exits with an RSB instruction.

input

Registers	Contents
R4	Address of device's CSR
R5	Address of UCB
Fields	**Contents**
UCB$L_IRP	Address of current IRP
IRP$V_DIAGBUF (in IRP$W_STS)	Determines whether diagnostic buffer is present; this bit is set if one exists.
IRP$L_DIAGBUF	Address of diagnostic buffer, if one is present
UCB$B_ERTCNT	Final error retry count
UCB$L_DDB	Address of DDB
DDB$L_DDT	Address of DDT
DDT$L_REGDUMP	Address of driver register-dumping routine
EXE$GQ_SYSTIME	Current system time (time at I/O request completion)
DDT$L_REGDUMP	Address of driver register-dumping routine

IPL at execution: caller's IPL

output

Registers	Contents
R0–R1	Destroyed
R2	Address of DDT
R3	Address of IRP
R4	Device CSR register
R5	Address of UCB
Fields	**Contents**
—	—

IPL at exit: caller's IPL

IOC$INITIATE

Module: IOSUBNPAG

IOC$INITIATE starts a driver fork process to process an IRP.

IOC$INITIATE writes the address of the IRP and its transfer parameters into the UCB and clears the device status bits. If the $QIO system service call specifies a diagnostic buffer, IOC$INITIATE writes the system time into that buffer. It exits with a JMP instruction to the entry point of the driver's start-I/O routine, as specified in the DDT.

input

Registers	Contents
R3	Address of IRP
R5	Address of UCB
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$L_BCNT	Size in bytes of transfer
IRP$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 diagnostic buffer, if one is present
EXE$GQ_SYSTIME	Current system time (when I/O processing began)
UCB$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

output

Registers	Contents
R0–R1	Destroyed
Fields	**Contents**
UCB$L_IRP	Address of IRP
UCB$L_SVAPTE	IRP$L_SVAPTE
UCB$W_BOFF	IRP$W_BOFF
UCB$W_BCNT	IRP$L_BCNT (low-order word)
UCB$V_CANCEL (in UCB$L_STS)	0
UCB$V_TIMOUT (in UCB$L_STS)	0
Diagnostic buffer	Current system time (first quadword)

IPL at exit: caller's IPL

IOC$IOPOST

Module: IOCIOPOST

This interrupt-servicing routine processes IRPs in an I/O postprocessing queue and gains control when the processor grants a software interrupt at IPL$_IOPOST. When the postprocessing queue is empty, IOC$IOPOST dismisses the interrupt with an REI instruction.

IOC$IOPOST performs several discrete tasks to complete either a direct or buffered I/O request:

- For a *buffered-I/O* request, it copies data from the system buffer to the process buffer and releases the system buffer to nonpaged pool.
- For a *direct-I/O* request, it unlocks those process buffer pages that were locked for the I/O transfer. (If an IRPE exists, the unlocked pages include any defined in the IRPE area descriptors.)

IOC$IOPOST performs the following tasks for *both* direct and buffered I/O requests:

- Adjusts direct-I/O or buffer-I/O quota use.
- Sets an event flag if one was specified in the $QIO system service call.
- Copies I/O completion status from the IRP to the process' I/O-status block (if one was specified in the $QIO system service call).
- Queues a user mode AST (if specified) to the process.
- Copies the diagnostic buffer (if specified) from system to process space and releases the system buffer.
- Deallocates the IRP and any IRPEs.

Note that many of these operations are performed by the special kernel-mode AST IOC$IOPOST queues to the process.

input

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$LOADMBAMAP

Module: LOADMREG

Driver fork processes for DMA transfers call IOC$LOADMBAMAP to load the MASSBUS adapter mapping registers required by the current transfer with a page-frame number.

IOC$LOADMBAMAP also loads the transfer size into the MASSBUS adapter's byte count register (MBA$L_BCR) and the byte offset of the transfer into the MASSBUS adapter's virtual address register (MBA$L_VAR). It confirms that enough mapping registers have been allocated and sets the last mapping register invalid to stop a wild transfer.

input

Registers	Contents
R4	Address of MBA configuration register (MBA$L_CSR)
R5	Address of UCB
Fields	**Contents**
UCB$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_SVAPTE	Address of PTE for the first page of the transfer

output

Registers	Contents
R0–R2	Destroyed
Fields	**Contents**
Allocated mapping registers	

IPL at exit: caller's IPL

IOC$LOADUBAMAP(A)

Module: LOADMREG

Driver fork processes for DMA transfers call IOC$LOADUBAMAP or IOC$LOADUBAMAPA to load the UNIBUS or MicroVAX II mapping registers required by the current transfer with a page-frame number, the data-path number, and, optionally, the byte-offset bit and longword-access-enable bit.

IOC$LOADUBAMAP and IOC$LOADUBAMAPA confirm that sufficient mapping registers and a data path have been previously allocated. In addition, they set the valid bit of all allocated mapping registers except the last, which remains clear to prevent a runaway block transfer.

input

Registers	Contents
R5	Address of UCB
Fields	**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$L_CRB	Address of controller's CRB
CRB$L_INTD+ VEC$B_DATAPATH	Number of the data path
VEC$V_LWAE (in CRB$L_INTD+ VEC$B_DATAPATH)	Determines length of buffering. Set if longword buffering used (instead of quadword buffering)
CRB$L_INTD+ VEC$B_NUMREG	Number of mapping registers allocated
CRB$L_INTD+ VEC$L_ADP	Address of ADP
UBA$L_MAP	Address of first UNIBUS or MicroVAX II mapping register
UCB$L_SVAPTE	Address of PTE for the first page of the transfer

output

Registers	Contents
R0–R2	Destroyed
Fields	**Contents**
Allocated mapping registers	Byte offset is set for entry IOC$LOADUBAMAP (never set for IOC$LOADUBAMAPA)

IPL at exit: caller's IPL

IOC$MOVFRUSER

Module: BUFFERCTL

IOC$MOVFRUSER moves a string from a user buffer to a system buffer.

To use this routine, you must first set bit DPT$M_SVP in field DPT$B_FLAGS in the driver's prologue table. (See the description of the DPTAB macro in Appendix B.) This bit causes the system to allocate a system page-table entry (PTE) to the driver. If this PTE is not allocated to the driver, this routine will cause an access violation when it attempts to refer to the location addressed by the contents of the field UCB$L_SVAPTE.

input

Registers	Contents
R1	Address of driver's buffer
R2	The number of bytes to move
R5	Address of UCB
Fields	**Contents**
—	—

IPL at execution: caller's IPL (must be called at fork IPL)

output

IPL at exit: caller's IPL (fork IPL)

IOC$MOVFRUSER2

Module: BUFFERCTL

IOC$MOVFRUSER2 moves a string from a user buffer to a system buffer.

IOC$MOVFRUSER2 is useful for moving blocks of data in several pieces, each piece beginning within a page rather than on a page boundary. To begin, the driver calls IOC$MOVFRUSER. For each subsequent piece, the driver calls this routine.

To use IOC$MOVFRUSER2, first set bit DPT$M_SVP in field DPT$B_FLAGS in the DPT. (See the description of the DPTAB macro in Appendix B.) This bit causes the system to allocate a system page-table entry (PTE) to the driver. If this PTE is not allocated to the driver, this routine will cause an access violation when it attempts to refer to the location addressed by the contents of the field UCB$L_SVAPTE.

input

Registers	Contents
R0	Address of first byte of the string to be moved
R1	Address of driver's buffer
R2	Number of bytes to move
R5	Address of UCB
Fields	**Contents**
—	—

IPL at execution: caller's IPL (must be called at fork IPL)

output

IPL at exit: caller's IPL (fork IPL)

IOC$MOVTOUSER

Module: BUFFERCTL

IOC$MOVTOUSER moves a string from a system buffer to a user buffer.

To use IOC$MOVTOUSER, first set bit DPT$M_SVP in field DPT$B_FLAGS in the DPT. (See the description of the DPTAB macro in Appendix B.) This bit causes the system to allocate a system page-table entry (PTE) to the driver. If this PTE is not allocated to the driver, this routine will cause an access violation when it attempts to refer to the location addressed by the contents of the field UCB$L_SVAPTE.

input

Registers	Contents
R1	Address of driver's buffer
R2	Number of bytes to move
R5	Address of UCB
Fields	**Contents**
—	—

IPL at execution: caller's IPL (must be called at fork IPL)

output

IPL at exit: caller's IPL (fork IPL)

IOC$MOVTOUSER2

Module: BUFFERCTL

IOC$MOVTOUSER2 moves a string from a system buffer to a user buffer.

IOC$MOVTOUSER2 is useful when moving blocks of data in several pieces, each piece beginning within a page rather than on a page boundary. To begin, the driver calls IOC$MOVTOUSER. For each subsequent piece, the driver calls this routine.

To use IOC$MOVTOUSER2, first set bit DPT$M_SVP in field DPT$B_FLAGS in the driver's prologue table. (See the description of the DPTAB macro in Appendix B.) This bit causes the system to allocate a system page-table entry (PTE) to the driver. If this PTE is not allocated to the driver, this routine will cause an access violation when it attempts to refer to the location addressed by the contents of the field UCB$L_SVAPTE.

input

Registers	Contents
R1	Address of driver's buffer
R2	Number of bytes to move
R5	Address of UCB
Fields	**Contents**
—	—

IPL at execution: caller's IPL (must be called at fork IPL)

output

IPL at exit: caller's IPL (fork IPL)

IOC$PURGDATAP

Module: LIOSUB

All device drivers that support DMA transfers, including those on processors that have no buffered data paths (such as the MicroVAX II and MicroVAX I), call IOC$PURGDATAP after a data transfer.[2]

IOC$PURGDATAP performs the following tasks:

1. Obtains the start of adapter register space using the following chain of pointers:

   UCB$L_CRB → CRB$L_INTD+VEC$L_ADP → ADP$L_CSR

2. Extracts the caller's data path number (buffered or direct) from the CRB.
3. Purges the data path if it is a buffered data path.
4. Stores the contents of the data path register in R1.
5. Clears any purge errors in the data path register.
6. Places the appropriate return status in R0.
7. Determines the base of UNIBUS or MicroVAX II bus mapping registers and writes the value into R2.

IOC$PURGDATAP alters R0 through R3, but preserves all other registers.

input

Registers	Contents
R5	Address of UCB
Fields	**Contents**
—	—

IPL at execution: caller's IPL

output

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 start of the I/O bus mapping registers (for the register-dumping routine)
R3	Address of CRB
Fields	**Contents**
—	—

IPL at exit: caller's IPL

[2] A purge of data path 0 is legal and always results in success status.

IOC$RELCHAN

Module: IOSUBNPAG

A driver fork process calls IOC$RELCHAN to release a controller data channel assigned to a device. If the channel wait queue contains waiting fork processes, IOC$RELCHAN dequeues a process, assigns the channel to that process, restores R3 through R5, and reactivates the suspended process.

input

Registers	Contents
R5	Address of UCB
Fields	**Contents**
UCB$L_CRB	Address of CRB
CRB$L_LINK	Address of secondary CRB
CRB$V_BSY (in CRB$B_MASK)	Set if the channel is busy
CRB$L_INTD+ VEC$L_IDB	Address of IDB
IDB$L_OWNER	Address of UCB of channel owner
CRB$L_WQFL	Head of queue of waiting UCBs

IPL at execution: caller's IPL

output

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 IOC$RELDATAP to release a UNIBUS adapter buffered data path. It should not be called unless the driver owns a buffered data path. However, IOC$RELDATAP performs no operation if a data path is permanently allocated to the controller.

If the data path wait queue contains waiting fork processes, IOC$RELDATAP dequeues a process, allocates the data path to that process, restores R3 through R5, and reactivates the suspended process. If the bit-map is corrupted, IOC$RELDATAP signals a bugcheck with the message code INCONSTATE. If it completes successfully, it exits with an RSB instruction.

input

Registers	Contents
R5	Address of UCB
Fields	**Contents**
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of ADP
CRB$L_INTD+ VEC$B_DATAPATH	Data path specifier
VEC$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

output

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 IOC$RELMAPREG to release a set of UNIBUS adapter or MicroVAX II mapping registers. This routine performs no operation if mapping registers are permanently allocated to the controller. IOC$RELMAPREG assumes that the caller is the current owner of the controller data channel.

If the mapping-register-wait queue contains waiting fork processes, IOC$RELMAPREG dequeues a process and attempts to allocate the required set of mapping registers. If successful, it restores R3 through R5 and reactivates the suspended process. If it fails, it reinserts the fork process in the mapping-register-wait queue and dequeues the next process.

IOC$RELMAPREG calls IOC$ALTUBAMAP and IOC$ALOUBAMAP and exits with an RSB instruction.

input

Registers	Contents
R5	Address of UCB
Fields	**Contents**
UCB$L_CRB	Address of CRB
VEC$V_MAPLOCK (in CRB$L_INTD+ VEC$W_MAPREG)	If set, indicates that mapping registers are permanently allocated to the controller
CRB$L_INTD+ VEC$L_ADP	Address of ADP
CRB$L_INTD+ VEC$W_MAPREG	Number of the starting mapping register
CRB$L_INTD+ VEC$B_NUMREG	Number of mapping registers to release
ADP$L_MRQFL	Head of the queue of waiting drivers

IPL at execution: caller's IPL

output

Registers	Contents
R0–R2	Destroyed
Fields	**Contents**
ADP$W_MRNREGARY, ADP$W_MRFREGARY	Mapping registers set to free

IPL at exit: caller's IPL

IOC$RELSCHAN

Module: IOSUBNPAG

IOC$RELSCHAN releases a secondary controller's data channel: that is, the MASSBUS adapter's controller data channel. For more information, refer to Appendix G.

IOC$RELSCHAN has the same inputs and outputs as IOC$RELCHAN.

IOC$REQCOM

Module: IOSUBNPAG

A driver fork process calls this routine after a device I/O operation and all device-dependent processing of an I/O request is complete.

IOC$REQCOM performs the following tasks:

1. Writes R0 and R1 into IRP$L_IOST1 and IRP$L_IOST2.
2. Inserts the IRP into the I/O postprocessing queue headed by IOC$GL_PSBL.
3. Writes final status into the error message buffer, if error logging has been specified, and calls ERL$RELEASEMB.
4. Dequeues an IRP from the pending I/O queue (at UCB$L_IOQFL) and calls IOC$INITIATE. If the queue is empty, it clears the unit busy bit (UCB$V_BSY) to indicate that the device is idle.
5. Exits by branching to IOC$RELCHAN.

input

Registers	Contents
R0	First longword of I/O status
R1	Second longword of I/O status
R5	Address of UCB
Fields	**Contents**
UCB$V_ERLOGIP (in UCB$L_STS)	Set or clear; determines whether error logging should be performed
UCB$L_STS	Final device status
UCB$B_ERTCNT	Final error counters
UCB$L_EMB	Address of error log message buffer
UCB$L_IRP	Address of IRP

IPL at execution: caller's IPL

output

Registers	Contents
R2–R3	Destroyed
All other registers	Destroyed if IOC$INITIATE is called

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	Increased by 1
EMB$B_ERTCNT	UCB$B_ERTCNT
EMB$B_ERTCNT+1	UCB$W_ERRCNT
EMB$W_DV_STS	UCB$W_STS
UCB$V_BSY	Clear

IPL at exit: caller's IPL

IOC$REQDATAP(NW)

Module: IOSUBNPAG

Driver fork processes call IOC$REQDATAP and IOC$REQDATAPNW to request a UNIBUS adapter buffered data path for a DMA transfer. These routines perform no operation if a data path is permanently allocated to the controller.

IOC$REQDATAP and IOC$REQDATAPNW locate a free data path and write the data path number in the CRB. If IOC$REQDATAP cannot allocate a data path, it saves process context by placing the contents of R3, R4 and the PC in the UCB fork block and R5 in the data path wait queue (ADP$L_DPQBL). IOC$REQDATAPNW, by contrast, does not suspend the process to wait for the data path.

input

Registers	Contents
R5	Address of UCB
0(SP)	Caller's return address
4(SP)	Return address of caller's caller
Fields	**Contents**
UCB$L_CRB	Address of CRB
VEC$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 ADP
ADP$W_DPBITMAP	Indicates what data paths are available

IPL at execution: caller's IPL

output

Registers	Contents
R0	SS$_NORMAL
Fields	**Contents**
CRB$L_INTD+ VEC$B_DATAPATH	Data path number
ADP$W_DPBITMAP	Bit for allocated data path clear

IPL at exit: caller's IPL

IOC$REQMAPREG

Module: IOSUBNPAG

Driver fork processes call IOC$REQMAPREG to request a set of UNIBUS adapter or MicroVAX II mapping registers for a DMA transfer. IOC$REQMAPREG performs no operation if mapping registers are permanently allocated to the controller.

IOC$REQMAPREG locates the required number of mapping registers and writes the number of registers and the number of the first register into the CRB. If sufficient mapping registers are not available, IOC$REQMAPREG suspends the process by saving the following context:

- R3 and R4 in UCB$L_FR3 and UCB$L_FR4, respectively
- PC in UCB$L_FPC
- R5 in the mapping-register-wait queue (ADP$L_MRQBL)

input

Registers	Contents
R5	Address of UCB
0(SP)	Return address of caller
4(SP)	Return address of 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$L_ADP	Address of ADP
VEC$V_MAPLOCK (in CRB$L_INTD+ VEC$W_MAPREG)	Determines status of map-lock bit
ADP$W_MRNREGARY, ADP$W_MRFREGARY	Adapter's mapping-register-allocation bit-map

IPL at execution: caller's IPL

output

Registers	Contents
R0	SS$_NORMAL
R1–R2	Destroyed

Fields	Contents
CRB$L_INTD+ VEC$W_MAPREG	The number of the first mapping register allocated
CRB$L_INTD+ VEC$B_NUMREG	Number of mapping registers allocated
ADP$W_MRNREGARY, ADP$W_MRFREGARY	Allocated mapping registers

IPL After execution: caller's IPL

IOC$REQPCHANH

Module: IOSUBNPAG

Driver fork processes call IOC$REQPCHANH to request a channel on the primary controller with high priority.

If the controller data channel is idle, IOC$REQPCHANH writes the UCB address in the IDB and returns the CSR address in R4. If the channel is busy, it suspends the driver fork process, saving its context as follows:

- R3 and R4 in UCB$L_FR3 and UCB$L_FR4, respectively
- The driver's return address (at 0(SP)) in UCB$L_FPC
- R5 in the device controller data channel wait queue (CRB$L_WQFL)

IOC$REQPCHANH exits by issuing an RSB instruction.

input

Registers	Contents
R5	Address of UCB
0(SP)	Return address of caller
4(SP)	Return address of caller's caller
Fields	**Contents**
UCB$L_CRB	Address of CRB
CRB$L_LINK	Address of secondary CRB
CRB$L_INTD+ VEC$L_IDB	Address of IDB
CRB$V_BSY in CRB$B_MASK	Set if channel is busy
IDB$L_CSR	Address of device CSR

IPL at execution: caller's IPL

output

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 IOC$REQPCHANL to request a channel on the primary controller with low priority. IOC$REQPCHANL performs in the same manner as IOC$REQPCHANH, except that, if the driver must wait for the channel, it places the UCB at the end of the channel wait queue.

IOC$REQSCHANH

Module: IOSUBNPAG

Driver fork processes call IOC$REQSCHANH to request a channel on the secondary controller with high priority. The input to and output from IOC$REQSCHANH are the same as that for IOC$REQPCHANH, except that the secondary controller data channel is assigned.

IOC$REQSCHANL

Module: IOSUBNPAG

Driver fork processes call IOC$REQSCHANL to request a channel on the secondary controller with low priority. The input to and output from IOC$REQSCHANL are the same as that for IOC$REQPCHANH, except that the secondary controller data channel is assigned.

IOC$RETURN

Module: IOSUBNPAG

IOC$RETURN merely returns by issuing an RSB instruction. It has no input requirements and produces no output.

IOC$VERIFYCHAN

Module: IOSUBPAGD

Drivers call IOC$VERIFYCHAN to validate a user-supplied channel number, construct a channel index, and obtain the address of the CCB to which the channel number points. Because IOC$VERIFYCHAN gains access to information stored in user process virtual address space, it should only be called when the user process is mapped.

input

Registers	Contents
R0	Channel number
Fields	**Contents**
CTL$GL_CCBASE	Base address of process CCB table

IPL at execution: IPL$_ASTDEL or below

output

Registers	Contents
R0	SS$_NORMAL, SS$_IVCHAN (invalid channel number), or SS$_NOPRIV (no privilege to access specified channel)
R1	Address of CCB
R2	Channel index number
Fields	**Contents**
—	—

IPL at exit: caller's IPL

IOC$WFIKPCH

Module: IOSUBNPAG

Driver fork processes call IOC$WFIKPCH to suspend driver processing to wait for an interrupt or device timeout while still retaining ownership of the controller data channel.

IOC$WFIKPCH performs the following operations:

- Saves R3, R4, and the driver's return PC from the top of stack in the UCB fork block.
- Sets UCB$V_INT to indicate an expected interrupt from the device unit.
- Sets UCB$V_TIM to indicate that timeouts are expected from the device unit.[3]
- Clears UCB$V_TIMOUT to indicate that the unit has not timed out.
- Lowers IPL to the IPL saved on the top of the stack (generally placed there by an invocation of the DSBINT macro prior to the setting of device registers).
- Returns to the caller of the driver fork process (that is, its caller's caller).

In the course of processing, IOC$WFIKPCH explicitly removes 0(SP) through 11(SP) from the stack and implicitly removes 12(SP) through 15(SP) by exiting with an RSB instruction.

input

Registers	Contents
R5	Address of UCB
0(SP)	Address following the JSB to IOC$WFIKPCH
4(SP)	Timeout value in seconds
8(SP)	IPL to which to lower before returning to the caller's caller
12(SP)	Return address of caller's caller
Fields	**Contents**
EXE$GL_ABSTIM	Absolute time; used to compute time at which the device times out

IPL at execution: Fork or device IPL (caller's IPL)

output

Registers	Contents
—	—

[3] The two bytes following the JSB to IOC$WFIKPCH contain the relative offset to the timeout-handling routine.

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$L_FR4	R4
UCB$L_FPC	0(SP)+2

IPL at exit: IPL specified in 8(SP)

IOC$WFIRLCH

Module: IOSUBNPAG

Driver fork processes call IOC$WFIRLCH to suspend driver processing to wait for an interrupt or device timeout, but first releasing the controller data channel. The input to and output from IOC$WFIRLCH is the same as that for IOC$WFIKPCH, except that IOC$WFIRLCH exits to IOC$RELCHAN, which releases the controller data channel.

MMG$UNLOCK

Module: IOLOCK

Drivers rarely use MMG$UNLOCK. At the completion of a direct-I/O transfer, IOC$IOPOST automatically unlocks the pages of both the user buffer and any additional buffers specified in region 1 (if defined) and region 2 (if defined) for all the IRPEs linked to the packet undergoing completion processing.

However, driver FDT routines do use MMG$UNLOCK when an attempt to lock IRPE buffers for a direct-I/O transfer fails. The buffer-locking routines called by such a driver—EXE$READLOCKR, EXE$WRITELOCKR, and EXE$MODIFYLOCKR—all perform coroutine calls back to the driver if an error occurs. When called as a coroutine, the driver must unlock all previously locked regions using MMG$UNLOCK, and deallocate the IRPE (using EXE$DEANONPAGED), before returning to the buffer-locking routine.

input

Registers	Contents
R1	Number of buffer pages to unlock
R3	System virtual address of PTE for the first buffer page
Fields	**Contents**
—	—

IPL at execution: IPL$_SYNCH

output

Registers	Contents
—	—
Fields	**Contents**
—	—

IPL at exit: caller's IPL

D Device Driver Entry Points

This appendix describes the entry points the VAX/VMS operating system uses to activate a device driver.

D.1 Alternate Start-I/O Routine

The alternate start-I/O routine is an optional entry point present only in drivers that, in some circumstances, initiate multiple, concurrent I/O operations on a device. Drivers that use an alternate start-I/O routine synchronize their access to the UCB and, thus, to the device.

How to Specify This Entry Point

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

Input

Registers	Contents
R3	Address of IRP
R5	Address of UCB

Output

The output of an alternate start-I/O routine is device activity.

Use of Registers

The contents of all registers except R0 through R5 must be preserved.

Context

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

IPL on Entry and Exit

Alternate start-I/O routines are called at fork IPL.

Which VAX/VMS Routines Use This Entry Point

The routine EXE$ALTQUEPKT, in module SYSQIOREQ, calls a driver's alternate start-I/O routine.

D.2 Cancel-I/O Routine

VAX/VMS calls a driver's cancel-I/O routine when the user calls the $CANCEL system service to cancel all requests for I/O activity on a channel. It performs the following functions:

- Determine whether an IRP associated with the cancellation request is actively being processed. It usually does so by first checking the bit UCB$V_BSY in the field UCB$L_STS to see if any request is being processed by the device. If so, the cancel-I/O routine tests whether the PID and channel number of the request being processed match the PID and channel number specified in the cancel-I/O request.
- Cause to be completed (canceled) as quickly as possible all active I/O requests on the specified channel that were made by the process that has requested the cancellation.

  The cancel-I/O routine usually accomplishes this by setting UCB$V_CANCEL in the field UCB$L_STS. When the next interrupt or timeout occurs for the device, the driver's start-I/O routine detects the presence of an active but canceled I/O request by testing this bit and takes appropriate action, such as completing the request without initiating any further device activity.

How to Specify This Entry Point

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

Input

Registers	Contents
R2	Channel index number
R3	Address of IRP
R4	Address of PCB of the process for which the I/O request is being canceled
R5	Address of UCB
R8	Code that stands for the caller of the cancel-I/O routine, one of the following:

Code	Meaning
CAN$C_CANCEL	$CANCEL or $DALLOC system service
CAN$C_DASSGN	$DASSGN system service

Output

The I/O requests on the specified channel are canceled, and the bit UCB$V_CANCEL is set in the field UCB$L_STS.

Use of Registers

The driver's cancel-I/O routine can use R0 through R3 freely. The contents of any other register must be restored before the cancel-I/O routine relinquishes control by means of an RSB instruction.

Context

A cancel-I/O routine executes in kernel mode in process context.

IPL on Entry and Exit

A cancel-I/O routine is called at driver fork IPL.

Which VAX/VMS Routines Use This Entry Point

The $CANCEL, $DASSGN, and $DALLOC system services use this entry point from modules SYSCANCEL, SYSDASSGN, and SYSDEVALC, respectively.

D.3 Controller-Initialization Routine

A controller-initialization routine prepares a controller for operation. Some controllers require initialization when the system's driver-loading routine loads the driver and when the system is recovering from a power failure.

How to Specify This Entry Point

Specify the name of the controller-initialization routine by using the DPT_STORE macro to place the address of the routine in the CRB, into the field CRB$L_INTD+VEC$L_INITIAL.

Input

The caller of the controller-initialization routine provides the following information.

Registers	Contents
R4	Address of device's CSR
R5	Address of IDB associated with the controller
R6	Address of DDB associated with the controller
R8	Address of controller's CRB

The System Generation Utility (SYSGEN) creates all the I/O data structures associated with a device before calling the controller-initialization routine.

Output

Depending on the device, a controller-initialization routine performs any and all of the following actions:

- Clear error-status bits in device registers.
- Enable controller interrupts.
- Store values in fields that are offset more than 256 bytes from the beginning of the data structure and consequently cannot be reached with the DPT_STORE macro.
- Allocate resources that must be permanently allocated to the controller.
- If the controller is dedicated to a single-unit device, such as a printer, fill in IDB$L_OWNER and set the online bit (UCB$V_ONLINE in UCB$L_STS).

Use of Registers

A controller-initialization routine must preserve the contents of all registers except R0, R1, and R2. If the controller-initialization routine uses these registers, it must save their contents first and then restore those contents before returning control to the caller.

Context

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

IPL on Entry and Exit

VAX/VMS calls a controller-initialization routine at IPL$_POWER. The controller-initialization routine must not lower IPL.

Which VAX/VMS Routines Use This Entry Point

SYSGEN calls a driver's controller-initialization routine when processing a CONNECT command. Also, VAX/VMS calls this routine if the device, controller, processor, or adapter to which the device is connected experiences a power failure.

D.4 Driver-Unloading Routine

A driver specifies a driver-unloading routine if there is any device-specific work to do when the driver is unloaded and reloaded.

The driver-unloading routine may perform the following operations:

- Deallocate mapping registers permanently allocated to the device.
- Deallocate a buffered data path permanently allocated to the device.
- Return any allocated system buffers to nonpaged pool.
- Flush the attention AST queue.

How to Specify This Entry Point

Specify the address of the driver-unloading routine in **unload** argument of the DPTAB macro.

Input

Registers	Contents
R6	Address of DDT
R10	Address of DPT

Output

The driver-unloading routine exits with an RSB instruction.

If R0 contains a code that indicates success (the low bit set), the System Generation Utility (SYSGEN) interprets it as meaning it can reload the new version of the driver.

If R0 contains a failure code (low bit cleared), SYSGEN interprets that as meaning it cannot reload the new version of the driver.

Use of Registers

The driver-unloading routine can use any registers.

Context

The driver-unloading routine executes in process context.

IPL on Entry and Exit

SYSGEN calls a driver-unloading routine at IPL$_POWER. The driver-unloading routine must not change IPL.

Which VAX/VMS Routines Use This Entry Point

SYSGEN calls the driver-unloading routine, if it exists, when executing a RELOAD command.

D.5 FDT Routines

FDT routines perform any device-dependent activities needed to prepare the I/O database to process an I/O request. This request may or may not involve the transfer of data.

How to Specify This Entry Point

Use the FUNCTAB macro to specify the set of FDT routines that preprocess requests for I/O activity of a given type. Specify the names of the routines in the order in which you want them to execute for each type of I/O operation.

Input

Registers	Contents
R0	Address of FDT routine being called
R3	Address of IRP
R4	Address of PCB of the requesting process
R5	Address of UCB of the device on which I/O activity is requested
R6	Address of CCB that describes the user-specified process-I/O channel
R7	Number of the bit that specifies the code for the requested I/O function
R8	Address of entry in the function-decision table that dispatched control to this FDT routine
AP	Address of first function-dependent argument (**p1**) specified in the $QIO request

Outputs

No direct outputs are required; but control must either be returned to the $QIO code by means of an RSB instruction, or passed, by means of a JMP instruction, to a routine that queues the IRP or to a routine that finishes or aborts the I/O request.

Use of Registers

FDT routines must preserve the contents of R3 through R8, the AP, and the FP.

Context

FDT routines execute in the context of the process that requested the I/O activity. FDT routines must not lower IPL below IPL$_ASTDEL. If they raise IPL, they must lower it to IPL$_ASTDEL before passing control to any other code. If an FDT routine alters the stack, it must restore the stack before returning control to the caller of the routine.

IPL on Entry and Exit

FDT routines are called at IPL$_ASTDEL and must exit at IPL$_ASTDEL.

Which VAX/VMS Routines Use This Entry Point

The $QIO system service calls an FDT routine from the executive module SYSQIOREQ.

Exiting Mechanisms

The way in which FDT routines exit depends on what I/O activity is requested. The choices are listed below.

For each function a device supports, a set of FDT routines must provide preprocessing of requests for that function. Except for the last FDT routine in such a set, each routine must return control to its caller by means of in RSB instruction. The last must exit by means of one of the routines listed below, not by means of an RSB instruction.

Exit Mechanism	Function
EXE$ABORTIO	Aborts an I/O request and returns to the caller of the $QIO system service, as status information, the contents of R0
EXE$ALTQUEPKT	Queues an IRP to the driver's alternate entry point without checking the status of the device
EXE$FINISHIO	Finishes the I/O processing, returning a quadword of status information to the caller of the $QIO system service. (EXE$FINISHIO takes the status information from R0 and R1 and returns it in the IOSB specified in the call to $QIO.)
EXE$FINISHIOC	Finishes the I/O processing, returning a longword of status information to the caller of the $QIO system service. (EXE$FINISHIOC takes the status information from R0 and returns it in the IOSB specified in the call to $QIO, clearing the second longword of the IOSB.)
EXE$QIODRVPKT	Queues an IRP to the pending I/O queue if the device is busy, or starts I/O activity if the device is idle
RSB	Returns control to the caller of the routine, that being the FDT-processing loop of the $QIO system service

D.6 Interrupt-Servicing Routine

An interrupt-servicing routine processes interrupts generated by the device. The interrupts can signal the completion of an I/O operation or an error. UNIBUS and Q22 bus devices require an interrupt-servicing routine for each UNIBUS or Q22 bus interrupt vector the device has.

Tape devices on the MASSBUS require an interrupt-servicing routine that interrogates the tape formatter (the controller) to determine which drive needs attention and if the interrupt is unsolicited.

Disk devices on the MASSBUS use an interrupt-servicing routine provided by VAX/VMS and do not need to provide an interrupt-servicing routine.

An interrupt-servicing routine performs the following functions:

1 Determine whether the interrupt is expected

2 Process or dismiss unexpected interrupts

3 Activate the suspended driver so it can process expected interrupts

For MASSBUS devices, the interrupt-servicing routine supplied with VAX/VMS provides these functions.

How to Specify This Entry Point

Use the DPT_STORE macro to place the address of the interrupt-servicing routine into the field CRB$L_INTD+4.

If the device has two different interrupts, use the DPT_STORE macro to specify the name of the second interrupt-servicing routine and to place the address of that routine into the longword field CRB$L_INTD2+4 within the CRB.

Input

When VAX/VMS invokes a driver's interrupt-servicing routine, the stack contains the following data:

Stack Location	Contents
0(SP)	Address of longword that contains the address of the IDB
4(SP) to 24(SP)	*For UNIBUS and Q22 bus devices*, the contents of R0 through R5 at the time of the interrupt; *for MASSBUS devices*, the contents of R2 through R5 at the time of the interrupt
28(SP)	PC at the time of the interrupt
32(SP)	PSL at the time of the interrupt

Output

Before an interrupt-servicing routine transfers control to the suspended driver, it must restore the contents of R3 and R4 from the UCB. It then transfers control to the address saved in UCB$L_FPC.

When it regains control (after the suspended driver forks), an interrupt-servicing routine removes the address of the pointer to the IDB from the top of the stack and restores the registers VAX/VMS saved when dispatching the interrupt (R0 through R5 for UNIBUS and Q22 bus interrupt-servicing

routines, R2 through R5 for MASSBUS interrupt-servicing routines). Finally, an interrupt-servicing routine dismisses the interrupt with an REI instruction.

Use of Registers

If an interrupt-servicing routine user R6 through R11, the AP, or the FP, it must first save the contents of those registers, restoring their contents before exiting by means of the REI instruction. MASSBUS drivers must also preserve the contents of R0 and R1.

Context

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

IPL on Entry and Exit

The interrupt-servicing routine is called, executes, and returns at device IPL.

Which VAX/VMS Routines Use This Entry Point

The interrupt-servicing routine is called by the VAX/VMS interrupt-servicing routines, the addresses of which are usually loaded into the ADP, the CRB, or both for the interrupting device.

D.7 Register-Dumping Routine

The VAX/VMS error-logging and diagnostic-buffer-filling routines call the register-dumping routine to copy the contents of a device's registers into an error-log entry or the diagnostic buffer.

How to Specify This Entry Point

Specify the name of the register-dumping routine in the **regdmp** argument of the DDTAB macro. This macro places the address of the routine into the DDT.

Input

The register-dumping routine has the following inputs.

Registers	Contents
R0	Address of buffer into which a register-dumping routine copies the contents of device registers
R4	Address of device's CSR
R5	Address of UCB

Output

The contents of the device's registers are copied into the buffer.

Use of Registers

The register-dumping routine preserves the contents of all registers except R0 through R2. If it uses the stack, the register-dumping routine must restore the stack before passing control to another routine, waiting for an interrupt, or returning control to its caller.

Context

A register-dumping routine executes within the context of an interrupt-servicing routine or a fork process, using the kernel-mode stack. As a result, it can refer only to those virtual addresses that reside in system (S0) space.

IPL on Entry and Exit

VAX/VMS calls a register-dumping routine at the same IPL at which the driver called the VAX/VMS routine ERL$DEVICERR, ERL$DEVICTMO, or IOC$DIAGBUFILL. A register-dumping routine must not change IPL.

Which VAX/VMS Routines Use This Entry Point

The routines ERL$DEVICERR and ERL$DEVICTMO in module ERRORLOG, and IOC$DIAGBUFILL in module IOSUBNPAG call the register-dumping routine.

D.8 Start-I/O Routine

The VAX/VMS routines IOC$REQCOM and IOC$INITIATE call a driver's start-I/O routine. The start-I/O routine activates a device.

How to Specify This Entry Point

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

Input

Registers	Contents
R3	Address of IRP
R5	Address of UCB

VAX/VMS copies the following information from the current IRP into the UCB fields listed below.

Fields	Contents
UCB$W_BCNT	Number of bytes to be transferred, copied from the low-order word of IRP$L_BCNT
UCB$W_BOFF	Offset from the beginning of the page of the first byte to be transferred, copied from IRP$W_BOFF
UCB$L_SVAPTE	System virtual address of first PTE that describes the buffer, copied from IRP$L_SVAPTE

Output

The output of a start-I/O routine is device activity.

Use of Registers

The contents of all registers except R0, R1, R2, and R4 must be preserved.

If the start-I/O routine uses the stack, it must restore the stack before completing the request, waiting for an interrupt, or requesting system resources.

Context

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

IPL on Entry and Exit

A start-I/O routine gains control of the processor, and relinquishes control, at fork IPL. For many devices, the start-I/O routine raises IPL to IPL$_POWER to check that a power failure has not occurred on the device. The start-I/O routine initiates device activity at device IPL.

Which VAX/VMS Routines Use This Entry Point

The start-I/O routine is called by IOC$INITIATE and IOC$REQCOM in module IOSUBNPAG.

D.9 Timeout-Handling Routine

A timeout-handling routine takes whatever action is necessary when a device has not yet responded to a request for device activity and the time allowed for a response has expired.

How to Specify This Entry Point

Specify the name of the timeout-handling routine in the **excpt** argument to the WFIKPCH or the WFIRLCH macro.

Input

Registers	Contents
R3	Contents of R3 when the last invocation of WFIKPCH or WFIRLCH took place
R4	Contents of R4 when the last invocation of WFIKPCH or WFIRLCH took place
R5	Address of UCB of the device

Output

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

Before calling a timeout-handling routine, VAX/VMS places the device in a state in which no interrupt is expected (by clearing the bit UCB$V_INT in field UCB$L_STS). If the requested interrupt occurs after this routine is called, it will appear to be an unsolicited interrupt. Many drivers handle this situation by disabling interrupts while the timeout-handling routine executes.

Use of Registers

A timeout-handling routine can use R0, R1, and R2 freely, but must preserve the contents of all other registers.

If a timeout-handling routine uses the stack, it must restore the stack before completing or canceling the current I/O request, waiting for an interrupt, or returning control to its caller.

Context

Because a timeout-handling routine executes in the context of a fork process, it can access only those virtual addresses that refer to system (S0) space.

IPL on Entry and Exit

A timeout-handling routine is called at device IPL. After taking whatever device-specific action is necessary at device IPL, a timeout-handling routine can lower IPL to fork IPL.

Which VAX/VMS Routines Use This Entry Point

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

Routines in the VAX/VMS module TIMESCHDL call the timeout-handling routine at the request of the WFIKPCH and WFIRLCH macros.

D.10 Unit-Delivery Routine

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

The System Generation Utility (SYSGEN) calls the unit-delivery routine once for each unit the controller is capable of controlling. This value is specified in the **defunits** argument to the DPTAB macro.

How to Specify This Entry Point

Specify the name of the unit-delivery routine in the **deliver** argument to the DPTAB macro.

Input

Registers	Contents
R3	Address of IDB; 0 if none exists
R4	Address of device's CSR
R5	Number of unit that the unit-delivery routine must decide to configure or not to configure
R6	Address of start of the UNIBUS adapter's I/O space
R7	Address of AUTOCONFIGURE command's configuration-control block (ACF)
R8	Address of ADP

Output

If bit 0 is set in R0, the unit should be configured; if it is cleared, the unit should not be configured.

Use of Registers

The unit-delivery routine can use R0, R1, and R2 freely, but must preserve the contents of all other registers.

Context

The unit-delivery routine executes in the context of the process within which SYSGEN executes.

IPL on Entry and Exit

The unit-delivery routine is called at IPL$_POWER, and must not lower IPL.

Which VAX/VMS Routines Use This Entry Point

SYSGEN's AUTOCONFIGURE command calls the unit-delivery routine.

D.11 Unit-Initialization Routine

A unit-initialization routine prepares a device for operation. In the case of a device on a dedicated controller, the unit-initialization routine also initializes the controller. The unit-initialization routine is called when the driver-loading routine loads the driver and when the system is recovering from a power failure.

How to Specify This Entry Point

You can specify a unit-initialization routine in two ways, either of which will suffice for all but a few specific devices.[2]

- Use the DDTAB macro to specify the unit-initialization routine by providing the name of the routine in the **unitinit** argument.
- Use the DPT_STORE macro to place the address of the unit-initialization routine in the CRB, into field CRB$L_INTD+VEC$L_UNITINIT.

Input

The caller of unit-initialization routines provides the following information.

Registers	Contents
R3	Address of primary CSR
R4	Address of secondary CSR, if it exists. (If it does not, the contents of R4 are the same as those of R3.)
R5	Address of UCB

In addition, the System Generation Utility (SYSGEN) creates the I/O data structures associated with a device before calling the unit-initialization routine.

[2] A MASSBUS device driver must specify the address of its unit-initialization routine in the DDT (using the **unitinit** argument to the DDTAB macro as discussed in Section 7.2). UNIBUS and Q22 bus drivers can specify the address in either the DPT or DDT.

Output

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

1. Clear error-status bits in device registers.
2. Enable controller interrupts.
3. Set the online bit (UCB$V_ONLINE in UCB$L_STS).
4. Store values in fields that are offset more than 256 bytes from the beginning of the UCB and, consequently, cannot be reached with the DPT_STORE macro.
5. Allocate resources that must be permanently allocated to the device or, for some devices, the controller.
6. If the device has a dedicated controller, as some printers do, fill in IDB$L_OWNER.

Use of Registers

A unit-initialization routine must preserve the contents of all registers except R0, R1, and R2. If the unit-initialization routine uses these registers, it must save their contents first and then restore those contents before returning control to the caller.

Context

Because VAX/VMS calls it in system context, a unit-initialization routine can refer to only those virtual addresses that reside in system (S0) space.

IPL on Entry and Exit

VAX/VMS calls a unit-initialization routine at IPL$_POWER. A unit-initialization routine must not lower IPL.

Which VAX/VMS Routines Use This Entry Point

SYSGEN calls a unit-initialization routine when processing a CONNECT command. VAX/VMS calls a unit-initialization routine when the device, the controller, the processor, or the adapter to which the device is connected undergoes power failure recovery.

D.12 Unsolicited-Interrupt-Servicing Routine

For MASSBUS disks, VAX/VMS calls the unsolicited-interrupt-servicing routine whenever a hardware event produces an interrupt that is not the result of a driver's request. Examples of such events are disks being placed on line or taken off line.

Only drivers of MASSBUS disks must provide unsolicited-interrupt-servicing routines. All other devices detect unsolicited interrupts in their interrupt-servicing routines.

The routine that handles these unsolicited interrupts must determine the nature of the interrupt and act accordingly, depending on the characteristics of the device and controller.

How to Specify This Entry Point

Provide the name of the unsolicited-interrupt routine in the **unsolic** argument to the DDTAB macro. This macro places the address of the routine into the DDT.

Input

Registers	Contents
R4	Address of MBA's CSR
R5	Address of UCB

Output

There are no required outputs.

Use of Registers

The unsolicited-interrupt-servicing routine must not alter the contents of registers R6 through R11 or the AP or FP.

Context

Because, the unsolicited interrupt-servicing routine executes in kernel mode on the interrupt stack, it can refer to only those addresses that reside in system (S0) space.

IPL on Entry and Exit

An unsolicited interrupt-servicing routine is called, executes, and returns at device IPL.

Which VAX/VMS Routines Use This Entry Point

The MBA$INT routine in module MBAINTDSP of the SYSLOA facility calls an unsolicited interrupt-servicing routine.

E Sample Driver for the RL11, RL01, and RL02

This example driver, DLDRIVER, drives devices on both the UNIBUS and the MicroVAX I and MicroVAX II Q22 bus. Specific code changes since VAX/VMS Version 4.0 are highlighted with change bars in the margin.

```
        .TITLE  DLDRIVER - VAX/VMS RL11/RL01,RL02 DISK DRIVER
        .IDENT  'V03-008'
;
;*****************************************************************************
;*                                                                           *
;*  COPYRIGHT (c) 1978, 1980, 1982, 1984, 1986 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 RL11/RL01,RL02 DISK DRIVER
;
; AUTHOR:
;
; C. F. Programmer     05-OCT-1979
;
; MODIFIED BY:
;
;       V04-002 RLRCPUDISPb     R. L. Programmer        22-Mar-1985
;               Modify CPUDISP invocations to use CONTINUE=YES and
;               thereby obviate the need to necessarily modify this
;               driver each time a new CPU comes along.
;
;       V04-001 JJ00003         J. J. Programmer        31-Jan-1985
;               Added MicroVAX II support.
;
;       V03-008 WHM0001         B. M. Programmer        15-May-1984
;               Added MicroVAX I/QBUS support.
```

```
;
;       V03-007 RAS0300         R. S. Programmer        27-Apr-1984
;               Add DEV$M_NNM characteristic to DECHAR2 so that these
;               devices will have the "node$" prefix.
;
;       V03-006 PRD0033         P. R. Programmer        09-Sep-1983
;               Added EXE$LCLDSKVALID to function decision table.
;
;       V03-005 ROW0211         R. O. Programmer        16-AUG-1983
;               Change device-dependent UCB definition base from UCB$W_BCR+2
;               to UCB$K_LCL_DISK_LENGTH.
;
;       V03-004 KDM0059         K. D. Programmer        14-Jul-1983
;               Change time-wait loops to use new TIMEDWAIT macro.
;
;       V03-003 PRD0020         P. R. Programmer        26-Apr-1983
;               Modified FATALERR routine to return SS$_PARITY only for
;               errors that possibly indicate bad media.  All other error
;               conditions which formerly returned SS$_PARITY now return
;               SS$_CNTLERR.
;
;       V03-002 KDM0002         K. D. Programmer        28-Jun-1982
;               Added $DYNDEF.
;
;       V03-001 KTA0100         K. T. Programmer        07-Jun-1982
;               Add code to set UCB$L_MEDIA_ID.
;
;**
.PAGE
; ABSTRACT:
;
;       THIS MODULE CONTAINS THE TABLES AND ROUTINES NECESSARY TO
;       PERFORM ALL DEVICE-DEPENDENT PROCESSING OF AN I/O REQUEST
;       FOR RL11/RL01,RL02 DISK TYPES ON A VAX/VMS SYSTEM.
;
;       THE DISKS HAVE THE FOLLOWING PHYSICAL GEOMETRY:
;
;                       TRACKS/     SECTORS/      BYTES/      MAXIMUM
;          # CYL         CYLINDER    TRACK         SECTOR      BLOCKS
;
; RL01       256            2          40            256       10240
; RL02       512            2          40            256       20480
;
```

```
;       SINCE THE SECTOR SIZE IS ONLY 1/2 BLOCK, LOGICAL TO PHYSICAL
;       CONVERSION OF THE DISK ADDRESS IS DONE IN THE DRIVER STARTIO
;       ROUTINE RATHER THAN IN THE IOC$CVTLOGPHY FDT ROUTINE.
;
;       OVERLAPPED SEEKS ARE NOT ATTEMPTED BECAUSE THE DEVICE DOES
;       NOT INTERRUPT AT THE COMPLETION OF A SEEK.
;
;       ALSO, THE DEVICE DOES NOT PERFORM AN IMPLICIT SEEK WHEN PERFORMING
;       A READ OR WRITE FUNCTION, SO SEEK FUNCTIONS ARE ISSUED BY THIS
;       DRIVER WHERE NECESSARY PRIOR TO ISSUING A READ OR WRITE FUNCTION.
;       THE READ OR WRITE FUNCTION IS THEN ISSUED AS SOON AS THE RL11
;       CONTROLLER COMES READY (WHILE THE SEEK IS IN PROGRESS), AND A
;       WAIT-FOR-INTERRUPT (UPON COMPLETION OF THE READ OR WRITE) IS
;       ISSUED.  IF A SEEK FUNCTION IS REQUESTED SEPARATELY FROM A READ OR
;       WRITE, A DUMMY READ HEADER FUNCTION IS ISSUED FOLLOWING THE SEEK
;       FUNCTION AND A WAIT-FOR-INTERRUPT (UPON COMPLETION OF THE READ
;       HEADER) IS ISSUED.
;
;       THE IO$X_INHSEEK FUNCTION MODIFIER IS TREATED AS A NO-OP BY
;       THIS DRIVER, SINCE AN EXPLICIT SEEK IS NECESSARY FOR THE RL02
;       TO TRANSFER DATA PROPERLY.
;
;       THE RL'S DO NOT READ OR WRITE BEYOND THE END OF TRACK (THEY DO NOT
;       AUTOMATICALLY SEEK THE NEXT TRACK), SO  ALL READ AND WRITE FUNCTIONS
;       ARE BROKEN UP BY THIS DRIVER INTO PARTIAL TRANSFERS TO THE END OF
;       TRACK, FOLLOWED BY A SEEK TO THE NEXT TRACK, THEN ANOTHER READ OR
;       WRITE FUNCTION UNTIL THE TOTAL DATA TRANSFER IS COMPLETE.
;
;--
        .PAGE
        .SBTTL  EXTERNAL AND LOCAL DEFINITIONS
;
; EXTERNAL SYMBOLS
;
        $ADPDEF                         ;DEFINE ADAPTER CONTROL BLOCK
        $CRBDEF                         ;DEFINE CHANNEL REQUEST BLOCK
        $DCDEF                          ;DEFINE DEVICE CLASS
        $DDBDEF                         ;DEFINE DEVICE DATA BLOCK
        $DEVDEF                         ;DEFINE DEVICE CHARACTERISTICS
        $DPTDEF                         ;DEFINE DRIVER PROLOGUE TABLE
        $DYNDEF                         ;DEFINE DYNAMIC DATA STRUCTURE TYPES
        $EMBDEF                         ;DEFINE ERROR MESSAGE BUFFER
        $IDBDEF                         ;DEFINE INTERRUPT DISPATCH BLOCK
        $IODEF                          ;DEFINE I/O FUNCTION CODES
        $IRPDEF                         ;DEFINE I/O REQUEST PACKET
        $PRDEF                          ;DEFINE PROCESSOR REGISTERS
        $PTEDEF                         ;DEFINE SYSTEM PTES
        $SSDEF                          ;DEFINE SYSTEM STATUS CODES
        $UCBDEF                         ;DEFINE UNIT-CONTROL BLOCK
        $VADEF                          ;DEFINE VIRTUAL ADDRESS BITS
        $VECDEF                         ;DEFINE INTERRUPT VECTOR BLOCK
;
; LOCAL MACROS
;
;       EXFUNCL
;       BRANCH TO SUBROUTINE WHICH REQUESTS CHANNEL (IF NOT ALREADY OWNED),
;       EXECUTES FCODE (OR R3) FUNCTION, AND BRANCHES TO BDST ON ERROR
        .MACRO  EXFUNCL BDST,FCODE
                .IF NB  FCODE           ;IS FCODE NON-BLANK?
                MOVZBL  #CD'FCODE,R3    ;IF NB, SPECIFY FCODE FUNCTION
                .ENDC                   ;IF B, SPECIFY FNTN IN EXISTING R3
                BSBW    FEXL            ;EXECUTE FUNCTION
                .BYTE   BDST-.-1        ;WHERE TO GO ON ERROR
        .ENDM
```

```
;       GENF
;       GENERATE FUNCTION TABLE ENTRY AND CASE TABLE INDEX SYMBOL
        .MACRO  GENF FCODE
                CD'FCODE=.-FTAB/2
                .WORD   FCODE!RL_CS_M_IE  ;FCODE WITH INT ENABLE BIT
        .ENDM

;       CKPWR
;       DISABLE INTERRUPTS, CHECK IF POWER HAS FAILED,
;       AND PUT DEVICE UNIT NUMBER IN R2<9:8>
;
        .MACRO  CKPWR ?L1
                CLRL    R2              ;CLEAR R2 FOR UNIT NUMBER
                INSV    UCB$W_UNIT(R5),-  ;PUT UNIT # IN R2<9:8>
                        #8,#2,R2        ;...
                DSBINT                  ;DISABLE INTERRUPTS
                BBC     #UCB$V_POWER,-  ;IF CLR, NO POWER FAILURE
                        UCB$W_STS(R5),L1  ;...
                ENBINT                  ;POWER FAILURE - ENABLE INTERRUPTS
                BRW     RETREG          ;EXIT
L1:                                     ;RETURN FOR NO POWER FAILURE
        .ENDM

;
; LOCAL SYMBOLS
;
RL_NUM_REGS     =4                      ;NUMBER OF DEVICE REGISTERS
RL_SLM          =5                      ;STATE=SEEK LINEAR MODE (READY TO GO)
UCB$B_DL_DCHEK  =UCB$W_OFFSET+1         ;REDEFINE FOR DATA CHECK USE

;
; UCB OFFSETS WHICH FOLLOW THE STANDARD UCB FIELDS
;
        $DEFINI UCB                     ;START OF UCB DEFINITIONS
.=UCB$K_LCL_DISK_LENGTH                 ;BEGIN DEFINITIONS AT END OF UCB
$DEF    UCB$W_DL_PBCR   .BLKW   1       ;PARTIAL BYTE COUNT
$DEF    UCB$W_DL_CS     .BLKW   1       ;CONTROL STATUS REGISTER
$DEF    UCB$W_DL_BA     .BLKW   1       ;BUS ADDRESS REGISTER
$DEF    UCB$W_DL_DA     .BLKW   1       ;DISK ADDRESS REGISTER
$DEF    UCB$W_DL_MP     .BLKW   1       ;MULTIPURPOSE REGISTER
$DEF    UCB$W_DL_DPN    .BLKW   1       ;DATA-PATH NUMBER
$DEF    UCB$L_DL_SVAPTE                 ;SAVED SVAPTE OF THE USER'S BUFFER
$DEF    UCB$L_DL_DPR    .BLKL   1       ;DATA-PATH REGISTER
$DEF    UCB$L_DL_BUFADR                 ;USER BUFFER ADDRESS
$DEF    UCB$L_DL_FMPR   .BLKL   1       ;FINAL MAP REGISTER
$DEF    UCB$A_DL_MOVRTN                 ;BUFFER MOVE ROUTINE ADDRESS
$DEF    UCB$L_DL_PMPR   .BLKL   1       ;PREVIOUS MAP REGISTER
$DEF    UCB$B_DL_DPPE   .BLKB   1       ;DATA-PATH PURGE ERROR
$DEF    UCB$W_DL_DB     .BLKW   3       ;DATA BUFFER REGISTER
$DEF    UCB$B_DL_XBA    .BLKB   1       ;BUS ADDRESS EXTENSION BITS
$DEF    UCB$W_DL_SBA    .BLKW   1       ;SAVED BUFFER ADDRESS
$DEF    UCB$A_DL_BUF_VA .BLKL   1       ;PHYSICAL BUFFER VIRTUAL ADDRESS
$DEF    UCB$A_DL_BUF_PA .BLKL   1       ;PHYSICAL BUFFER PHYSICAL ADDRESS
$DEF    UCB$W_DL_FLAGS  .BLKW   1       ;FLAGS
        $VIELD  UCB,0,<-                ;START THE FLAG DEFINITIONS
                <DL_22BIT,,M>,-         ;22 BIT ADDRESSING
                <DL_MAPPING,,M>,-       ;ADAPTER MAPPING
                >                       ;END OF FLAG DEFINITIONS
$DEF    UCB$K_DL_LEN    .BLKW   1       ;LENGTH OF UCB
$EQU    UCB$K_DL_BUFSZ 20               ;BUFFER SIZE = 40 SECTORS *
                                        ;256 BYTES/SECTOR / 512 BYTES/PAGE
        $DEFEND UCB                     ;END OF UCB DEFINITONS
```

```
;
; RL11/RL01 REGISTER OFFSETS FROM CSR ADDRESS
;
        $DEFINI RL                      ; START OF REGISTER DEFINITIONS
$DEF    RL_CS           .BLKW   1       ;CONTROL STATUS REGISTER (CSR)
        _VIELD  RL_CS,0,<-              ;START OF CSR BIT DEFINITIONS
                <DRDY,,M>,-             ; DRIVE READY
                <FCODE,3>,-             ; FUNCTION CODE
                <XBA,2>,-               ; BUS ADDRESS EXTENSION BITS
                <IE,,M>,-               ; INTERRUPT ENABLE
                <CRDY,,M>,-             ; CONTROLLER READY
                <DS,2>,-                ; DRIVE SELECT
                <OPI,,M>,-              ; OPERATION INCOMPLETE
                <CRC,,M>,-              ; DATA CRC OR HEADER CRC
                <DLT,,M>,-              ; DATA LATE OR HEADER NOT FOUND
                <NXM,,M>,-              ; NON-EXISTENT MEMORY
                <DE,,M>,-               ; DRIVE ERROR
                <CE,,M>-                ; COMPOSITE ERROR
        >                               ;END CSR BIT DEFINITIONS
$DEF    RL_BA           .BLKW   1       ;BUS ADDRESS REGISTER (BAR)
$DEF    RL_DA           .BLKW   1       ;DISK ADDRESS REGISTER (DAR)
        _VIELD  RL_DA,0,<-              ;START OF DAR BIT DEFINITIONS
                <MRK,,M>,-              ; MARK (ALWAYS 1)
                <STS,,M>,-              ; GET STATUS
                <,1>,-                  ; RESERVED BIT
                <RST,,M>,-              ; RESET
                <,12>,-                 ; RESERVED BITS
        >                               ;END OF DAR BIT DEFINITIONS
$DEF    RL_MP           .BLKW   1       ;MULTIPURPOSE REGISTER (MPR)
        _VIELD  RL_MP,0,<-              ;START OF MPR BIT DEFINITIONS
                <STA,3>,-               ; DRIVE STATE
                <BH,,M>,-               ; BRUSH HOME
                <HO,,M>,-               ; HEADS OUT
                <CO,,M>,-               ; COVER OPEN
                <HS,,M>,-               ; HEAD SELECT
                <TYP,,M>,-              ; DRIVE TYPE
                <DSE,,M>,-              ; DRIVE SELECT ERROR
                <VC,,M>,-               ; VOLUME CHECK
                <WGE,,M>,-              ; WRITE GATE ERROR
                <SPE,,M>,-              ; SPIN ERROR
                <SKTO,,M>,-             ; SEEK TIME OUT
                <WL,,M>,-               ; WRITE LOCK
                <CHE,,M>,-              ; CURRENT HEAD ERROR
                <WDE,,M>-               ; WRITE DATA ERROR
        >                               ;END MPR BIT DEFINITIONS
$DEF    RL_BAE          .BLKW   1       ; BUS ADDRESS EXTENSION REGISTER(BAE)
        $DEFEND RL                      ;END RL11/RL01 REGISTER DEFINITIONS
;
; HARDWARE FUNCTION CODES
;
F_NOP=0*2                               ;NO OPERATION
F_UNLOAD=F_NOP                          ;NO OPERATION
F_SEEK=3*2                              ;SEEK CYLINDER
F_RECAL=F_NOP                           ;NO OPERATION
F_DRVCLR=2*2                            ;DRIVE CLEAR (GET STATUS)
F_RELEASE=F_NOP                         ;NO OPERATION
F_OFFSET=F_NOP                          ;NO OPERATION
F_RETCENTER=F_NOP                       ;NO OPERATION
F_PACKACK=2*2                           ;PACK ACKNOWLEDGE (SET VOLUME VALID)
F_SEARCH=F_NOP                          ;NO OPERATION
F_WRITECHECK=1*2                        ;WRITE CHECK
F_WRITEDATA=5*2                         ;WRITE DATA
F_WRITEHEAD=F_NOP                       ;NO OPERATION
F_READDATA=6*2                          ;READ DATA
F_READHEAD=4*2                          ;READ HEADER
F_AVAILABLE=F_NOP                       ;NO OPERATION
F_GETSTATUS=2*2                         ;GET STATUS (DRIVER INTERNAL USE)
```

```
        .PAGE
        .SBTTL  STANDARD TABLES
;
; DRIVER PROLOGUE TABLE
;
;       THE DPT DESCRIBES DRIVER PARAMETERS AND I/O DATABASE FIELDS
;       THAT ARE TO BE INITIALIZED DURING DRIVER LOADING AND RELOADING
;
        DPTAB   -                       ;DPT CREATION MACRO
                END=DL_END,-            ;END OF DRIVER LABEL
                ADAPTER=UBA,-           ;ADAPTER TYPE = UNIBUS
                FLAGS=DPT$M_SVP,-       ;SYSTEM PAGE TABLE ENTRY REQUIRED
                UCBSIZE=UCB$K_DL_LEN,-  ;LENGTH OF UCB
                NAME=DLDRIVER           ;DRIVER NAME
        DPT_STORE INIT                  ;START CONTROL BLOCK INIT VALUES
        DPT_STORE DDB,DDB$L_ACPD,L,<^A\F11\>  ;DEFAULT ACP NAME
        DPT_STORE DDB,DDB$L_ACPD+3,B,DDB$K_CART ;ACP CLASS
        DPT_STORE UCB,UCB$B_FIPL,B,8    ;FORK IPL
        DPT_STORE UCB,UCB$L_DEVCHAR,L,- ;DEVICE CHARACTERISTICS
                <DEV$M_FOD-             ; FILES ORIENTED
                !DEV$M_DIR-             ; DIRECTORY STRUCTURED
                !DEV$M_AVL-             ; AVAILABLE
                !DEV$M_ELG-             ; ERROR LOGGING
                !DEV$M_SHR-             ; SHAREABLE
                !DEV$M_IDV-             ; INPUT DEVICE
                !DEV$M_ODV-             ; OUTPUT DEVICE
                !DEV$M_RND>             ; RANDOM ACCESS
        DPT_STORE UCB,UCB$L_DEVCHAR2,L,-; DEVICE CHARACTERISTICS
                <DEV$M_NNM>             ; PREFIX NAME WITH "node$"
        DPT_STORE UCB,UCB$B_DEVCLASS,B,DC$_DISK  ;DEVICE CLASS
        DPT_STORE UCB,UCB$W_DEVBUFSIZ,W,512  ;DEFAULT BUFFER SIZE
        DPT_STORE UCB,UCB$B_SECTORS,B,40  ;NUMBER OF SECTORS PER TRACK
        DPT_STORE UCB,UCB$B_TRACKS,B,2  ;NUMBER OF TRACKS PER CYLINDER
        DPT_STORE UCB,UCB$B_DIPL,B,21   ;DEVICE IPL
        DPT_STORE UCB,UCB$B_ERTMAX,B,8  ;MAX ERROR RETRY COUNT
        DPT_STORE UCB,UCB$W_DEVSTS,W,-  ;INHIBIT LOG TO PHYS CONVERSION IN FDT
                <UCB$M_NOCNVRT>         ;...
        DPT_STORE REINIT                ;START CONTROL BLOCK RE-INIT VALUES
        DPT_STORE CRB,CRB$L_INTD+4,D,DL_INT  ;INTERRUPT SERVICE ROUTINE ADDRESS
        DPT_STORE CRB,CRB$L_INTD+VEC$L_INITIAL,-  ;CONTROLLER INIT ADDRESS
                  D,DL_RL11_INIT        ;...
        DPT_STORE CRB,CRB$L_INTD+VEC$L_UNITINIT,- ;UNIT INIT ADDRESS
                  D,DL_RLOX_INIT        ;...
        DPT_STORE DDB,DDB$L_DDT,D,DL$DDT        ;DDT ADDRESS
        DPT_STORE END                   ;END OF INITIALIZATION TABLE
;
; DRIVER DISPATCH TABLE
;
;       THE DDT LISTS ENTRY POINTS FOR DRIVER SUBROUTINES WHICH ARE
;       CALLED BY THE OPERATING SYSTEM.
;
        DDTAB   -                       ;DDT CREATION MACRO
                DEVNAM=DL,-             ;NAME OF DEVICE
                START=DL_STARTIO,-      ;START I/O ROUTINE
                UNSOLIC=DL_UNSOLNT,-    ;UNSOLICITED INTERRUPT
                FUNCTB=DL_FUNCTABLE,-   ;FUNCTION DECISION TABLE
                CANCEL=0,-              ;CANCEL=NO-OP FOR FILES DEVICE
                REGDMP=DL_REGDUMP,-     ;REGISTER DUMP ROUTINE
                DIAGBF=<<RL_NUM_REGS+5+5+3+1>*4>,-  ;BYTES IN DIAG BUFFER
                ERLGBF=<<<RL_NUM_REGS+5+1>*4>+EMB$L_DV_REGSAV>  ;BYTES IN
                                                ;ERROR LOG BUFFER
```

```
; DIAGNOSTIC BUFFER SIZE = <<4 RLO2 REGISTER LONGWORDS + 5 UCB FIELD LONGWORDS
;                        + 5 IOC$DIAGBUFILL LONGWORDS + 3 BUFFER ALLOCATION
;                        LONGWORDS + 1 LONGWORD FOR # REGISTERS IN DL_REGDUMP>
;                        * 4 BYTES/LONGWORD>
;
; ERROR LOG BUFFER SIZE  = <<<4 RLO2 REGISTER LONGWORDS + 5 UCB FIELD LONGWORDS
;                        + 1 LONGWORD FOR # REGISTERS IN DL_REGDUMP>
;                        * 4 BYTES/LONGWORD> + BYTES NEEDED FOR ERROR LOGGER
;                        TO SAVE SOFTWARE REGISTERS>

;
; HARDWARE FUNCTION CODE TABLE
;
;       THIS TABLE MERGES THE FUNCTION CODE BITS WITH THE
;       INTERRUPT ENABLE BIT AND GENERATES THE CASE TABLE
;       INDEX SYMBOL.
FTAB:   GENF    F_NOP                   ;NO-OP
        GENF    F_UNLOAD                ;UNLOAD VOLUME (NOP)
        GENF    F_SEEK                  ;SEEK
        GENF    F_RECAL                 ;RECALIBRATE (NOP)
        GENF    F_DRVCLR                ;DRIVE CLEAR (RESET & GET STATUS)
        GENF    F_RELEASE               ;RELEASE PORT (NOP)
        GENF    F_OFFSET                ;OFFSET HEADS (NOP)
        GENF    F_RETCENTER             ;RETURN HEADS TO CENTERLINE (NOP)
        GENF    F_PACKACK               ;PACK ACKNOWLEDGE (RESET & GET STATUS)
        GENF    F_SEARCH                ;SEARCH (NOP)
        GENF    F_WRITECHECK            ;WRITE CHECK
        GENF    F_WRITEDATA             ;WRITE DATA
        GENF    F_READDATA              ;READ DATA
        GENF    F_WRITEHEAD             ;WRITE HEADERS (NOP)
        GENF    F_READHEAD              ;READ HEADERS
        GENF    F_NOP                   ;PLACE HOLDER
        GENF    F_NOP                   ;PLACE HOLDER
        GENF    F_AVAILABLE             ;AVAILABLE

        .PAGE
;
; FUNCTION DECISION TABLE
;
;       THE FDT LISTS VALID FUNCTION CODES, SPECIFIES WHICH
;       CODES ARE BUFFERED, AND DESIGNATES SUBROUTINES TO
;       PERFORM PREPROCESSING FOR PARTICULAR FUNCTIONS.
;
DL_FUNCTABLE:
        FUNCTAB ,-                      ;LIST LEGAL FUNCTIONS
                <NOP,-                  ; NO-OP
                UNLOAD,-                ; UNLOAD
                SEEK,-                  ; SEEK
                DRVCLR,-                ; DRIVE CLEAR
                PACKACK,-               ; PACK ACKNOWLEDGE
                SENSECHAR,-             ; SENSE CHARACTERISTICS
                SETCHAR,-               ; SET CHARACTERISTICS
                SENSEMODE,-             ; SENSE MODE
                SETMODE,-               ; SET MODE
                WRITECHECK,-            ; WRITE CHECK
                READHEAD,-              ; READ HEADER
                READLBLK,-              ; READ LOGICAL BLOCK
                WRITELBLK,-             ; WRITE LOGICAL BLOCK
                READPBLK,-              ; READ PHYSICAL BLOCK
                WRITEPBLK,-             ; WRITE PHYSICAL BLOCK
                READVBLK,-              ; READ VIRTUAL BLOCK
                WRITEVBLK,-             ; WRITE VIRTUAL BLOCK
                AVAILABLE,-             ; AVAILABLE
                ACCESS,-                ; ACCESS FILE / FIND DIRECTORY ENTRY
                ACPCONTROL,-            ; ACP CONTROL FUNCTION
                CREATE,-                ; CREATE FILE AND/OR DIRECTORY ENTRY
                DEACCESS,-              ; DEACCESS FILE
                DELETE,-                ; DELETE FILE AND/OR DIRECTORY ENTRY
                MODIFY,-                ; MODIFY FILE ATTRIBUTES
                MOUNT-                  ; MOUNT VOLUME
                >
```

```
FUNCTAB ,-                      ;BUFFERED FUNCTIONS
        <NOP,-                  ; NO-OP
        UNLOAD,-                ; UNLOAD
        SEEK,-                  ; SEEK
        DRVCLR,-                ; DRIVE CLEAR
        PACKACK,-               ; PACK ACKNOWLEDGE
        SENSECHAR,-             ; SENSE CHARACTERISTICS
        SETCHAR,-               ; SET CHARACTERISTICS
        SENSEMODE,-             ; SENSE MODE
        SETMODE,-               ; SET MODE
        AVAILABLE,-             ; AVAILABLE
        ACCESS,-                ; ACCESS FILE / FIND DIRECTORY ENTRY
        ACPCONTROL,-            ; ACP CONTROL FUNCTION
        CREATE,-                ; CREATE FILE AND/OR DIRECTORY ENTRY
        DEACCESS,-              ; DEACCESS FILE
        DELETE,-                ; DELETE FILE AND/OR DIRECTORY ENTRY
        MODIFY,-                ; MODIFY FILE ATTRIBUTES
        MOUNT-                  ; MOUNT VOLUME
        >
FUNCTAB DL_ALIGN,-              ;TEST ALIGNMENT FUNCTIONS
        <READHEAD,-             ; READ HEADER
        READLBLK,-              ; READ LOGICAL BLOCK
        READPBLK,-              ; READ PHYSICAL BLOCK
        READVBLK,-              ; READ VIRTUAL BLOCK
        WRITECHECK,-            ; WRITE CHECK
        WRITELBLK,-             ; WRITE LOGICAL BLOCK
        WRITEPBLK,-             ; WRITE PHYSICAL BLOCK
        WRITEVBLK-              ; WRITE VIRTUAL BLOCK
        >
FUNCTAB +ACP$READBLK,-          ;READ FUNCTIONS
        <READHEAD,-             ; READ HEADER
        READLBLK,-              ; READ LOGICAL BLOCK
        READPBLK,-              ; READ PHYSICAL BLOCK
        READVBLK-               ; READ VIRTUAL BLOCK
        >
FUNCTAB +ACP$WRITEBLK,-         ;WRITE FUNCTIONS
        <WRITECHECK,-           ; WRITE CHECK
        WRITELBLK,-             ; WRITE LOGICAL BLOCK
        WRITEPBLK,-             ; WRITE PHYSICAL BLOCK
        WRITEVBLK-              ; WRITE VIRTUAL BLOCK
        >
FUNCTAB +ACP$ACCESS,-           ;ACCESS FUNCTIONS
        <ACCESS,-               ; ACCESS FILE / FIND DIRECTORY ENTRY
        CREATE-                 ; CREATE FILE AND/OR DIRECTORY ENTRY
        >
FUNCTAB +ACP$DEACCESS,-         ;DEACCESS FUNCTION
        <DEACCESS-              ; DEACCESS FILE
        >
FUNCTAB +ACP$MODIFY,-           ;MODIFY FUNCTIONS
        <ACPCONTROL,-           ; ACP CONTROL FUNCTION
        DELETE,-                ; DELETE FILE AND/OR DIRECTORY ENTRY
        MODIFY-                 ; MODIFY FILE ATTRIBUTES
        >
FUNCTAB +ACP$MOUNT,-            ;MOUNT FUNCTION
        <MOUNT-                 ; MOUNT VOLUME
        >
FUNCTAB +EXE$LCLDSKVALID,-      ;LOCAL DISK VALID FUNCTIONS
        <UNLOAD,-               ;UNLOAD VOLUME
         AVAILABLE,-            ;UNIT AVAILABLE
         PACKACK-               ;PACK ACKNOWLEDGE
        >
FUNCTAB +EXE$ZEROPARM,-         ;ZERO PARAMETER FUNCTIONS
        <NOP,-                  ; NO-OP
        UNLOAD,-                ; UNLOAD
        DRVCLR,-                ; DRIVE CLEAR
        PACKACK,-               ; PACK ACKNOWLEDGE
        AVAILABLE,-             ; AVAILABLE
        >
```

```
        FUNCTAB +EXE$ONEPARM,-          ;ONE PARAMETER FUNCTION
                <SEEK-                  ; SEEK
                >
        FUNCTAB +EXE$SENSEMODE,-        ;SENSE FUNCTIONS
                <SENSECHAR,-            ; SENSE CHARACTERISTICS
                SENSEMODE-              ; SENSE MODE
                >
        FUNCTAB +EXE$SETCHAR,-          ;SET FUNCTIONS
                <SETCHAR,-              ; SET CHARACTERISTICS
                SETMODE-                ; SET MODE
                >
        .PAGE
        .SBTTL  CONTROLLER INITIALIZATION ROUTINE
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       THIS ROUTINE IS A NO-OP FOR THE RL11 BUT MUST BE INCLUDED
;       SINCE IT IS CALLED WHEN THE RL02 IS BOOTED AS A SYSTEM DEVICE.
;
;       THE OPERATING SYSTEM CALLS THIS ROUTINE:
;               - AT SYSTEM STARTUP
;               - DURING DRIVER LOADING
;               - DURING RECOVERY FROM POWER FAILURE
;
; INPUTS:
;
;       R4      - CSR ADDRESS (DEVICE CONTROL STATUS REGISTER)
;       R5      - IDB ADDRESS (INTERRUPT DATA BLOCK)
;       R6      - DDB ADDRESS (DEVICE DATA BLOCK)
;       R8      - CRB ADDRESS (CHANNEL REQUEST BLOCK)
;       ALL INTERRUPTS ARE LOCKED OUT
;
; OUTPUTS:
;
;       ALL REGISTERS EXCEPT R0-R3 ARE PRESERVED.
;       CONTROL IS RETURNED TO THE CALLER.
;
;--
DL_RL11_INIT:                           ;CONTROLLER INITIALIZATION
        ;
        ; FOR MICROVAX I, ALLOCATE A PHYSICALLY CONTIGUOUS BUFFER
        ; AREA FOR PERFORMING I/O.
        ;
        CPUDISP <<UV1,10$>>,-           ;FOR MICROVAX I, ALLOCATE BUFFER AREA
                CONTINUE=YES            ;FOR ALL CPU TYPES, WHICH INCLUDE, UP
                                        ;UNTIL NOW, 780, 785, 790, 750, 730,
                                        ;UV2, 8SS, 8NN
        BRB     20$                     ;FOR ALL OTHERS, SKIP BUFFER AREA
10$:    MOVZWL  #UCB$K_DL_BUFSZ,R1      ;LOAD SIZE OF BUFFER
        JSB     G^EXE$ALOPHYCNTG        ;ALLOCATE PHYSICALLY-CONTIGUOUS MEMORY
        BLBC    R0,20$                  ;EXIT ON ERROR
        MOVL    R2,CRB$L_AUXSTRUC(R8)   ;GET BUFFER VIRTUAL ADDRESS
        RSB                             ;RETURN TO CALLER
20$:    CLRL    CRB$L_AUXSTRUC(R8)      ;INDICATE MEMORY ALLOCATION FAILURE
        RSB                             ;RETURN TO CALLER
```

```
        .PAGE
        .SBTTL  UNIT INITIALIZATION ROUTINE
;++
;
; DL_RLOX_INIT - UNIT INITIALIZATION ROUTINE
;
; FUNCTIONAL DESCRIPTION:
;
;       THIS ROUTINE READIES THE RLO1/RLO2 UNITS FOR I/O OPERATIONS.
;
;       THE OPERATING SYSTEM CALLS THIS ROUTINE:
;               - AT SYSTEM STARTUP
;               - DURING DRIVER LOADING
;               - DURING RECOVERY FROM POWER FAILURE
;
; INPUTS:
;
;       R4      - CSR ADDRESS (CONTROLLER STATUS REGISTER)
;       R5      - UCB ADDRESS (UNIT-CONTROL BLOCK)
;
; OUTPUTS:
;
;       THE DRIVE UNIT IS RESET, UCB FIELDS ARE INITIALIZED, AND THE
;       ROUTINE WAITS FOR ONLINE UNITS TO SPIN UP.  ALL REGISTERS
;       EXCEPT RO-R3 ARE PRESERVED.
;
;--
DL_RLOX_INIT:                           ;RLO1/RLO2 UNIT INITIALIZATION
        MOVW    #1@UCB$V_DL_MAPPING,-   ;DEFAULT TO ADAPTER MAPPING
                UCB$W_DL_FLAGS(R5)      ;AND 18 BIT ADDRESSING
        ;
        ; SET CPU DEPENDENT UCB FLAGS FOR DL
        ;
        CPUDISP <<UV1,5$>,-
                 <UV2,1$>>,-
                CONTINUE=YES            ;FOR ALL OTHER CPU TYPES CONTINUE.
        BRB     10$                     ;FOR 790,785,780,750,730,8SS,8NN
1$:     BISW    #1@UCB$V_DL_22BIT,-     ;FOR MICROVAX II 22 BIT
                UCB$W_DL_FLAGS(R5)      ;ADDRESSING AS WELL AS ADAPTER MAPPING
        BRB     10$
5$:     MOVW    #1@UCB$V_DL_22BIT,-     ;FOR MICROVAX I 22 BIT
                UCB$W_DL_FLAGS(R5)      ;ADDRESSING AND NO ADAPTER MAPPING
10$:    MOVZWL  UCB$W_STS(R5),R3        ;SAVE CURRENT UNIT STATUS
        BICW    #UCB$M_ONLINE!UCB$M_VALID,-  ;ASSUME OFFLINE/INVALID
                UCB$W_STS(R5)           ;...

;
; WAIT FOR CONTROLLER (6 SECONDS MAX) IF CHANNEL IS BUSY WITH ANOTHER UNIT
;
        MOVL    UCB$L_CRB(R5),RO        ;GET CRB ADDRESS
        BBC     #CRB$V_BSY,CRB$B_MASK(RO),20$  ;IF CLEAR, CHANNEL NOT BUSY
        TIMEDWAIT TIME=#600*1000,-      ;6 SECOND WAIT LOOP
                INS1=<TSTB      RL_CS(R4)>,- ;IS CONTROLLER READY
                INS2=<BLSS      15$>,-  ;IF LSS, YES
                DONELBL=15$             ;LABEL TO EXIT WAIT LOOP
        BLBC    RO,25$                  ;TIME EXPIRED - EXIT
;
; GET CURRENT DRIVE STATUS AND RESET DRIVE
;
20$:    MOVW    #RL_DA_M_RST!-          ;PUT RESET AND GET STATUS IN DAR
                RL_DA_M_STS!RL_DA_M_MRK,RL_DA(R4)  ;...
        CLRL    R1                      ;CLEAR R1 FOR UNIT NUMBER
        INSV    UCB$W_UNIT(R5),#8,#8,R1 ;GET UNIT NUMBER
        BISW3   R1,#F_GETSTATUS,RL_CS(R4)  ;EXECUTE GET STATUS FUNCTION
        BSBW    DL_WAIT                 ;WAIT FOR CONTROLLER
        TSTB    RL_CS(R4)               ;WAS CONTROLLER READY?
        BGEQ    25$                     ;IF GEQ, NO
```

```
;
; CLASSIFY DRIVE TYPE
;
        MOVL    #^X2324C001,-
                 UCB$L_MEDIA_ID(R5)      ;SET MEDIA IDENT "DL RL01"
        BITW    #RL_MP_M_TYP,RL_MP(R4)  ;IS DRIVE TYPE = RL02?
        BNEQ    30$                     ;IF NEQ, YES
        MOVB    S^#DT$_RL01,-
                 UCB$B_DEVTYPE(R5)      ;SET RL01 DEVICE TYPE
        MOVW    #256,UCB$W_CYLINDERS(R5);SET NUMBER OF RL01 CYLINDERS
        MOVZWL  #10240,UCB$L_MAXBLOCK(R5) ;SET MAX RL01 BLOCK NUMBER
        BRB     40$
25$:    BRB     70$                     ;BRANCH TO COMMON EXIT
30$:    MOVB    S^#DT$_RL02,-
                 UCB$B_DEVTYPE(R5)      ;SET RL02 DEVICE TYPE
        MOVW    #512,UCB$W_CYLINDERS(R5);SET NUMBER OF RL02 CYLINDERS
        MOVZWL  #20480,UCB$L_MAXBLOCK(R5) ;SET MAX RL02 BLOCK NUMBER
        INCL    UCB$L_MEDIA_ID(R5)      ;SET MEDIA IDENT "DL RL02"
40$:    BBC     #UCB$V_VALID,R3,60$     ; BRANCH AROUND WAIT FOR DRIVE TO SPINUP
                                        ;  IF THE DRIVE DID NOT HAVE A VALID
                                        ;  VOLUME ON IT BEFORE POWER FAILURE.
;
; INITIALIZE UCB FIELDS AND WAIT FOR ONLINE UNITS TO SPIN UP
;
45$:    BITW    #RL_CS_M_DRDY,RL_CS(R4) ; IS DRIVE READY?
        BNEQ    50$                     ;IF NEQ, YES
        JSB     G^EXE$PWRTIMCHK         ;IS MAX TIME EXCEEDED?
        BLBS    R0,45$                  ;IF LBS, NO, MORE TIME NEEDED
        BRB     60$                     ;POWER UP TIME EXCEEDED
50$:
        BISW    #UCB$M_VALID,UCB$W_STS(R5) ;SET UCB STATUS VOLUME VALID
60$:    BBS     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),65$  ;IF BS, YES
        MOVL    UCB$L_CRB(R5),R1        ;GET CRB ADDRESS
        MOVL    CRB$L_AUXSTRUC(R1),R2   ;MEMORY ALLOC FAILURE DURING CTL INIT?
        BEQL    70$                     ;IF EQL, YES, LEAVE OFFLINE
        MOVL    R2,UCB$A_DL_BUF_VA(R5)  ;SAVE BUFFER'S VIRTUAL ADDRESS
        EXTZV   #VA$V_VPN,#VA$S_VPN,R2,R1;GET VIRTUAL PAGE NUMBER OF BUFFER
        MOVL    G^MMG$GL_SPTBASE,R0     ;GET BASE ADDRESS OF SPTS
        MOVL    (R0)[R1],R0             ;GET THE PTE CONTENTS
        BICL3   #^C<VA$M_BYTE>,R2,R1    ;GET BUFFER OFFSET (BA00-BA08)
        ASSUME  PTE$S_PFN GE 13
        INSV    R0,#9,#13,R1            ;COPY BA09-BA21
        MOVL    R1,UCB$A_DL_BUF_PA(R5)  ;SAVE PHYSICAL ADDRESS OF BUFFER
65$:    BISW    #UCB$M_ONLINE,UCB$W_STS(R5)  ;SET UCB STATUS VOLUME VALID
70$:    RSB
        .PAGE
        .SBTTL  DRIVER SPECIFIC SUBROUTINES
;
; DL_WAIT - WAIT FOR CONTROLLER READY
;
; INPUTS:
;       R4      - DEVICE CSR ADDRESS
;
; FUNCTIONAL DESCRIPTION:
;
;       THIS ROUTINE IS CALLED FROM THE DRIVER UNIT INITIALIZATION ROUTINE
;       TO WAIT UNTIL THE RL11 CONTROLLER IS READY.  TO PREVENT HANGING UP
;       AT HIGH IPL, A MAXIMUM OF 30 USEC ELAPSES BEFORE CONTROL IS
;       RETURNED TO THE CALLER.
;
DL_WAIT:                                ;WAIT FOR CONTROLLER READY
        MOVQ    R0,-(SP)                ;SAVE R0, R1
        DSBINT                          ;DISABLE INTERRUPTS
        TIMEWAIT        #3,#RL_CS_M_CRDY,RL_CS(R4),W
        ENBINT                          ;ENABLE INTERRUPTS
        MOVQ    (SP)+,R0                ;RESTORE R0, R1
        RSB                             ;RETURN TO UNIT INIT OR STARTIO
```

```
        .PAGE
        .SBTTL  FDT ROUTINE - TEST TRANSFER BYTE COUNT ALIGNMENT
;++
;
; DL_ALIGN - FDT ROUTINE TO TEST XFER BYTE COUNT
;
; FUNCTIONAL DESCRIPTION:
;
;       THIS ROUTINE IS CALLED FROM THE FUNCTION DECISION TABLE DISPATCHER
;       TO CHECK THE BYTE COUNT PARAMETER SPECIFIED BY THE USER PROCESS
;       FOR AN EVEN NUMBER OF BYTES (WORD BOUNDARY).
;
; INPUTS:
;
;       R3      - IRP ADDRESS (I/O REQUEST PACKET)
;       R4      - PCB ADDRESS (PROCESS CONTROL BLOCK)
;       R5      - UCB ADDRESS (UNIT-CONTROL BLOCK)
;       R6      - CCB ADDRESS (CHANNEL CONTROL BLOCK)
;       R7      - BIT NUMBER OF THE I/O FUNCTION CODE
;       R8      - ADDRESS OF FDT TABLE ENTRY FOR THIS ROUTINE
;       4(AP)   - ADDRESS OF FIRST FUNCTION DEPENDENT QIO PARAMETER
;
; OUTPUTS:
;
;       IF THE QIO BYTE COUNT PARAMETER IS ODD, THE I/O OPERATION IS
;       TERMINATED WITH AN ERROR. IF IT IS EVEN, CONTROL IS RETURNED
;       TO THE FDT DISPATCHER.
;
;--
DL_ALIGN:                               ;CHECK BYTE COUNT AT P1(AP)
        BLBS    4(AP),10$               ;IF LBS, ODD BYTE COUNT
        RSB                             ;EVEN - RETURN TO CALLER
10$:    MOVZWL  #SS$_IVBUFLEN,R0        ;SET BUFFER ALIGNMENT STATUS
        JMP     G^EXE$ABORTIO           ;ABORT I/O
        .PAGE
        .SBTTL  START I/O ROUTINE
;++
;
; DL_STARTIO - START I/O ROUTINE
;
; FUNCTIONAL DESCRIPTION:
;
;       THIS FORK PROCESS IS ENTERED FROM THE EXECUTIVE AFTER AN I/O REQUEST
;       PACKET HAS BEEN DEQUEUED, AND PERFORMS THE FOLLOWING:
;
;               - ACTIVATES THE DISK AFTER SETTING UCB FIELDS, OBTAINING
;                 UBA AND CONTROLLER RESOURCES, AND SETTING RL11 REGISTERS
;
;               - WAITS FOR AN INTERRUPT
;
;               - REGAINS CONTROL AFTER THE ISR SERVICES THE INTERRUPT, AND
;                       -  RE-ACTIVATES THE DISK IF THE ORIGINAL FUNCTION
;                          IS NOT YET COMPLETE, OR
;                       -  COMPLETES THE I/O REQUEST BY RELEASING RESOURCES,
;                          SETTING STATUS CODES, AND RETURNING TO THE EXECUTIVE.
```

```
;
; INPUTS:
;
;       R3              - IRP ADDRESS (I/O REQUEST PACKET)
;       R5              - UCB ADDRESS (UNIT-CONTROL BLOCK)
;       IRP$L_MEDIA     - PARAMETER LONGWORD (LOGICAL BLOCK NUMBER)
;
; OUTPUTS:
;
;       R0      - FIRST I/O STATUS LONGWORD: STATUS CODE & BYTES XFERED
;       R1      - SECOND I/O STATUS LONGWORD: 0 FOR DISKS
;
;       THE I/O FUNCTION IS EXECUTED.
;
;       ALL REGISTERS EXCEPT R0-R4 ARE PRESERVED.
;
;--
DL_STARTIO:                             ;START I/O OPERATION
;
;       COMPUTE PHYSICAL MEDIA ADDRESS
;
;               LBN = LBN * (SECTORS/BLOCK)
;               LBN/(SECTORS/TRACK) = D + SECTOR
;               D/(TRACKS/CYLINDER) = CYLINDER + TRACK
;
;
;       PREPROCESS UCB FIELDS
;
PREPROCESS:                             ;
        MOVL    IRP$L_MEDIA(R3),-       ; COPY GIVEN MEDIA ADDRESS (LOGICAL)
                UCB$L_MEDIA(R5)         ;  TO THE UCB.
        BBS     #IRP$V_PHYSIO,-         ;IF SET, PHYSICAL I/O
                IRP$W_STS(R3),10$
        MULL3   #2,UCB$L_MEDIA(R5),R0   ;SCALE LBN IN R0
        MOVZBL  UCB$B_SECTORS(R5),R2    ;GET NUMBER OF SECTORS PER TRACK
        CLRL    R1                      ;CLEAR HIGH PART OF DIVIDEND
        EDIV    R2,R0,R0,UCB$L_MEDIA(R5);CALCULATE SECTOR NUMBER AND STORE
        MOVZBL  UCB$B_TRACKS(R5),R2     ;GET NUMBER OF TRACKS PER CYLINDER
        EDIV    R2,R0,R0,R1             ;CALCULATE TRACK AND CYLINDER
        MOVB    R1,UCB$L_MEDIA+1(R5)    ;STORE TRACK NUMBER
        MOVW    R0,UCB$L_MEDIA+2(R5)    ;STORE CYLINDER NUMBER
10$:
        MOVB    UCB$B_ERTMAX(R5),-      ;INITIALIZE ERROR RETRY COUNT
                UCB$B_ERTCNT(R5)        ;...
        MNEGW   UCB$W_BCNT(R5),UCB$W_BCR(R5)  ;INIT NEG BYTES LEFT TO XFER
        CLRW    UCB$W_DL_DPN(R5)        ;CLEAR DATA-PATH NO. FOR USE AS A
                                        ; UBA-RESOURCE-ALLOCATION FLAG
        CLRB    UCB$B_DL_DPPE(R5)       ;CLEAR DATA-PATH-PURGE-ERROR REGISTER
        MOVW    IRP$W_FUNC(R3),UCB$W_FUNC(R5)  ;SAVE FUNCTION CODE
        EXTZV   #IRP$V_FCODE,-          ;EXTRACT I/O FUNCTION CODE
                #IRP$S_FCODE,IRP$W_FUNC(R3),R1  ;...
        MOVB    R1,UCB$B_FEX(R5)        ;STORE FUNCTION DISPATCH INDEX
        CMPB    #IO$_SEEK,R1            ;SEEK FUNCTION?
        BNEQ    20$                     ;IF NEQ, NO
        MOVW    IRP$L_MEDIA(R3),-       ;STORE CYLINDER ADDRESS
                UCB$W_DC(R5)            ;...
20$:
        BICW    #UCB$M_DIAGBUF,-
                UCB$W_DEVSTS(R5)        ;CLR DIAGNOSTIC BUFFER PRESENT
        BBC     #IRP$V_DIAGBUF,-        ;IF CLR, NO DIAG BUFFER
                IRP$W_STS(R3),FDISPATCH ;...
        BISW    #UCB$M_DIAGBUF,UCB$W_DEVSTS(R5)  ;SET DIAG BUFFER PRESENT
```

```
;
;       CENTRAL FUNCTION DISPATCH
;
FDISPATCH:                              ;FUNCTION DISPATCH
        MOVL    UCB$L_IRP(R5),R3        ;GET IRP ADDRESS
        BBS     #IRP$V_PHYSIO,-         ;IF SET, PHYSICAL I/O FUNCTION
                IRP$W_STS(R3),10$       ;...
        BBS     #UCB$V_VALID,-          ;IF SET, VOLUME SOFTWARE VALID
                UCB$W_STS(R5),10$       ;...
        MOVZWL  #SS$_VOLINV,R0          ;SET VOLUME INVALID STATUS
        BRW     RESETXFR                ;RESET BYTE COUNT AND EXIT
10$:    CLRB    UCB$B_DL_DCHEK(R5)      ;CLEAR DATA CHECK IN PROGRESS
        MOVZBL  UCB$B_FEX(R5),R3        ;GET FUNCTION DISPATCH INDEX
        CASE    R3,<-                   ;DISPATCH TO FUNCTION HANDLING ROUTINE
                UNLOAD,-                ; UNLOAD
                SEEK,-                  ; SEEK
                NOP,-                   ; RECALIBRATE (UNSUPPORTED)
                DRVCLR,-                ; DRVCLR
                NOP,-                   ; RELEASE PORT (UNSUPPORTED)
                NOP,-                   ; OFFSET HEADS (UNSUPPORTED)
                NOP,-                   ; RETURN TO CENTER (UNSUPPORTED)
                PACKACK,-               ; PACK ACKNOWLEDGE
                NOP,-                   ; SEARCH (UNSUPPORTED)
                WRITECHECK,-            ; WRITE CHECK
                WRITEDATA,-             ; WRITE DATA
                READDATA,-              ; READ DATA
                NOP,-                   ; WRITE HEADER (UNSUPPORTED)
                READHEAD,-              ; READ HEADER
                NOP,-                   ; PLACE HOLDER
                NOP,-                   ; PLACE HOLDER
                AVAILABLE-              ; AVAILABLE
                >,LIMIT=#CDF_UNLOAD     ;

NOP:                                    ;NO-OP
SEEK:                                   ;SEEK
DRVCLR:                                 ;DRIVE CLEAR (GET STATUS & RESET)
DO_FUNCTION:
        EXFUNCL RETRYERR                ;EXECUTE FUNCTION - RETRY IF FAILURE
        BRB     NORMAL                  ;SUCCESSFUL - EXIT WITH NORMAL STATUS

PACKACK:                                ;PACK ACKNOWLEDGE (GET STATUS & RESET)
        BISW    #UCB$M_VALID, -         ;SET SOFTWARE VOLUME VALID BIT.
                UCB$W_STS(R5)
        BRB     DO_FUNCTION             ;THEN GO DO HARDWARE FUNCTION.

UNLOAD:                                 ;UNLOAD
AVAILABLE:                              ;AVAILABLE
        BICW    #UCB$M_VALID, -         ;CLEAR SOFTWARE VOLUME VALID BIT,
                UCB$W_STS(R5)           ;AND GO COMPLETE OPERATION WITHOUT
        BRB     NORMAL                  ;ANY HARDWARE INTERACTION.

WRITECHECK:                             ;WRITE CHECK
READHEAD:                               ;READ HEADER
        BICW    #IO$M_DATACHECK,-       ;CLEAR DATA CHECK REQUEST
                UCB$W_FUNC(R5)          ;TO PREVENT EXTRA WRITE CHECK
WRITEDATA:                              ;WRITE DATA
READDATA:                               ;READ DATA
        EXFUNCL RETRYERR,F_SEEK         ;EXECUTE EXPLICIT SEEK - RETRY IF FAIL
        MOVZBL  UCB$B_FEX(R5),R3        ;GET FUNCTION DISPATCH INDEX
        EXFUNCL RETRYERR                ;EXECUTE TRANSFER FUNCTION

;
;       OPERATON COMPLETION
;
NORMAL:                                 ;SUCCESSFUL OPERATION COMPLETE
        MOVZWL  #SS$_NORMAL,R0          ;SET NORMAL COMPLETION STATUS
        BRW     FUNCXT                  ;FUNCTION EXIT

RETRYERR:                               ;RETRIABLE ERROR
        DECB    UCB$B_ERTCNT(R5)        ;ANY RETRIES LEFT?
        BEQL    FATALERR                ;IF EQL, NO
        BRW     FDISPATCH               ;RETRY FUNCTION
```

```
FATALERR:                                   ;UNRECOVERABLE ERROR
        MOVZWL  #SS$_VOLINV,R0              ;ASSUME VOLUME INVALID STATUS
        BBS     #RL_MP_V_VC,-               ;IF SET, VOLUME INVALID
                UCB$W_DL_MP(R5),FUNCXT      ;...
        MOVZWL  #SS$_WRITLCK,R0             ;ASSUME WRITE LOCK ERROR STATUS
        BBC     #RL_MP_V_WL,-               ;IF CLR, VOLUME NOT WRITE LOCKED
                UCB$W_DL_MP(R5),5$          ;...
        BBS     #RL_MP_V_WGE,-              ;IF SET, WRITE GATE ERROR
                UCB$W_DL_MP(R5),FUNCXT      ;IF WL & WGE SET, WRITE LOCK ERROR
5$:     MOVZWL  #SS$_DATACHECK,R0           ;ASSUME DATA CHECK ERROR STATUS
        TSTB    UCB$B_DL_DCHEK(R5)          ;WRITE CHECK IN PROGRESS?
        BEQL    10$                         ;IF EQL, NO
        BBS     #RL_CS_V_OPI,-              ;IF SET, NOT WRITE CHECK ERROR
                UCB$W_DL_CS(R5),10$         ;...
        BBS     #RL_CS_V_CRC,-              ;IF SET, WRITE CHECK ERROR
                UCB$W_DL_CS(R5),FUNCXT      ;...
10$:    MOVZWL  #SS$_PARITY,R0              ;ASSUME PARITY ERROR STATUS
        BBS     #RL_CS_V_CRC,-              ;IF SET, CRC ERROR
                UCB$W_DL_CS(R5),FUNCXT      ;OR DATA-PATH-PURGE ERROR
20$:    MOVZWL  #SS$_DRVERR,R0              ;ASSUME DRIVE ERROR STATUS
        BBS     #RL_CS_V_DE,-               ;IF SET, DRIVE ERROR
                UCB$W_DL_CS(R5),FUNCXT      ;...
        MOVZWL  #SS$_CTRLERR,R0             ;ASSUME CONTROLLER ERROR STATUS
FUNCXT:                                     ;FUNCTION EXIT
        PUSHL   R0                          ;SAVE FINAL REQUEST STATUS
        JSB     G^IOC$DIAGBUFILL            ;FILL DIAGNOSTIC BUFFER IF PRESENT
        CMPB    #CDF_WRITECHECK,UCB$B_FEX(R5)  ;DRIVE RELATED FUNCTION?
        BGTRU   10$                         ;IF GTRU, YES
        CMPB    #CDF_AVAILABLE,UCB$B_FEX(R5)   ;DRIVE RELATED FUNCTION?
        BEQL    10$                         ;IF EQL, YES
        MOVL    UCB$L_IRP(R5),R3            ;RETRIEVE ADDRESS OF IRP
        ADDW3   UCB$W_BCR(R5),-             ;CALCULATE BYTES TRANSFERRED
                IRP$W_BCNT(R3),2(SP)        ;...
        TSTW    UCB$W_DL_DPN(R5)            ;ARE UBA RESOURCES ALLOCATED?
        BEQL    20$                         ;IF EQL, NO
        BBC     #UCB$V_DL_MAPPING,-         ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),10$      ;IF BC, NO
        RELDPR                              ;RELEASE DATA PATH
        RELMPR                              ;RELEASE MAP REGISTERS
        BRB     20$                         ;JOIN COMMON CODE
10$:    MOVL    UCB$L_DL_SVAPTE(R5),-       ;RESTORE ORIGINAL SVAPTE
                UCB$L_SVAPTE(R5)            ;
20$:    RELCHAN                             ;RELEASE CHANNEL IF OWNED
        CLRL    R1                          ;CLEAR SECOND STATUS LONGWORD
        POPL    R0                          ;RETRIEVE FINAL REQUEST STATUS
        REQCOM                              ;COMPLETE REQUEST
        .PAGE
;
; FEXL - RL11 HARDWARE FUNCTION EXECUTION
;
; THIS ROUTINE IS CALLED VIA A BSB WITH A BYTE IMMEDIATELY FOLLOWING THAT
; SPECIFIES THE ADDRESS OF AN ERROR ROUTINE. ALL DATA IS ASSUMED TO HAVE BEEN
; SET UP IN THE UCB BEFORE THE CALL. THE APPROPRIATE PARAMETERS ARE LOADED
; INTO DEVICE REGISTERS AND THE FUNCTION IS INITIATED. THE RETURN ADDRESS
; IS STORED IN THE UCB AND A WAITFOR INTERRUPT IS EXECUTED. WHEN THE
; INTERRUPT OCCURS, CONTROL IS RETURNED TO THE CALLER.
;
```

```
; INPUTS:
;
;       R3 = FUNCTION TABLE DISPATCH INDEX
;       R5 = DEVICE UNIT UCB ADDRESS
;
;       00(SP) = RETURN ADDRESS OF CALLER
;       04(SP) = RETURN ADDRESS OF CALLER'S CALLER
;
;       IMMEDIATELY FOLLOWING INLINE AT THE CALL SITE IS A BYTE WHICH CONTAINS
;       A BRANCH DESTINATION TO AN ERROR RETRY ROUTINE.
;
; OUTPUTS:
;
;       THERE ARE FOUR EXITS FROM THIS ROUTINE:
;
;       1. SPECIAL CONDITION - THIS EXIT IS TAKEN IF A POWER FAILURE OCCURS
;              OR THE OPERATION TIMES OUT. IT IS A JUMP TO THE APPROPRIATE
;              ERROR ROUTINE.
;
;       2. FATAL ERROR - THIS EXIT IS TAKEN IF A FATAL CONTROLLER OR DRIVE
;              ERROR OCCURS OR IF ANY ERROR OCCURS AND ERROR RETRY IS EITHER
;              INHIBITED OR EXHAUSTED. IT IS A JUMP TO THE FATAL ERROR EXIT
;              ROUTINE.
;
;       3. RETRIABLE ERROR - THIS EXIT IS TAKEN IF A RETRIABLE CONTROLLER
;              OR DRIVE ERROR OCCURS AND ERROR RETRY IS NEITHER INHIBITED
;              NOR EXHAUSTED. IT CONSISTS OF TAKING THE ERROR BRANCH EXIT
;              SPECIFIED AT THE CALL SITE.
;
;       4. SUCCESSFUL OPERATION - THIS EXIT IS TAKEN IF NO ERRORS OCCUR
;              DURING THE OPERATION. IT CONSISTS OF A RETURN INLINE.
;
;       IN ALL CASES IF AN ERROR OCCURS, AN ATTEMPT IS MADE TO LOG THE ERROR.
;
;       IN ALL CASES FINAL DEVICE REGISTERS ARE RETURNED VIA THE UCB.
;
;       UCB$W_BCR(R5) = NEGATIVE BYTES REMAINING TO TRANSFER
        .PAGE
FEXL:                                   ;FUNCTION EXECUTOR
        POPL    UCB$L_DPC(R5)           ;SAVE DRIVER PC VALUE
        MOVB    R3,UCB$B_CEX(R5)        ;SAVE CASE INDEX
        MOVL    UCB$L_CRB(R5),R0        ;GET ADDRESS OF PRIMARY CRB
        MOVL    CRB$L_INTD+VEC$L_IDB(R0),R1  ;GET ADDRESS OF IDB
        CMPL    R5,IDB$L_OWNER(R1)      ;DOES THIS PROCESS OWN CHANNEL?
        BNEQ    10$                     ;IF NEQ, NO
        MOVL    IDB$L_CSR(R1),R4        ;SET ASSIGNED CHANNEL CSR ADDRESS
        BRB     20$                     ;
10$:    REQPCHAN                        ;REQUEST CHANNEL (RETURNS R4 = CSR ADR)
20$:    CASE    R3,<-                   ;DISPATCH TO PROPER FUNCTION ROUTINE
                IMMED,-                 ;NO OPERATION
                IMMED,-                 ;UNLOAD VOLUME (NOP)
                POSIT,-                 ;SEEK CYLINDER
                IMMED,-                 ;RECALIBRATE (NOP)
                DRCLR,-                 ;DRIVE CLEAR (GET STATUS & RESET)
                IMMED,-                 ;RELEASE DRIVE (NOP)
                IMMED,-                 ;OFFSET HEADS (NOP)
                IMMED,-                 ;RETURN TO CENTERLINE (NOP)
                DRCLR,-                 ;PACK ACKNOWLEDGE
                IMMED,-                 ;SEARCH (NOP)
                >                       ;
        BRW     XFER                    ;TRANSFER FUNCTION
```

```
        .PAGE
;
; IMMEDIATE FUNCTION EXECUTION
;
;       FUNCTIONS INCLUDE:
;
;               NO OPERATION,
;               DRIVE CLEAR, AND
;               PACK ACKNOWLEDGE
;
; INPUTS:
;       R3      - CASE INDEX
;       R4      - CSR ADDRESS
;       R5      - UCB ADDRESS
;
; FUNCTIONAL DESRIPTION:
;
; INTERRUPTS ARE LOCKED OUT, THE APPROPRIATE FUNCTION IS INITIATED WITH
; INTERRUPT ENABLE, AND A WAITFOR INTERRUPT AND KEEP CHANNEL IS EXECUTED.
;
DRCLR:                                  ;DRIVE CLEAR
        BISW    #RL_DA_M_STS!-          ;SET GETSTATUS,RESET,AND MARK IN DAR
                RL_DA_M_RST!RL_DA_M_MRK,RL_DA(R4)  ;...
IMMED:                                  ;IMMEDIATE FUNCTION EXECUTION
        CKPWR                           ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,FTAB[R3],RL_CS(R4)   ;MERGE UNIT WITH FNTN AND EXECUTE
        WFIKPCH RETREG,#2               ;WAITFOR INTERRUPT
        IOFORK                          ;RETURN FROM ISR-
                                        ;CREATE FORK PROCESS (&JSB BACK TO ISR)
        BRW     RETREG                  ;
        .PAGE
;
; POSITIONING FUNCTION EXECUTION
;
;       FUNCTIONS INCLUDE:
;
;               SEEK CYLINDER
;
; INPUTS:
;       R3      - CASE INDEX
;       R4      - DEVICE CSR ADDRESS
;       R5      - UCB ADDRESS
;
; FUNCTIONAL DESRIPTION:
;
; THE CYLINDER DIFFERENCE WORD IS CALCULATED AND LOADED INTO THE DISK
; ADDRESS REGISTER, INTERRUPTS ARE LOCKED OUT, AND THE SEEK FUNCTION
; IS INITIATED WITHOUT INTERRUPT ENABLE. THE CONTROLLER IS THEN POLLED
; FOR READY, AND DEVICE INTERRUPTS ARE ENABLED.
;
; SINCE THE RL01/RL02 DO NOT ISSUE AN INTERRUPT UPON COMPLETION OF A
; SEEK, OVERLAPPED SEEKS ARE NOT ATTEMPTED, AND ONE OF THE FOLLOWING IS
; PERFORMED.
;
;       IF ONLY A SEEK FUNCTION IS BEING REQUESTED, A DUMMY READ HEADER
;       FUNCTION IS ISSUED AND A WAITFOR INTERRUPT IS INITIATED.
;       THE READ HEADER IS USED TO SIGNAL THE END OF THE SEEK, SINCE IT
;       WILL ISSUE AN INTERRUPT SHORTLY (315 USEC AVG) AFTER THE SEEK IS
;       COMPLETE. IT WILL ALSO SENSE FOR A TIMEOUT DURING THE SEEK.
;
;       IF THE SEEK IS ASSOCIATED WITH A DATA TRANSFER REQUEST (RL01/RL02
;       TRANSFER FUNCTIONS REQUIRE EXPLICIT SEEKS), THE PROGRAM KEEPS THE
;       CHANNEL AND RETURNS TO FDISPATCH TO ISSUE THE TRANSFER REQUEST
;       WHILE THE SEEK IS STILL IN PROGREES. WHEN THE SEEK COMPLETES, THE
;       RL11 CONTROLLER WILL BEGIN THE TRANSFER.
;
```

```
POSIT:                                  ;POSITIONING FUNCTION
;
; OBTAIN CURRENT DISK ADDRESS
;
; IF THERE HAS NOT BEEN A PREVIOUS TRANSFER DURING THIS REQUEST,
; A READ HEADER IS EXECUTED TO DETERMINE THE CURRENT DISK ADDRESS.
;
        TSTW    UCB$W_DL_DPN(R5)        ;WAS THERE A PREVIOUS TRANSFER?
        BEQL    10$                     ;IF EQL, NO, READ HEADER
        BICW3   #^O77,UCB$W_DL_DA(R5),R1  ;PUT CURRENT CYL & SURFACE IN R1
        BRW     60$                     ;CALCULATE DIFFERENCE WORD
10$:    MOVZBL  #8,R3                   ;SET READ HEADER RETRY COUNT IN R3
20$:    CKPWR                           ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,#F_READHEAD!RL_CS_M_IE,-  ;EXECUTE READ HEADER
                RL_CS(R4)               ;...
        WFIKPCH 40$,#2                  ;WAIT FOR INTERRUPT OR TIMEOUT
        IOFORK                          ;CREATE FORK PROCESS
        BITW    #RL_CS_M_CE,UCB$W_DL_CS(R5)  ;ANY ERRORS?
        BEQL    50$                     ;IF EQL, NO
        DECB    R3                      ;DECREMENT READ HEADER RETRY COUNT
        BNEQ    20$                     ;IF NEQ, RETRY READ HEADER
                                        ;IF EQL, READ HEADER RETRY EXHAUSTED -
                                        ;TRY PREVIOUS TRACK
        MOVW    #^O200!RL_DA_M_MRK,-    ;LOAD REVERSE SEEK DIFFERENCE WORD
                RL_DA(R4)               ;...
        CKPWR                           ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,#F_SEEK!RL_CS_M_IE,- ;EXECUTE REVERSE SEEK
                RL_CS(R4)               ;...
        WFIKPCH 40$,#2                  ;WAIT FOR SEEK TO BEGIN (INTERRUPT)
        IOFORK                          ;CREATE FORK PROCESS
        CKPWR                           ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,#F_READHEAD!RL_CS_M_IE,-  ;TRY READ HEADER ON NEW TRACK
                RL_CS(R4)               ;...
        WFIKPCH 40$,#2                  ;WAITFOR INTERRUPT OR TIMEOUT
        IOFORK                          ;CREATE FORK PROCESS
        BITW    #RL_CS_M_CE,UCB$W_DL_CS(R5)  ;READ HEADER ERROR?
        BEQL    50$                     ;IF EQL, NO
40$:                                    ;CANNOT READ CURRENT DISK ADDRESS
;       CLRB    UCB$B_ERTCNT(R5)        ;CLEAR RETRY COUNT
        BRW     RETREG                  ;
50$:                                    ;FOUND CURRENT DISK ADDRESS
        BICW3   #^O77,UCB$W_DL_MP(R5),R1  ;PUT CURRENT CYL & SURFACE IN R1
;
; CALCULATE CYLINDER DIFFERENCE WORD
;
60$:    CLRL    R0                      ;CLEAR R0 FOR DESIRED ADDRESS
        INSV    UCB$W_DA+1(R5),#6,#1,R0 ;INSERT DESIRED SURFACE IN R0<6>
        INSV    UCB$W_DC(R5),#7,#9,R0   ;INSERT DESIRED CYLINDER IN R0<15:7>
        CMPW    R0,R1                   ;IS A SEEK NEEDED?
        BEQL    80$                     ;IF EQL, NO
        BICW    #^O177,R1               ;REMOVE SURFACE BIT
        BICW    #^O177,R0               ;REMOVE SURFACE BIT
        SUBW    R0,R1                   ;SUBTRACT DESIRED FROM ACTUAL
        BEQL    70$                     ;IF EQL, ONLY CHANGE SURFACE
        BCC     70$                     ;IF CC, ACTUAL>=DESIRED
        MNEGW   R1,R1                   ;ACTUAL<DESIRED, MAKE POSITIVE DIFF
        BISW    #4,R1                   ;SET SIGN FOR MOVE TO CENTER OF DISK
70$:    INSV    UCB$W_DA+1(R5),#4,#1,R1 ;INSERT SURFACE BIT
        BISW3   #RL_DA_M_MRK,R1,RL_DA(R4)  ;SET MARKER AND LOAD DIFFERENCE WORD
```

```
;
; EXECUTE SEEK
;
        CKPWR                           ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,#F_SEEK!RL_CS_M_IE,- ;EXECUTE SEEK FUNCTION
                RL_CS(R4)               ;...
        WFIKPCH 40$,#2                  ;WAIT FOR SEEK TO BEGIN (INTERRUPT)
        IOFORK                          ;CREATE FORK PROCESS
80$:    CMPB    #IO$_SEEK,UCB$B_FEX(R5) ;IS SEEK ASSOCIATED WITH A TRANSFER?
        BEQL    90$                     ;IF EQL, NO, SEEK ONLY
;
; RETURN FOR SEEK ASSOCIATED WITH A TRANSFER REQUEST
;
        INCL    UCB$L_DPC(R5)           ;ADJUST TO CORRECT RETURN ADDRESS
        JMP     @UCB$L_DPC(R5)          ;RETURN TO DRIVER FOR TRANSFER
;
; RETURN FOR SEEK ONLY REQUEST
;
90$:    CKPWR                           ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,#F_READHEAD!RL_CS_M_IE,-  ;EXECUTE DUMMY READ HEADER
                RL_CS(R4)               ;...
        WFIKPCH RETREG,#2               ;WAIT FOR SEEK TO COMPLETE (INTERRUPT)
        IOFORK                          ;CREATE FORK PROCESS
        BRW     RETREG                  ;
        .PAGE
;
; TRANSFER FUNCTION EXECUTION
;
;       FUNCTIONS INCLUDE:
;
;               WRITE CHECK
;               WRITE DATA
;               READ DATA, AND
;               READ HEADER
;
; INPUTS:
;       R3      - CASE INDEX
;       R4      - DEVICE CSR ADDRESS
;       R5      - UCB ADDRESS
;
; FUNCTIONAL DESCRIPTION:
;
; A UNIBUS DATA PATH IS REQUESTED FOLLOWED BY THE APPROPRIATE NUMBER OF MAP
; REGISTERS REQUIRED FOR THE TRANSFER. THE TRANSFER PARAMETERS ARE LOADED
; INTO THE DEVICE REGISTERS, INTERRUPTS ARE LOCKED OUT, THE FUNCTION IS
; INITIATED, AND A WAITFOR INTERRUPT AND KEEP CHANNEL IS EXECUTED.
;
; UPON RETURN FROM THE INTERRUPT SERVICE ROUTINE, IF THE TRANSFER IS
; COMPLETE, THE APPROPRIATE EXIT IS TAKEN. IF THE FUNCTION IS NOT COMPLETE
; TRANSFER PARAMETERS ARE UPDATED AND A RETURN TO FDISPATCH IS EXECUTED TO
; RE-ISSUE SEEK AND TRANSFER FUNCTIONS WHILE KEEPING CHANNEL AND UBA
; RESOURCES. IF A DATA CHECK HAS BEEN REQUESTED, IT IS PERFORMED
; BEFORE RETURNING TO FDISPATCH.
;
XFER:                                   ;TRANSFER FUNCTION EXECUTION
        BBS     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),2$   ;BRANCH IF ADAPTER MAPPING.
        MOVW    UCB$A_DL_BUF_PA(R5),UCB$W_DL_SBA(R5);GET 1ST WORD OF BUFFER ADDR
        MOVZWL  UCB$A_DL_BUF_PA+2(R5),R0;GET BITS 16:21 OF BUFFER ADDRESS
        MOVW    R0,RL_BAE(R4)           ;SET MEMORY EXTENSION BITS IN BAE
        ASHL    #4,R0,R0                ;PUT MEMORY EXTENSION BITS IN <5:4>
        MOVB    R0,UCB$B_DL_XBA(R5)     ;OF CSR
```

```
;
; FIRST TRANSFER OF THIS I/O REQUEST - ALLOCATE RESOURCES
;
        TSTW    UCB$W_DL_DPN(R5)        ;RESOURCES ALREADY ALLOCATED?
        BNEQ    5$                      ;IF NEQ, YES
        CLRL    UCB$A_DL_MOVRTN(R5)     ;ASSUME READ
        CMPB    #CDF_WRITEDATA,R3       ;WRITE DATA?
        BNEQ    1$                      ;IF NEQ, NO
        MOVAB   G^IOC$MOVFRUSER,-       ;SET MOVE ROUTINE ADDRESS FOR
                UCB$A_DL_MOVRTN(R5)     ;1ST PARTIAL WRITE
1$:     MOVL    UCB$L_SVAPTE(R5),UCB$L_DL_SVAPTE(R5);SAVE SVAPTE FOR BUFFER COPY
        MNEGW   #1,UCB$W_DL_DPN(R5)     ;SET FIRST XFER FLAG
        BRB     5$                      ;JOIN COMMON CODE

;
; FIRST TRANSFER OF THIS I/O REQUEST - ALLOCATE RESOURCES
;
2$:     TSTW    UCB$W_DL_DPN(R5)        ;UBA RESOURCES ALREADY ALLOCATED?
        BNEQ    5$              ;IF NEQ, YES
        REQDPR                  ;REQUEST DATA PATH
        REQMPR                  ;REQUEST MAP REGISTERS
        LOADUBA                 ;LOAD UNIBUS MAP REGISTERS
        MOVL    UCB$L_CRB(R5),R1        ;GET CRB ADDRESS
        EXTZV   #VEC$V_DATAPATH,#VEC$S_DATAPATH,-  ;EXTRACT DATA-PATH NUMBER
                CRB$L_INTD+VEC$B_DATAPATH(R1),R0   ; FOR UBA-RESOURCE FLAG
        MOVW    R0,UCB$W_DL_DPN(R5)     ;INDICATE UBA RESOURCES ALLOCATED
        MOVZWL  UCB$W_BOFF(R5),R0       ;GET BYTE OFFSET IN PAGE
        INSV    CRB$L_INTD+VEC$W_MAPREG(R1),-  ;INSERT HIGH 7 BITS OF ADDRESS
                #9,#7,R0                ;...
        MOVW    R0,UCB$W_DL_SBA(R5)     ;SET BUFFER ADDRESS
        EXTZV   #7,#2,CRB$L_INTD+VEC$W_MAPREG(R1),R0  ;GET MEMORY EXTENSION BITS
        MULB3   #16,R0,UCB$B_DL_XBA(R5) ;POSITION MEMORY EXTENSION BITS TO <5:4>

;
; COMMON TRANSFER POINT
;

;
; FOR A READ OPERATION WHEN NO ADAPTER MAPPING IS PRESENT EMPTY THE
; INTERNAL PHYSICALLY CONTIGUOUS BUFFER FROM THE PREVIOUS READ TO THE
; USER'S BUFFER.
;
5$:     BSBW    DL_MOVE_TO_BUFFER       ;COPY TO USER BUFFER
;
; PUT BUFFER ADDRESS, WORD COUNT, AND DISK ADDRESS IN DEVICE REGISTERS
;
        MOVW    UCB$W_DL_SBA(R5),RL_BA(R4)  ;SET BUFFER ADDRESS
        MNEGW   UCB$W_BCR(R5),-         ;GET BYTES LEFT TO TRANSFER AND -
                UCB$W_DL_PBCR(R5)       ;ASSUME ONLY ONE TRANSFER NEEDED
        MOVZBL  UCB$B_SECTORS(R5),R2    ;GET SECTORS/SURFACE
        MOVZBL  UCB$W_DA(R5),R1         ;GET DESIRED SECTOR
        SUBW    R1,R2                   ;CALCULATE SECTORS LEFT ON SURFACE
        MULW    #256,R2                 ;CONVERT TO BYTES LEFT ON SURFACE
        CMPW    UCB$W_DL_PBCR(R5),R2    ;ARE ADDITIONAL TRANSFERS REQUIRED?
        BLEQU   10$                     ;IF LEQU, NO
        MOVW    R2,UCB$W_DL_PBCR(R5)    ;SET BYTE COUNT FOR THIS TRANSFER
```

```
;
; FOR A WRITE OPERATION WHEN NO ADAPTER MAPPING IS PRESENT
; FILL INTERNAL PHYSICALLY CONTIGUOUS BUFFER FROM THE USER'S BUFFER.
;
10$:    BSBW    DL_MOVE_FROM_BUFFER     ;COPY FROM USER BUFFER
        MOVZBL  UCB$B_DL_XBA(R5),R0     ;SET MEMORY EXTENSION BITS
        BISW    FTAB[R3],R0             ;MERGE XBA BITS WITH FUNCTION
        DIVW3   #2,UCB$W_DL_PBCR(R5),R2 ;CALCULATE TRANSFER WORD COUNT
        MNEGW   R2,RL_MP(R4)            ;SET TRANSFER WORD COUNT
        MOVZBL  UCB$W_DA(R5),R1         ;PUT DESIRED SECTOR IN R1<5:0>
        INSV    UCB$W_DA+1(R5),#6,#1,R1 ;INSERT DESIRED SURFACE IN R1<6>
        INSV    UCB$W_DC(R5),#7,#9,R1   ;INSERT DESIRED CYLINDER IN R1<15:7>
        MOVW    R1,RL_DA(R4)            ;SET DESIRED DISK ADDRESS
;
; EXECUTE THE TRANSFER FUNCTION
;
        CKPWR                           ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,R0,RL_CS(R4)         ;EXECUTE FUNCTION
        WFIKPCH RETREG,#6               ;WAITFOR INTERRUPT AND KEEP CHANNEL
                                        ;RETURN HERE FROM ISR SAVING REGISTERS
        IOFORK                          ;CREATE FORK PROCESS (RETURN TO ISR)
                                        ;RETURN HERE FROM ISR REI ROUTINE
;
; PURGE DATA PATH
;
        CLRB    UCB$B_DL_DPPE(R5)       ;CLEAR DATA-PATH-PURGE ERROR
        JSB     G^IOC$PURGDATAP         ;PURGE DATA PATH
        BLBS    R0,20$                  ;IF SET, NO PURGE ERRORS
        INCB    UCB$B_DL_DPPE(R5)       ;SET DATA-PATH-PURGE ERROR
;
; SAVE UBA REGISTERS FOR UPDATE AND REGDUMP ROUTINES
;
20$:    BBC     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),30$  ;IF BC, NO
        MOVL    R1,UCB$L_DL_DPR(R5)     ;SAVE DATA-PATH REGISTER
        EXTZV   #9,#7,UCB$W_DL_BA(R5),R0  ;EXTRACT LOW BITS OF FINAL MAP REG NO.
        EXTZV   #4,#2,UCB$W_DL_CS(R5),R1  ;EXTRACT HIGH BITS OF FINAL MAP REG NO.
        INSV    R1,#7,#2,R0             ;INSERT HIGH BITS OF FINAL MAP REGISTER
        CMPW    #495,R0                 ;LEGAL MAP REGISTER NUMBER?
        BGEQ    25$                     ;IF GEQ, YES
        MOVZWL  #495,R0                 ;RESTRICT MAP REGISTER NUMBER
25$:    MOVL    (R2)[R0],UCB$L_DL_FMPR(R5)  ;SAVE FINAL MAP REGISTER NUMBER
        CLRL    UCB$L_DL_PMPR(R5)       ;CLEAR PREVIOUS MAP REGISTER CONTENTS
        DECL    R0                      ;CALCULATE PREVIOUS MAP REGISTER NUMBER
        CMPV    #VEC$V_MAPREG,#VEC$S_MAPREG,-  ;ANY PREVIOUS MAP REGISTER?
                CRB$L_INTD+VEC$W_MAPREG(R3),R0 ;...
        BGTR    30$                     ;IF GTR, NO
        MOVL    (R2)[R0],UCB$L_DL_PMPR(R5)  ;SAVE PREVIOUS MAP REGISTER
30$:    BBC     #RL_CS_V_CE,UCB$W_DL_CS(R5),40$  ;IF CLR, NO RL ERRORS
        BRW     RETREG                  ;DEVICE ERROR
40$:    BLBC    UCB$B_DL_DPPE(R5),45$   ;IF CLR, NO PURGE ERROR
        BRW     RETREG                  ;PURGE ERROR
;
; RETURN HEADER INFORMATION FOR READ HEADER FUNCTION
;
45$:    CMPB    #CDF_READHEAD,UCB$B_CEX(R5)  ;READ HEADER FUNCTION?
        BNEQ    DATACHECK               ;IF NEQ, NO
        PUSHL   UCB$W_BCR(R5)           ;SAVE NEG BYTES REMAINING
        PUSHL   UCB$L_SVAPTE(R5)        ;SAVE ADDRESS OF PTE
        MOVAB   UCB$W_DL_DB(R5),R1      ;SET ADDRESS OF INTERNAL BUFFER
        MOVL    #6,R2                   ;SET NUMBER OF BYTES TO MOVE
        CMPW    R2,UCB$W_BCNT(R5)       ;ROOM FOR FULL HEADER?
        BLSSU   50$                     ;IF LSSU, YES
        MOVZWL  UCB$W_BCNT(R5),R2       ;SET LENGTH OF PARTIAL HEADER
```

```
50$:    SUBW3   UCB$W_BCNT(R5),R2,UCB$W_BCR(R5)  ;CALCULATE TRANSFER BYTE COUNT
        JSB     G^IOC$MOVTOUSER         ;MOVE HEADER TO USER BUFFER
        POPL    UCB$L_SVAPTE(R5)        ;RESTORE ADDRESS OF PTE
        POPL    UCB$W_BCR(R5)           ;RESTORE NEG BYTES REMAINING

;
; PERFORM DATA CHECK, IF REQUESTED
;
DATACHECK:                              ;DATACHECK AFTER PARTIAL TRANSFER
        BBC     #IO$V_DATACHECK,-       ;IF CLR, DATA CHECK NOT REQUESTED
                UCB$W_FUNC(R5),UPDATE   ;...
        BBSC    #0,UCB$B_DL_DCHEK(R5),- ;IF SET, DATA CHECK ALREADY PERFORMED
                UPDATE                  ;...
        INCB    UCB$B_DL_DCHEK(R5)      ;SET DATA CHECK IN PROGRESS
        MOVZBL  #IO$_WRITECHECK,R3      ;SET CASE INDEX TO WRITE CHECK
        BRW     XFER                    ;BRANCH TO PERFORM WRITE CHECK

;
; UPDATE BUFFER ADDRESS, CURRENT DISK ADDRESS, AND BYTES REMAINING
; FOR NEXT TRANSFER
;
UPDATE:                                 ;UPDATE TRANSFER PARAMETERS
        BBC     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),10$  ;IF BC, NO
        BICB3   #^XCF,UCB$W_DL_CS(R5),- ;SAVE MEMORY EXTENSION BITS
                UCB$B_DL_XBA(R5)        ;...
        MOVW    UCB$W_DL_BA(R5),-       ;UPDATE SAVED BUFFER ADDRESS
                UCB$W_DL_SBA(R5)        ;...
10$:    CLRB    UCB$W_DA(R5)            ;UPDATE DESIRED SECTOR TO ZERO
        ADDL3   #^O100,UCB$W_DL_DA(R5),R1  ;INCREMENT CYLINDER & SURFACE
        EXTZV   #6,#1,R1,R2             ;EXTRACT DESIRED DISK SURFACE
        MOVB    R2,UCB$W_DA+1(R5)       ;UPDATE DESIRED DISK SURFACE
        EXTZV   #7,#9,R1,R2             ;EXTRACT DESIRED DISK CYLINDER
        MOVW    R2,UCB$W_DC(R5)         ;UPDATE DESIRED DISK CYLINDER
        ADDW    UCB$W_DL_PBCR(R5),-     ;UPDATE NEG BYTES REMAINING TO XFER
                UCB$W_BCR(R5)           ;...
        BEQL    RETREG                  ;IF EQL, TRANSFER COMPLETE
        BRW     FDISPATCH               ;MORE BYTES REMAINING - CONTINUE

;
; GET STATUS AND RESET ERRORS
;
RETREG:                                 ;GET STATUS AND RESET ERRORS
;
; FOR A READ OPERATION WHEN NO ADAPTER MAPPING IS PRESENT
; EMPTY INTERNAL BUFFER INTO USER'S BUFFER FOR LAST READ
;
        BSBW    DL_MOVE_TO_BUFFER       ;MOVE LAST READ INTO USER'S BUFFER
        SETIPL  UCB$B_FIPL(R5)          ;MAKE SURE AT FORK IPL (TIMEOUT)
        MOVW    #RL_DA_M_STS!-          ;PUT GET STATUS IN DAR
                RL_DA_M_MRK,RL_DA(R4)   ;...
        CLRL    R2                      ;CLEAR R2 FOR UNIT NUMBER
        INSV    UCB$W_UNIT(R5),#8,#8,R2 ;GET UNIT NUMBER
        BISW3   R2,#F_GETSTATUS,RL_CS(R4)  ;EXECUTE GET STATUS
        BSBW    DL_WAIT                 ;WAIT FOR CONTROLLER
        MOVW    RL_MP(R4),UCB$W_DL_MP(R5)  ;RETRIEVE ERROR REGISTER
        MOVW    #RL_DA_M_RST!-          ;PUT GET STATUS & RESET IN DAR
                RL_DA_M_STS!RL_DA_M_MRK,RL_DA(R4)  ;...
        BISW3   R2,#F_GETSTATUS,RL_CS(R4)  ;EXECUTE RESET
        BSBW    DL_WAIT                 ;WAIT FOR CONTROLLER
```

```
;
; DETERMINE EXIT - SPECIAL CONDITION, FATAL ERROR, RETRIABLE ERROR, OR SUCCESS
;
        CMPZV   #0,#5,UCB$W_DL_MP(R5),- ;HEADS, BRUSHES, STATE OK?
                #RL_MP_M_BH!RL_MP_M_HO!RL_SLM  ;...
        BEQL    1$                      ;IF EQL, YES, ONLINE
        BICW    #UCB$M_TIMOUT,UCB$W_STS(R5)  ;CLEAR DEVICE TIME OUT
        MOVZWL  #SS$_MEDOFL,R0          ;SET MEDIUM OFFLINE STATUS
        BRW     FUNCXT                  ;RETURN
1$:     BITW    #UCB$M_POWER!-          ;POWER FAIL OR DEVICE TIMEOUT?
                UCB$M_TIMOUT,UCB$W_STS(R5)  ;...
        BNEQ    SPECOND                 ;IF NEQ, YES, SPECIAL CONDITION
        BBS     #RL_MP_V_VC,UCB$W_DL_MP(R5),20$  ;IF SET, VOLUME INVALID
        BBS     #RL_CS_V_CE,UCB$W_DL_CS(R5),2$  ;IF SET, RL ERROR
        BLBC    UCB$B_DL_DPPE(R5),10$   ;IF CLR, NO PURGE ERROR
2$:     JSB     G^ERL$DEVICERR          ;ALLOCATE AND FILL ERROR MESSAGE BUFFER
        BBS     #IO$V_INHRETRY,UCB$W_FUNC(R5),20$  ;IF SET, RETRY INHIBITED
        BBS     #RL_CS_V_NXM,UCB$W_DL_CS(R5),20$  ;IF SET, NONEXISTENT MEMORY
        BBC     #RL_CS_V_DE,UCB$W_DL_CS(R5),5$  ;IF CLR, NO DRIVE ERRORS
        BBC     #RL_MP_V_WL,UCB$W_DL_MP(R5),4$  ;IF CLR, NOT WRITE LOCKED
        BBS     #RL_MP_V_WGE,UCB$W_DL_MP(R5),20$  ;IF WL & WGE SET, WL ERROR
4$:     BITW    #RL_MP_M_WDE!-          ;WRITE DATA ERROR, OR
                RL_MP_M_CHE!-           ;CURRENT HEAD ERROR, OR
                RL_MP_M_WGE!-           ;WRITE GATE ERROR, OR
                RL_MP_M_DSE,UCB$W_DL_MP(R5)  ;DRIVE SELECT ERROR?
        BNEQ    20$                     ;IF NEQ, YES
;
; RETRIABLE ERROR EXIT
;
5$:     CVTBL   @UCB$L_DPC(R5),-(SP)    ;GET BRANCH DISPLACEMENT
        ADDL    (SP)+,UCB$L_DPC(R5)     ;CALCULATE RETURN ADDRESS - 1
;
; SUCCESSFUL OPERATION EXIT
;
10$:    INCL    UCB$L_DPC(R5)           ;ADJUST TO CORRECT RETURN ADDRESS
        JMP     @UCB$L_DPC(R5)          ;RETURN TO DRIVER
;
; FATAL ERROR EXIT
;
```

```
20$:    BRW     FATALERR                ;FATAL ERROR EXIT
;
; SPECIAL CONDITION EXIT (POWER FAILURE OR DEVICE TIMEOUT)
;
SPECOND:
        BBS     #UCB$V_POWER,UCB$W_STS(R5),PWRFAIL  ;IF SET, POWER FAILURE
                                        ;IF CLR, DEVICE TIMEOUT
        JSB     G^ERL$DEVICTMO          ;LOG DEVICE TIMEOUT
        BICW    #UCB$M_TIMOUT,UCB$W_STS(R5)  ;CLEAR TIMEOUT STATUS
        MOVZWL  #SS$_TIMEOUT,R0         ;SET DEVICE TIMEOUT STATUS
        DECB    UCB$B_ERTCNT(R5)        ;ANY ERROR RETRIES REMAINING?
        BEQL    RESETXFR                ;IF EQL, NO
        BRW     FDISPATCH               ;RETURN
RESETXFR:                               ;RESET TRANSFER BYTE COUNT
        MOVL    UCB$L_IRP(R5),R3        ;GET ADDRESS OF I/O PACKET
        MNEGW   IRP$W_BCNT(R3),UCB$W_BCR(R5)  ;RESET BYTE COUNT
        BRW     FUNCXT                  ;EXIT
PWRFAIL:                                ;POWER FAILURE
        BICW    #UCB$M_POWER,UCB$W_STS(R5)  ;CLEAR POWER FAILURE BIT
        TSTW    UCB$W_DL_DPN(R5)        ;ARE UCB RESOURCES ALLOCATED?
        BEQL    50$                     ;IF EQL, NO
        BBC     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),50$  ;IF BC, NO
        RELDPR                          ;RELEASE DATA PATH
        RELMPR                          ;RELEASE MAP REGISTERS
50$:    RELCHAN                         ;RELEASE CHANNEL IF OWNED
        MOVL    UCB$L_IRP(R5),R3        ;GET ADDRESS OF I/O PACKET
        MOVQ    IRP$L_SVAPTE(R3),-      ;RESTORE TRANSFER PARAMETERS
                UCB$L_SVAPTE(R5)        ;...
        BRW     PREPROCESS              ;RETURN TO PREPROCESS UCB FIELDS
        .PAGE
        .SBTTL  INTERRUPT SERVICE ROUTINE
;++
; DL$INT - RL11 INTERRUPT SERVICE ROUTINE
;
; FUNCTIONAL DESCRIPTION:
;
;       THIS ROUTINE IS ENTERED VIA A JSB INSTRUCTION WHEN AN INTERRUPT
;       OCCURS ON AN RL11 DISK CONTROLLER. IF THE INTERRUPT IS NOT EXPECTED,
;       THE UNSOLICITED INTERRUPT ROUTINE DISMISSES THE INTERRUPT. IF
;       THE INTERRUPT IS EXPECTED, DEVICE REGISTERS ARE SAVED AND THE
;       DRIVER IS CALLED AT ITS INTERRUPT RETURN ADDRESS. THE DRIVER FORKS,
;       CAUSING A RETURN TO THIS ROUTINE, WHICH RESTORES GENERAL REGISTERS
;       AND DISMISSES THE INTERRUPT.
;
; INPUTS:
;
;       00(SP)  - POINTER TO ADDRESS OF THE 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)  - PC AT THE TIME OF THE INTERRUPT
;       32(SP)  - PSL AT THE TIME OF THE INTERRUPT
;
; OUTPUTS:
;
;       DEVICE REGISTERS ARE SAVED, IPL IS LOWERED TO FORK LEVEL, THE
;       INTERRUPT IS DISMISSED, ALL REGISTERS EXCEPT R0-R5 ARE PRESERVED.
;
;--
```

```
DL_INT::                                ;INTERRUPT SERVICE ROUTINE
        MOVL    @(SP)+,R3               ;REMOVE ADDRESS OF IDB FROM STACK
        MOVQ    (R3),R4                 ;GET ADDRESS OF CSR AND UCB
        TSTL    R5                      ;IS R5 A ZERO
        BEQL    DL_UNSOLNT              ;IF EQL, NO OWNER
        BBCC    #UCB$V_INT,-            ;IF CLR, INTERRUPT NOT EXPECTED
                UCB$W_STS(R5),DL_UNSOLNT  ;...
        CMPB    #CDF_READHEAD,UCB$B_CEX(R5)  ;READ HEADER FUNCTION?
        BNEQ    10$                     ;IF NEQ, NO
        MOVW    RL_MP(R4),UCB$W_DL_DB(R5)    ;SAVE SECTOR HEADER INFORMATION
        MOVW    RL_MP(R4),UCB$W_DL_DB+2(R5)  ;...
        MOVW    RL_MP(R4),UCB$W_DL_DB+4(R5)  ;...
10$:    MOVAB   RL_CS(R4),R2            ;GET ADDRESS OF CONTROL STATUS REGISTER
        MOVAB   UCB$W_DL_CS(R5),R3      ;GET ADDRESS OF REGISTER SAVE AREA
        MOVW    (R2)+,(R3)+             ;SAVE CONTROL STATUS REGISTER
        MOVW    (R2)+,(R3)+             ;SAVE BUFFER ADDRESS REGISTER
        MOVW    (R2)+,(R3)+             ;SAVE DISK ADDRESS REGISTER
        MOVW    (R2)+,(R3)+             ;SAVE MULTIPURPOSE REGISTER
20$:    MOVQ    UCB$L_FR3(R5),R3        ;RESTORE DRIVER CONTEXT
        JSB     @UCB$L_FPC(R5)          ;CALL DRIVER AT INTERRUPT RETURN ADDRESS
DL_UNSOLNT:                             ;UNSOLICITED INTERRUPT
        POPR    #^M<R0,R1,R2,R3,R4,R5>  ;RESTORE R0-R5
        REI                             ;RETURN FROM INTERRUPT
        .PAGE
        .SBTTL  REGISTER DUMP ROUTINE
;++
;
; DL_REGDUMP - REGISTER DUMP ROUTINE
;
; FUNCTIONAL DESCRIPTION:
;
; THIS ROUTINE IS CALLED TO SAVE THE DEVICE REGISTERS AND UBA RESOURCE
; 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 DEVICE CONTROL STATUS REGISTER (CSR)
;       R5      - ADDRESS OF UNIT-CONTROL BLOCK (UCB)
;
; OUTPUTS:
;
;       THE DEVICE AND UBA REGISTERS ARE SAVED IN THE SPECIFIED BUFFER.
;       R0 CONTAINS THE ADDRESS OF THE NEXT EMPTY LONGWORD IN THE BUFFER.
;       ALL REGISTERS EXCEPT R1 AND R2 ARE PRESERVED.
;
;--
DL_REGDUMP:                             ;REGISTER DUMP ROUTINE
        MOVL    #<RL_NUM_REGS+5>,(R0)+  ;INSERT NUMBER OF REGISTERS
        MOVAL   UCB$W_DL_CS(R5),R1      ;GET ADDRESS OF SAVED DEVICE REGISTERS
        MOVZBL  #RL_NUM_REGS,R2         ;GET NUMBER OF DEVICE REGISTERS TO MOVE
10$:    MOVZWL  (R1)+,(R0)+             ;DUMP REGISTER IN BUFFER
        SOBGTR  R2,10$                  ;IF GTR, STILL MORE TO MOVE
        MOVZWL  (R1)+,(R0)+             ;DUMP DATA-PATH NUMBER
        MOVL    (R1)+,(R0)+             ;DUMP DATA-PATH REGISTER
        MOVL    (R1)+,(R0)+             ;DUMP FINAL MAP REGISTER
        MOVL    (R1)+,(R0)+             ;DUMP PREVIOUS MAP REGISTER
        MOVZBL  (R1)+,(R0)+             ;DUMP DATA-PATH-PURGE-ERROR REGISTER
        RSB                             ;RETURN
```

```
        .PAGE
        .SBTTL  MOVE TO USER BUFFER ROUTINE
;++
;
; DL_MOVE_TO_BUFFER - MOVE TO USER BUFFER
;
; FUNCTIONAL DESCRIPTION:
;
; THIS ROUTINE MOVES DATA BETWEEN THE PHYSICALLY CONTIGUOUS BUFFER AND
; THE USER'S BUFFER.
;
; INPUTS:
;
;       R5 - UCB ADDRESS
;
; OUTPUTS:
;
;       DATA MOVE BETWEEN THE PHYSICALLY CONTIGUOUS BUFFER AND THE USER'S BUFFER.
;       REGISTER'S RO,R1, AND R2 ARE DESTROYED
;
;--
DL_MOVE_TO_BUFFER:                      ;BUFFER MOVE ROUTINE
        BBS     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),10$  ;IF BS, YES, NOTHING TO MOVE
        CMPB    #CDF_READDATA,UCB$B_CEX(R5);READ DATA OPERATION?
        BNEQ    10$                     ;IF NEQ, NOT A READ
        BBS     #0,UCB$B_DL_DCHEK(R5),- ;DATA CHECK IN PROGRESS?
                10$                     ;IF BS, YES, NOTHING TO MOVE
        TSTL    UCB$A_DL_MOVRTN(R5)     ;ANYTHING TO MOVE?
        BEQL    20$                     ;IF EQL, NO
        MOVL    UCB$L_DL_BUFADR(R5),RO  ;GET USER BUFFER POINTER
        MOVL    UCB$A_DL_BUF_VA(R5),R1  ;GET PHYSICALLY CONTIGUOUS BUFFER ADDRESS
        MOVZWL  UCB$W_DL_PBCR(R5),R2    ;GET NUMBER OF BYTES TO TRANSFER
        JSB     @UCB$A_DL_MOVRTN(R5)    ;CALL MOVE ROUTINE
        MOVL    RO,UCB$L_DL_BUFADR(R5)  ;SAVE INTERNAL BUFFER POINTER
        MOVAB   G^IOC$MOVTOUSER2,-      ;SET NEXT MOVE ROUTINE TO BE USED
                UCB$A_DL_MOVRTN(R5)     ;
10$:    RSB                             ;RETURN
20$:    MOVAB   G^IOC$MOVTOUSER,-       ;SET NEXT MOVE ROUTINE TO BE USED
                UCB$A_DL_MOVRTN(R5)     ;
        RSB                             ;RETURN
        .PAGE
        .SBTTL  MOVE FROM USER BUFFER ROUTINE
;++
;
; DL_MOVE_FROM_BUFFER - MOVE FROM USER BUFFER
;
; FUNCTIONAL DESCRIPTION:
;
; THIS ROUTINE MOVES DATA BETWEEN THE PHYSICALLY CONTIGUOUS BUFFER AND
; THE USER'S BUFFER.
```

```
;
; INPUTS:
;
;       R5 - UCB ADDRESS
;
; OUTPUTS:
;
;       DATA MOVE BETWEEN THE PHYSICALLY CONTIGUOUS BUFFER AND THE USER'S BUFFER.
;       REGISTER'S R0,R1, AND R2 ARE DESTROYED
;
;--
DL_MOVE_FROM_BUFFER:                    ;BUFFER MOVE ROUTINE
        BBS     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),10$  ;IF BS, YES, NOTHING TO MOVE
        CMPB    #CDF_WRITEDATA,UCB$B_CEX(R5);WRITE DATA OPERATION?
        BNEQ    10$                     ;IF NEQ, NOT A WRITE
        BBS     #0,UCB$B_DL_DCHEK(R5),- ;DATA CHECK IN PROGRESS?
                10$                     ;IF BS, YES, NOTHING TO MOVE
        MOVL    UCB$L_DL_BUFADR(R5),R0  ;GET USER BUFFER POINTER
        MOVL    UCB$A_DL_BUF_VA(R5),R1  ;GET PHYSICALLY CONTIGUOUS BUFFER ADDRESS
        MOVZWL  UCB$W_DL_PBCR(R5),R2    ;GET NUMBER OF BYTES TO TRANSFER
        JSB     @UCB$A_DL_MOVRTN(R5)    ;CALL MOVE ROUTINE
        MOVL    R0,UCB$L_DL_BUFADR(R5)  ;SAVE INTERNAL BUFFER POINTER
        MOVAB   G^IOC$MOVFRUSER2,-      ;SET NEXT MOVE ROUTINE TO BE USED
                UCB$A_DL_MOVRTN(R5)     ;
10$:    RSB                             ;RETURN
DL_END:                                 ;ADDRESS OF LAST LOCATION IN DRIVER
        .END
```

F Sample Driver for the DR11–W and DRV11-WA

The following driver, XADRIVER, controls the DR11–W, a 16-bit parallel DMA interface on Unibus systems. The driver also controls the DRV11-WA, a 16-bit parallel DMA interface on the Q-bus. Operational details of these devices, as well as the capabilities controlled by the driver can be found in the *VAX/VMS I/O User's Reference Manual: Part II*. Specific code changes since VAX/VMS V4.0, including the code added to support the DRV11-WA, are highlighted with change bars in the margin.

You can find a copy of the driver code (XADRIVER.MAR) in SYS$EXAMPLES.

```
        .TITLE  XADRIVER - VAX/VMS DR11-W AND DRV11-WA DRIVER
        .IDENT  'X-5'
;
;*******************************************************************************
;*                                                                             *
;*  COPYRIGHT (c) 1978, 1980, 1982, 1984, 1986                                 *
;*  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 driver for the DR11-W (UNIBUS) and
; DRV11-WA (Q22 bus).  Since the driver was originally written for the
; DR11-W, many inline comments refer to the "DR11-W" and "UNIBUS," but
; apply as well to the DRV11-WA and the Q22 bus. It includes:
;
;               Tables for loading and dispatching
;               Controller-initialization routine
;               FDT routine
;               Start-I/O routine
;               Interrupt-servicing routine
;               Device-specific cancel-I/O routine
;               Error-logging, register-dumping routine
;
; ENVIRONMENT:
;
;       Kernel mode, nonpaged
;
; AUTHOR:
;
;       C. A. Programmer 10-JAN-79
;
;
; MODIFIED BY:
;
;       V04-005 DGB0127         D. G. Programmer        19-Sep-1985
;               Clean up and document MicroVAX II support
;
;       V04-005 DGB0124         D. G. Programmer        25-Jul-1985
;               Add support for the DRV11-WA on MicroVAX II
;
;       V04-003 DGB0112         D. G. Programmer        31-Jan-1985
;               Move the IO$M_RESET bit to a new location so it no
;               longer coincides with the IO$M_INHERLOG bit.
;
;       V04-002 DGB0106         D. G. Programmer        07-Dec-1984
;               Fix synchronization problem which occurs in the
;               cancel routine if an I/O completes while we're trying
;               to cancel it.
;
;       V04-001 JLV0395         J. V. Programmer        06-Sep-1984
;               Add AVL bit to DEVCHAR.
```

```
;
;       V03-006 TMK0001         T. M. Programmer        07-Dec-1983
;               Fix a broken branch.
;
;       V03-005 JLV0304         J. V. Programmer        24-Aug-1983
;               Several bug fixes. All word writes to XA_CSR now have
;               ATTN set so as to prevent lost interrupts. Attention
;               AST list is synchronized at device IPL in DEL_ATTNAST.
;               Correct status is returned on a set mode ast that
;               it returns through EXE$FINISHIO. REQCOM's are always
;               done at FIPL.  Signed division that prevented full size
;               transfers has been fixed.
;
;
;       V03-004 KDM0059         K. D. Programmer        14-Jul-1983
;               Change time-wait loops to use new TIMEDWAIT macro.
;               Add $DEVDEF.
;
;       V03-003 KDM0002         K. D. Programmer        28-Jun-1982
;               Added $DYNDEF, $DCDEF, and $SSDEF.
;--
        .SBTTL  External and local symbol definitions

; External symbols
        $ACBDEF                         ; AST control block
        $CRBDEF                         ; Channel request block
        $DCDEF                          ; Device types
        $DDBDEF                         ; Device data block
        $DEVDEF                         ; Device characteristics
        $DPTDEF                         ; Driver prolog table
        $DYNDEF                         ; Dynamic data structure types
        $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
        $SSDEF                          ; System status codes
        $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+9>/10>            ; Delay N microseconds after RESET
                                        ;  (rounded up to 10 microsec intervals)

; 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's contents
                .BLKL   1
$DEF    UCB$L_XA_FMPR                   ; Final map register's contents
                .BLKL   1
$DEF    UCB$L_XA_PMPR                   ; Previous map register's contents
                .BLKL   1
$DEF    UCB$W_XA_DPRN                   ; Saved data-path register's number
                .BLKW   1               ;   and data-path-parity-error flag
$DEF    UCB$W_XA_BAETMP                 ; Temporary storage of BAE (DRV11-WA
                .BLKW   1               ;   only
$DEF    UCB$W_XA_BAE                    ; Saved BAE register (DRV11-WA only)
                .BLKW   1

; Bit positions for device-dependent status field in UCB
        $VIELD  UCB,0,<-                ; UCB device-specific bit definitions
                <ATTNAST,,M>,-          ; ATTN AST requested
                <UNEXPT,,M>,-           ; Unexpected interrupt received
                <IGNORE_UNEXPT,,M>,-    ; Ignore initial interrupt on DRV11-WA
                >
UCB$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_BAE                          ; Buffer address extension (DRV11-WA)
                        .BLKW   1
$DEF    XA_CSR                          ; Control/status
                        .BLKW   1
; 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 occurred
                <DLT,,M>,-              ; timeout for successive burst transfer
                <PAR,,M>,-              ; Parity error during DATI/P
                <ACLO,,M>,-             ; Power fail on this processor
                <MULTI,,M>,-            ; Multicycle 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_AVL!-                     ; Available
                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=<<15*4>+<<3+5+1>*4>>,-   ; Diagnostic buffer size
                ERLGBF=<<15*4>+<1*4>+<EMB$L_DV_REGSAV>> ; Error log buffer size
```

```
;
; Function dispatch 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"
        CPUDISP <<UV1,3$>,-             ; Branch to handle MicroVAX I
                 <UV2,5$>>,-            ; Branch to handle MicroVAX II
                CONTINUE=YES            ; Else continue for all other processors
        BRB     9$
3$:     BUG_CHECK UNSUPRTCPU,FATAL      ; DRV11-WA not supported on MicroVAX I
5$:     MOVB    #DT$_XA_DRV11WA,-       ; If this is a Q22 bus, then this is
                  UCB$B_DEVTYPE(R0)     ;   a DRV11-WA rather than a DR11-W.
; On the DRV11-WA, the interrupt enable bit normally remains set at all
; times since an interrupt is generated if the bit makes a low-to-high
; transition when there isn't a DMA transfer in progress.  Since the
; device has the IE bit clear at power-up, an interrupt will be generated
; when we set the IE bit.  Therefore, we tell the interrupt service
; routine to ignore the first unexpected interrupt that occurs.
        BBS     #XA_CSR$V_IE,-          ; Branch if IE bit already set
                  XA_CSR(R4),9$
        BBSS    #UCB$V_IGNORE_UNEXPT,-  ; Else interrupt will occur
                  UCB$W_DEVSTS(R0),9$
; If powerfail has occurred and device was active, force device timeout.
; The user can set his own timeout interval for each request.  Timeout
; is forced so a very long timeout period will be short-circuited.
9$:     BBS     #UCB$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$L_SEGVBN = FNCT bits for DR11-W CSR and ODR image
;
;--
XA_READ_WRITE:
; The IO$M_INHERLOG ("inhibit error logging") function modifier was not
; intended to be used by this driver.  However, since the definition for
; the IO$M_RESET modifier used to be the same as that for IO$M_INHERLOG,
; the error logging routines incorrectly used the IO$M_RESET bit to
; determine whether it should log errors.  To solve this problem, the
; definition for IO$M_RESET was changed.  For the sake of old programs, we
; manually move the RESET bit to its new location.
        BBCC    #IO$V_INHERLOG,IRP$W_FUNC(R3),1$
                                        ; Branch if old reset bit not set
        BISW    #IO$M_RESET,IRP$W_FUNC(R3)
                                        ; Set new reset bit
1$:     BLBC    P2(AP),10$              ; 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 users 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            ; Check 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 parameter 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$:    MOVZBL  #SS$_NORMAL,R0          ; Set status
        JMP     G^EXE$FINISHIOC         ; That's all for now (clears R1)

; 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 privilege
        JMP     G^EXE$SETMODE           ; Queue packet to start I/O

; Error, abort IO
45$:    MOVZWL  #SS$_NOPRIV,R0          ; No privilege for operation
50$:    CLRL    R1
        JMP     G^EXE$ABORTIO           ; Abort IO 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 data path.
;
; OUTPUTS:
;
;       CRB is flagged as to which data path 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 data path
        BBC     #XA$V_DATAPATH,UCB$L_DEVDEPEND(R5),2$ ;  Were we right?
        BICB    #VEC$M_PATHLOCK,CRB$L_INTD+VEC$B_DATAPATH(R0) ; Set buffered data path
2$:
        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 (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
        BISW    #XA_CSR$M_ATTN,R0       ; Force ATTN on to prevent lost interrupt
        MOVW    R0,XA_CSR(R4)
        BBC     #XA$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$:
        MOVZWL  UCB$W_BCNT(R5),R0       ; Fetch byte count
        DIVL3   #2,R0,UCB$L_XA_DPR(R5)  ; Make byte count into word count
        ;
        ; Set up UCB$W_CSRTMP used for loading CSR later.
        ;
        MOVZWL  XA_CSR(R4),R0
        BICW    #^C<XA_CSR$M_FNCT>,R0
        BISW    #XA_CSR$M_IE!XA_CSR$M_ATTN,R0   ; Set Interrupt Enable and ATTN
        BBC     S^#IO$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, function-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 transfer
        MOVL    UCB$L_CRB(R5),R2        ; Address of CRB
        INSV    CRB$L_INTD+VEC$W_MAPREG(R2),#9,#9,R1
                                        ; Insert page number
        EXTZV   #16,#2,R1,R2            ; Extract bits 17:16 of bus address
        CMPB    #DT$_DR11W,-            ; If this is a DR11-W,
                  UCB$B_DEVTYPE(R5)
        BEQL    100$                    ;   then branch
        MOVW    R2,UCB$W_XA_BAETMP(R5)  ; Save value of BAE prior to transfer
        CLRL    R2                      ; Clear XBA bits
100$:   ASHL    #XA_CSR$V_XBA,R2,R2     ; Shift extended memory bits for CSR
        BISW    #XA_CSR$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 into directly.  This prevents inadvertently setting
; the EIR-select flag (writing bit 15) if error happens to become true.
```

```
        DSBINT                          ; Disable interrupts (powerfail)
        MNEGW   UCB$L_XA_DPR(R5),XA_WCR(R4)
                                        ; Load negative of transfer count
        MOVW    R1,XA_BAR(R4)           ; Load low 16 bits of bus address
        CMPB    #DT$_DR11W,-            ; If this is a DR11-W,
                  UCB$B_DEVTYPE(R5)
        BEQL    200$                    ;   then branch
        MOVW    UCB$W_XA_BAETMP(R5),-   ; Load high bits of bus address
                  XA_BAE(R4)
200$:   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 bit is set
                                        ;  in device 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 data-path 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$V_DATAPATH,-       ; Get data-path-register number
                #VEC$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 number
        CMPB    #DT$_DR11W,-            ; If this is a DR11-W,
                  UCB$B_DEVTYPE(R5)
        BEQL    300$                    ;   then branch
        MOVZWL  UCB$W_XA_BAE(R5),R1     ; Fetch high bits of map register no.
        BRB     310$
300$:   EXTZV   #4,#2,UCB$W_XA_CSR(R5),R1 ; High bits of map register no.
310$:   INSV    R1,#7,#2,R0             ; Entire map register number
        CMPW    R0,#496                 ; Is map-register number in range?
        BGTR    28$                     ; 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$                     ; If GTR, no
        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?
        CMPW    (SP),#SS$_OPINCOMPL     ; Log the error, unless this is
        BNEQ    37$                     ;   a DRV11-WA running in link mode
        CMPB    #DT$_DR11W,-            ;   and the operation is incomplete,
                  UCB$B_DEVTYPE(R5)     ;   in which case it is an expected
        BEQL    37$                     ;   error and not worth logging.
        BBS     #XA$V_LINK,-            ; ...
                UCB$L_DEVDEPEND(R5),40$ ; ...
37$:    JSB     G^ERL$DEVICERR          ; Log the error.
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 without 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
        CMPB    #DT$_DR11W,-            ; Branch if this is a DR11-W
                  UCB$B_DEVTYPE(R5)
        BEQL    17$
        BBC     #XA_CSR$V_ERROR,-       ; DRV11-WA - check ERROR bit in CSR
                  UCB$W_XA_CSR(R5),20$  ; Branch on success
        BRW     40$                     ; Branch on error
17$:    BITW    #XA_EIR$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$DIAGBUFILL        ; Fill diagnostic buffer if present
        BSBW    DEL_ATTNAST             ; Deliver outstanding ATTN ASTs
        MOVZWL  #SS$_NORMAL,RO          ; Complete success status
22$:    MULW3   #2,UCB$L_XA_DPR(R5),R1  ; Calculate actual bytes transferred
        SUBW3   R1,UCB$W_BCNT(R5),R1    ; From requested number of bytes
        INSV    R1,#16,#16,RO           ; And place in high word of RO
        MOVL    UCB$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 word 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 occurred, assume it
; is for this READ request and return value to user buffer without
; waiting for an interrupt.
        BBCC    #UCB$V_UNEXPT,-
                UCB$W_DEVSTS(R5),32$    ; Branch if no unexpected interrupt
        ENBINT                          ; Enable interrupts
        BRB     37$                     ; continue
32$:
        SETIPL  #IPL$_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$:    CMPB    #DT$_DR11W,-            ; Branch if this is a DR11-W
                  UCB$B_DEVTYPE(R5)
        BEQL    1037$
        BBC     #XA_CSR$V_ERROR,-       ; DRV11-WA - check ERROR bit in CSR
                 UCB$W_XA_CSR(R5),1038$ ; Branch on success
        BRW     40$                     ; Branch on error
1037$:  BITW    #XA_EIR$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
1038$:  BSBW    MOVTOUSER               ; Store two bytes into user buffer
; Send interrupt back to sender.  Acknowledge receipt of last word.
        DSBINT
        MOVW    UCB$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
        BRW     RETURN_STATUS           ; Finish request in common code
; Error detected in word mode transfer
40$:
        BSBW    DEL_ATTNAST             ; Deliver ATTN ASTs
        BSBW    XA_DEV_RESET            ; Error, reset DR11-W
        JSB     G^IOC$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 users buffer
;       Buffer descriptor in UCB is updated.
;
        .ENABL  LSB
```

```
MOVFRUSER:
        MOVAL   -(SP),R1                ; Address of temporary stack location
        MOVZBL  #2,R2                   ; Fetch two bytes
        JSB     G^IOC$MOVFRUSER         ; Call exec routine to do the deed
        MOVL    (SP)+,R1                ; Retreive the bytes
        BRB     20$                     ; Update UCB buffer pointers
;
; MOVTOUSER - Routine to store two bytes into users 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 ASTs, 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$B_FIPL(R5)          ; Lower to FORK IPL
        PURDPR                          ; Purge buffered data path in UBA
        RELMPR                          ; Release UBA map registers
        RELDPR                          ; Release UBA data path
        BRB     10$                     ; Continue

XA_TIME_OUTW:                           ; Timeout for WORD mode transfer
        SETIPL  UCB$B_FIPL(R5)          ; Lower to FORK IPL
10$:    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$DEVICTMO          ; Log device timeout
        BSBW    DEL_ATTNAST             ;   and deliver the ASTs
        BSBW    XA_DEV_RESET            ; Reset controller
        MOVZWL  #SS$_TIMEOUT,R0         ; Assume error status
        BBC     #UCB$V_CANCEL,-
                UCB$W_STS(R5),20$       ; Branch if not cancel
        MOVZWL  #SS$_CANCEL,R0          ; Set status
```

```
20$:    CLRL    R1
        BICW    #UCB$M_ATTNAST!UCB$M_UNEXPT,UCB$W_DEVSTS(R5)
                                        ; Clear unwanted 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 ASTs 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 CSRs 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 ASTs if no interrupt expected and set unexpected
; interrupt flag.
25$:    BBSC    #UCB$V_IGNORE_UNEXPT,-  ; Ignore spurious interrupt -
                  UCB$W_DEVSTS(R5),30$  ;   (DRV11-WA only)
        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$M_FNCT>,R1   ; Clear all uninteresting bits for later
        CMPB    #DT$_XA_DRV11WA,-       ; If this is a DRV11-WA,
                 UCB$B_DEVTYPE(R5)
        BEQL    57$                     ;  then branch
        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
        BRB     59$
57$:    BISW    #XA_CSR$M_IE,R1         ; On the DRV11-WA, if the IE bit makes
                                        ; a 0-->1 transition while READY=1, a
                                        ; spurious interrupt in generated.
                                        ; Therefore, we leave IE high at all times.
59$:    MOVW    R1,XA_CSR(R4)           ; Clear EIR flag and errors
        MOVW    XA_CSR(R4),R1           ; Read CSR back
        BBC     #XA_CSR$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)
        CMPB    #DT$_DR11W,-            ; If this is a DR11-W,
                  UCB$B_DEVTYPE(R5)
        BEQL    70$                     ;   then branch
        MOVW    XA_BAE(R4),UCB$W_XA_BAE(R5) ; Save BAE in UCB
70$:    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 = negated value of channel index
;       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 ASTs 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 ASTs 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$:
        DSBINT  UCB$B_DIPL(R5)          ; Lock out device interrupts
        BBC     #UCB$V_INT,-            ; Branch if I/O not in progress
                  UCB$W_STS(R5),30$
        JSB     G^IOC$CANCELIO          ; Check if transfer going
        BBC     #UCB$V_CANCEL,-
                  UCB$W_STS(R5),30$     ; Branch if not for this process
;
; Force timeout
;
        CLRL    UCB$L_DUETIM(R5)        ; Clear timer
        BISW    #UCB$M_TIM,UCB$W_STS(R5) ; Set timed bit
        BICW    #UCB$M_TIMOUT,-
                UCB$W_STS(R5)           ; Clear timed-out bit
30$:
        ENBINT                          ; Lower to FORK IPL
        RSB                             ; Return

        .PAGE
        .SBTTL  DEL_ATTNAST, Deliver ATTN ASTs
;++
; DEL_ATTNAST, Deliver all outstanding ATTN ASTs
;
; Functional Description:
;
;       This routine is used by the DR11-W driver to deliver all of the
;       outstanding attention ASTs.  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 parameter.
;
; Inputs:
;
;       R5 = UCB of DR11-W unit
;
; Outputs:
;
;       R0,R1,R2 Destroyed
;       R3,R4,R5 Preserved
;--
```

```
DEL_ATTNAST:
        DSBINT  UCB$B_DIPL(R5)          ; Device IPL
        BBCC    #UCB$V_ATTNAST,UCB$W_DEVSTS(R5),30$
                                        ; Any ATTN ASTs 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 parameter
        MOVW    UCB$W_XA_CSR(R1),ACB$L_KAST+4(R5)
                                        ; Store CSR in AST parameter
        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)
                                        ; Rearrange 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$:    ENBINT                          ; Enable interrupts
        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 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 data-path-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-data-path error
;                       0 = no purge-data-path error
;                       1 = parity error when data path was purged
;               BAETMP - The last value written to the BAE by the driver
;                        during a block mode transfer (DRV11-WA only)
;               BAE - The BAE image at the last interrupt (DRV11-WA only)
;
```

```
;       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  #15,(R0)+               ; Fifteen registers are stored
        MOVAB   UCB$W_XA_CSRTMP(R5),R1  ; Get address of saved register images
        MOVZBL  #8,R2                   ; Return eight registers here
10$:    MOVZWL  (R1)+,(R0)+
        SOBGTR  R2,10$                  ; Move them all
        MOVZBL  UCB$W_XA_DPRN(R5),(R0)+ ; Save data-path register number
        MOVZBL  #3,R2                   ; and three more here
20$:    MOVL    (R1)+,(R0)+             ; Move UBA register contents
        SOBGTR  R2,20$
        MOVZBL  UCB$W_XA_DPRN+1(R5),(R0)+ ; Save data-path-parity-error flag
        MOVZWL  UCB$W_XA_BAETMP(R5),(R0)+ ; Save BAE stored prior to transfer
        MOVZWL  UCB$W_XA_BAE(R5),(R0)+  ; Save BAE stored following transfer
        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 controller, and reenables 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
        CMPB    #DT$_DR11W,-            ; If this is a DR11-W,
                  UCB$B_DEVTYPE(R5)
        BEQL    20$                     ;   then branch
        MOVW    #XA_CSR$M_IE,XA_CSR(R4) ; Clear all writeable bits but IE
        BITB    #XA_CSR$M_RDY,XA_CSR(R4); If not READY then no xfer in progress,
        BNEQ    40$                     ; So no need to reset device
        MNEGW   #1,XA_WCR(R4)           ; Tell it only 1 byte left to xfer
        MOVB    #XA_CSR$M_CYCLE/256,-   ; and complete the transfer.
                  XA_CSR+1(R4)
        BRB     30$
20$:    MOVB    #<XA_CSR$M_MAINT/256>,XA_CSR+1(R4)
        CLRB    XA_CSR+1(R4)
; ***  Must delay here depending on reset interval
30$:    TIMEDWAIT TIME=#XA_RESET_DELAY  ; Number of 10 micro-sec intervals to wait
        MOVB    #XA_CSR$M_IE,XA_CSR(R4) ; Reenable device interrupts
        ENBINT                          ; Restore IPL
        POPR    #^M<R0,R1,R2>           ; Restore registers
        RSB
XA_END:                                 ; End of driver label
        .END
```

G MASSBUS Adapter

This appendix describes the data structures and macros used by DIGITAL for its standard magnetic tape and disk products.

The MASSBUS adapter (MBA) is the hardware interface between the backplane interconnect and MASSBUS storage devices. The MASSBUS is the communication path linking the MASSBUS adapter to the mass storage devices.

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 between main memory and the MASSBUS
- Transfer of interrupts from MASSBUS devices to the backplane interconnect

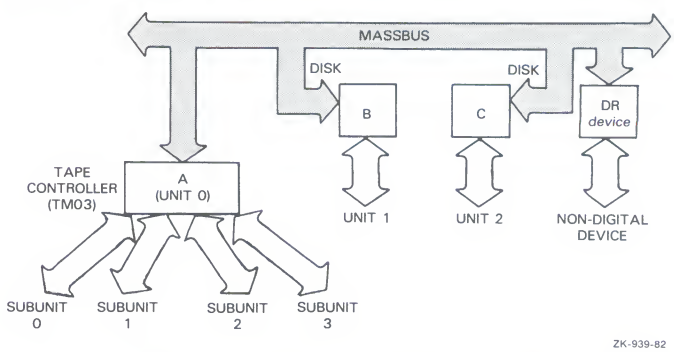
A MASSBUS adapter supports any combination of up to eight device controllers. Typical MASSBUS controllers include the TM03 tape controller and the RP06, RM03, and RM80 disk controllers. Only one controller can transfer data over the MASSBUS at a time.

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 database 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 a controller for several devices, the unit number is distinct from the subunit numbers assigned to the individual drives connected to the controller.

Figure G–1 illustrates a possible MASSBUS configuration.

Figure G–1 MASSBUS Configuration



G.1 MASSBUS Adapter Registers

The MASSBUS adapter has three sets of registers:

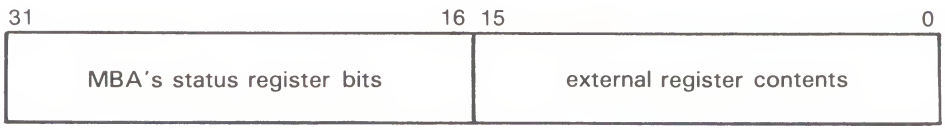
- The MASSBUS adapter's registers
- External registers for each device (controller) on the MASSBUS
- 256 mapping registers

To allow competing devices to share these resources, access to and modification of all MASSBUS adapter registers (internal, external, and mapping registers) are governed by certain rules and conventions. In particular, access to registers might, at times, require ownership of either the device controller or the MASSBUS adapter itself, or both. Subsequent sections in this chapter 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 might 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's status register (one of the MBA's internal registers) to the contents of the specified external register. Figure G–2 illustrates the resulting longword.

Figure G–2 MASSBUS External-Register Longword



ZK-1796-84

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 mapping 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's internal or mapping registers.

Bits 21 through 30 of each of the MBA's mapping registers are reserved; they cannot be written. Use of the MBA's mapping registers is analogous to use of the UNIBUS adapter's mapping registers with the following exceptions:

- Because the MASSBUS can handle only one transfer at a time, ownership of the MASSBUS adapter implies ownership of all its mapping registers. Thus, the driver need not independently request mapping registers.

- The MBA's mapping 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 at the start of a data transfer. Use of the MBA$L_VAR register is described below.
- The MBA's mapping 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 allocate the data path independently.

G.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's mapping registers and associated internal registers. The LOADMBA macro invokes the subroutine IOC$LOADMBAMAP, which performs the following steps:

- Determines the number of mapping 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 mapping registers, beginning with mapping register 0, with the contents of the page-table entries to which UCB$L_SVAPTE points. This step maps the data area for the transfer into the low portion of the MBA's virtual address space. The routine also loads the next mapping register beyond the number used to map the data area with zeros (an invalid map entry). This procedure stops the transfer should a hardware failure occur.
- Loads the MBA$L_VAR register with the zero extended contents of UCB$W_BOFF. Because the first byte of the data area is located at offset UCB$W_BOFF within the page of memory mapped by mapping register 0, the 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's byte-count register (MBA$L_BCR).

Note that if a driver is to perform a data transfer in the reverse direction (for example, read reverse on a tape), it must modify the contents of the MBA$L_VAR, as established by IOC$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$W_BCNT to the contents of the MBA$L_VAR register.

During the progress of a data transfer over the MASSBUS, the MBA$L_VAR register is continuously updated so that it points to the current position in the data area. The *VAX Hardware Handbook* illustrates the mapping of the contents of the MBA$L_VAR register into physical memory.

G.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 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 G–1.

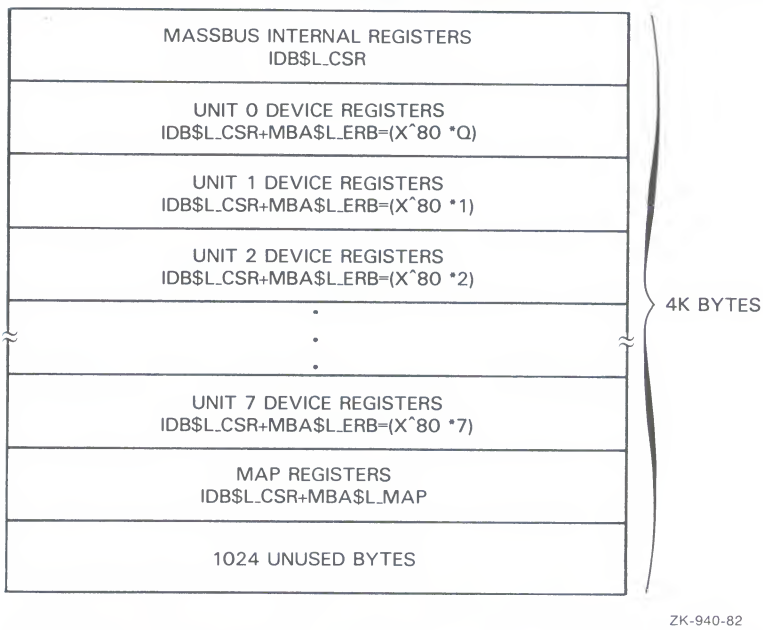
Table G–1 Major Offsets Defined by $MBADEF

Symbol	MBA Register Name	Hex Offset
MBA$L_CSR	Configuration register	0
MBA$L_CR	Control register	4
MBA$L_SR	Status register	8
MBA$L_VAR	Virtual-address register	C
MBA$L_BCR	Byte-count register	10
MBA$L_DR	Diagnostic register	14
MBA$L_SMR	Selected mapping register	18
MBA$L_CAR	Command-address register	1C
MBA$L_ERB	External register base	400
MBA$L_AS	Attention-summary register	414
MBA$L_MAP	Base of mapping registers	800

The MASSBUS adapter's internal registers occupy the low order 1024 bytes of address space even though there are only eight internal MBA registers. Beyond the internal registers, there are eight blocks of 32 longwords (128 bytes) each, 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 controller. Beyond the device-register space is the area reserved for the MASSBUS adapter's 256 mapping registers.

Figure G–3 illustrates the relative positions of the MASSBUS adapter's registers and the values device drivers use to gain access to them. The base address of the MASSBUS adapter's address space, stored in IDB$L_CSR, is the address of the first of the MASSBUS adapter's internal registers. IDB$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's register space occupies only the first 3K out of the 8K allotted to physical I/O address space. However, by convention, VAX/VMS allocates 4K of virtual addresses to each MASSBUS adapter.

Figure G–3 Location of MASSBUS Registers in Physical Address Space



To address a mapping register in the MASSBUS adapter, the driver constructs the following address:

IDB$L_CSR + MBA$L_MAP + mapping-register-index

To address a device register, the driver constructs the following address:

IDB$L_CSR + MBA$L_ERB + (unit-number * 80_{16}) + 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's starting virtual address (the contents of IDB$L_CSR), MBA$L_ERB, the unit number of the device controller multiplied by 80_{16}, and the offset of the specified register.

The attention-summary register (MBA$L_AS), as shown in Table G–1, appears to reside within the external-register space reserved for MASSBUS adapter controller 0. Actually, the attention-summary register is a composite register. Each of the MASSBUS adapter's controllers contributes one bit of information to the register. This composite register appears in each of the eight device register spaces at offset 10_{16} from the base of the device registers for that device. Thus, MBA$L_AS can be defined as either 410_{16}, 490_{16}, 510_{16}, 590_{16}, and so on. For convenience, it has been defined as 410_{16}.

G.1.3 Modifying MASSBUS Adapter Registers

The driver for a MASSBUS device must obtain ownership of the MBA before modifying any of the MBA's internal registers or mapping 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 dedicated controllers, invoke the REQPCHAN macro. Because 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 can share the controller, and so must contend for its use. The controller for several devices 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.

G.2 I/O Database for MASSBUS Devices

During initialization, the system creates an ADP, a CRB, and an IDB 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 dedicated controller or the controller of several devices.

The system builds a unit-control block (UCB) for each single unit controller. Figure G–4 illustrates the I/O database for a MASSBUS adapter with one dedicated 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 dedicated 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.

A controller of several devices, however, requires 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 G–5 shows the I/O database created for a MASSBUS adapter with one disk unit and two tape units.

Figure G–5 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 dedicated controllers, the IDB longword points to the device's UCB, whereas, for a controller for several devices, the longword (or longwords) points to a field within the CRB associated with the 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 G–5 is how multiunit IDBs also maintain an array of longwords. Each longword points to the individual UCBs for the units attached to the controller. Figure G–6 illustrates in more detail the set of I/O data structures for the MASSBUS adapter and its devices.

Figure G–4 I/O Database for MASSBUS Disk Unit

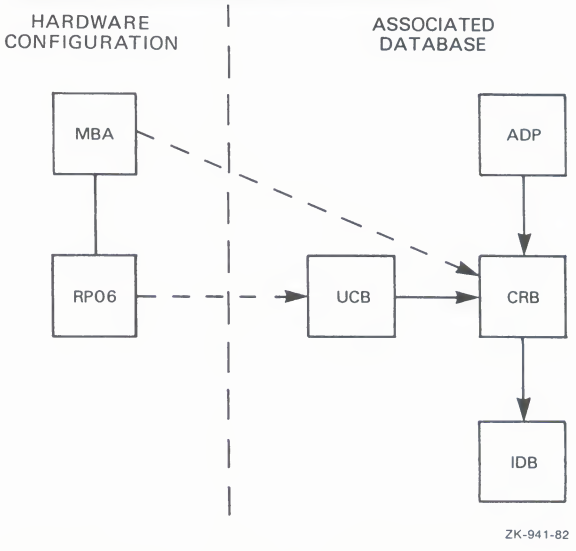
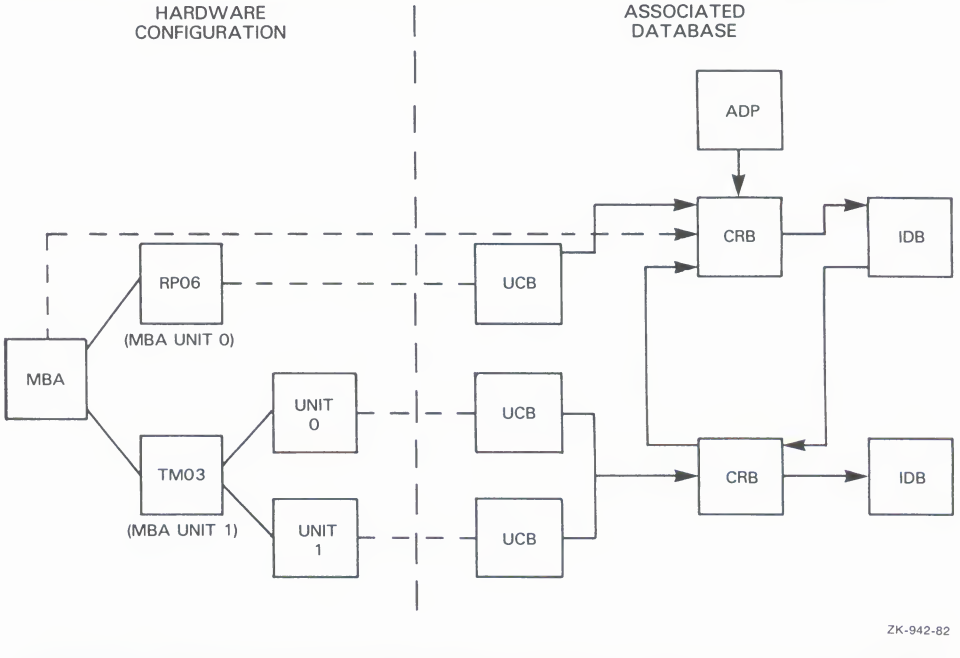


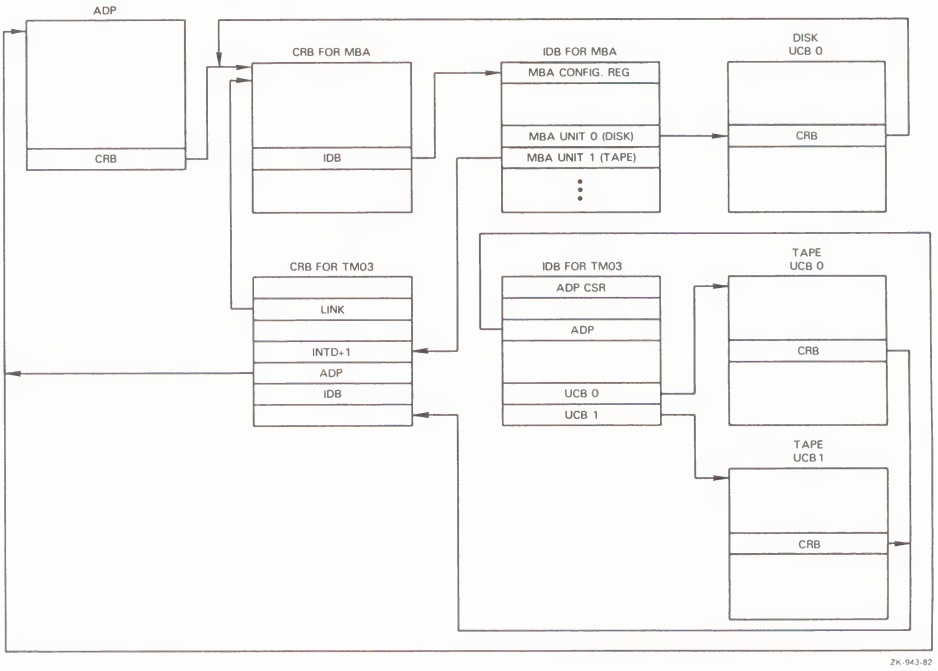
Figure G–5 I/O Database for MASSBUS Disk and Tape Units



G.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.

Figure G–6 I/O Data Structures Used in Dispatching an Interrupt



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 G.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's interrupt dispatcher determines whether the interrupt is for a data transfer or an attention condition.

Data-transfer interrupts occur when a data transfer either completes or is aborted. When the interrupt occurs, the MBA's status register (MBA$L_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 coming 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 can 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 attention-summary register but suppresses the attention interrupt. The interrupt generated when the data transfer is completed allows the MASSBUS adapter's interrupt dispatcher to gain control, handle the data-transfer interrupt, check the attention-summary register and then invoke the proper driver to handle the interrupt.

G.4 MASSBUS Adapter's Interrupt Dispatching

When interrupts occur on the MASSBUS adapter, the MASSBUS adapter's 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.

G.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$V_INT in UCB$L_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 0.5 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 initiate an interrupt. We will assume that disk A's IO$_SEEK is completed first. The operation sets disk A's bit in the attention-summary register and generates the MASSBUS adapter's interrupt.

The MASSBUS adapter's 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's 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.

G.4.2 Dispatching a Device Interrupt

Once the MASSBUS adapter's 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's status register (MBA$L_SR) before dispatching the interrupt to the driver. The status register contains information used only in data-transfer interrupt dispatching.

How the interrupt dispatcher dispatches the interrupt to the driver differs depending on the type of controller.

The MASSBUS adapter's interrupt dispatcher handles a solicited interrupt on a dedicated controller by transferring control to the driver through the fork block in the UCB. On unsolicited interrupts on dedicated controllers, the interrupt dispatcher calls the driver's unsolicited-interrupt-servicing routine.

On dedicated controllers, the MASSBUS adapter's interrupt dispatcher always clears the attention bit in the attention-summary register before it calls back the driver after an interrupt.

Dispatching interrupts to the driver of a device that shares its controller with several other devices differs in two ways from dispatching interrupts to the driver of a device with a dedicated controller.

First, the interrupt dispatcher never clears the attention bit. This task is left to the driver because some controllers that control more than one device use this bit to synchronize their activities, and guarantee the integrity of device registers only while the bit is set. If the interrupt dispatcher clears the bit before returning control to the driver, the driver can no longer rely on the contents of the device's registers.

Second, a controller that controls several devices needs another interrupt dispatcher to handle simultaneous requests from its several subunits. This second-level interrupt dispatcher resides in the driver. After an interrupt, the MASSBUS adapter's interrupt dispatcher indirectly calls this second, driver's interrupt dispatcher using code in the controller's CRB. The driver-loading procedure installs this code when it establishes the I/O database.

G.5 Special Considerations for MASSBUS Device Drivers

MASSBUS adapter considerations affect a driver's device unit initialization routine, start-I/O routines and, for multiunit controllers only, the driver's use of the DPTAB macro. MBA considerations also affect interrupt handling, as described in Section G.4.2. The next sections in this chapter discuss programming details for writing a MASSBUS device driver.

G.5.1 Unit-Initialization Routine

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, which 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's external registers to this controller's device registers. The value of this longword offset is always 32 times the unit number of the controller.

Initialization of a device attached to a dedicated 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 controller that controls several devices, 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 controller's device registers

R4 Address of the MBA's configuration register

R5 Address of device's 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's external registers to the start of the device's device registers. The difference, when divided by 128, is the controller's MBA position number.

G.5.2 The MASSBUS Adapter and the I/O Database

The UCB of a device connected to a single-unit controller, at offset UCB$L_CRB, contains the address of the MASSBUS adapter's CRB. This CRB in turn contains, at offset CRB$L_INTD+VEC$L_IDB, the address of the MASSBUS IDB. This IDB points to the base address of the MASSBUS adapter registers at offset IDB$L_CSR.

A controller that controls several devices maintains a more complicated I/O database. The device UCB, at offset UCB$L_CRB, points to the controller's CRB, and this structure points to the CRB for the MASSBUS adapter at offset CRB$L_LINK. Also, the controller's CRB points to its own IDB at offset CRB$L_INTD+VEC$L_IDB. This IDB points to the controller's device registers at offset IDB$L_CSR.

Thus, the UCB for a device always points to that device's primary CRB, whether it is the MASSBUS adapter's CRB or the controller's CRB. The primary CRB points to the secondary CRB, if one exists for the device.

Figure G–6 shows these relationships among I/O data structures.

G.5.3 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 G.5.3.1
- Clears errors on the MASSBUS adapter by placing the value –1 into the MBA's status register; this is a write-ones-to-clear register (MASSBUS device registers and the MBA's registers are all longwords)
- Invokes the LOADMBA macro to load the MBA's mapping registers as described in Section G.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 G.5.3.3
- Finishes the request like other drivers

G.5.3.1 Requesting Controller Data Channels

Device drivers for MASSBUS devices must request and receive ownership of the MASSBUS adapter channel before loading the MBA's internal registers or mapping 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 can be shared among the units connected to the controller.

Drivers for dedicated controllers must request ownership of the MASSBUS adapter channel by invoking the macro REQPCHAN.

Device drivers for controllers that control several devices invoke the REQPCHAN macro when the operation requires ownership of only the primary channel (the controller's 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.

G.5.3.2 Loading Mapping Registers

MASSBUS device drivers invoke the LOADMBA macro before they initiate a data transfer to load the MBA's mapping registers, the MBA's virtual-address register (MBA$L_VAR), and the MBA's byte-count register (MBA$L_BCR). Drivers cannot modify these registers during a transfer. The LOADMBA macro expects the following register contents:

- The address of the MBA's configuration register (MBA$L_CSR) 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:

1. 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
2. Beginning with the page-table entry to which UCB$L_SVAPTE points and continuing for the number of page-table entries determined in the step above, copies the page-frame numbers from the page-table entries to the corresponding mapping registers, starting at mapping register 0
3. Deposits an invalid value into the mapping register that immediately follows the last mapping register loaded with a PFN so that a hardware fault does not modify memory
4. Moves the negative value of the transfer byte count (UCB$W_BCNT) into the MBA's byte-count register (MBA$L_BCR)
5. Moves the byte offset in the first page of the transfer (UCB$W_BOFF) into the MBA's virtual-address register (MBA$L_VAR)
6. 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 operation (a read-reverse on tape), the driver must modify the contents of the MBA's virtual-address register set up by LOADMBA. Because reverse operations access the I/O buffer from its highest address through its lowest address, the value to be loaded into the MBA's virtual-address register must be the virtual address, in MBA's virtual memory, of the last byte of the buffer. This number is equal to one less than the sum of the contents of UCB$W_BOFF and UCB$W_BCNT.

G.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.

G.5.4 DPTAB Macro

The device driver for a MASSBUS device that shares its controller with other devices 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 CRB and an IDB for the controller.

G.6 Interrupt-Servicing Routines for MASSBUS Devices

The VAX 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 can request attention simultaneously, some device drivers might 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's status register at the time of the interrupt so that the driver can have access to this value.

For I/O operations that involve no data transfer, MBA$INT clears this register before invoking the driver. MBA$INT only preserves the contents of registers R2 through R5. Drivers that use other registers must save the contents of those registers, and must restore them before exiting the interrupt-servicing routine.

G.6.1 Transferring Control to the Interrupt-Servicing Routine

The method by which MBA$INT invokes a driver depends upon whether the driver serves a device connected to a dedicated controller or a device that shares its controller with several other devices. Furthermore, if the device is connected to a dedicated controller, the method of transfer from MBA$INT to the driver depends upon whether or not the interrupt is expected.

For a device on dedicated controller when the driver is expecting an interrupt, 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 instruction.

For a device on a dedicated controller when the driver is not expecting interrupts, MBA$INT obtains the address of the driver's unsolicited-interrupt routine from the driver-dispatch table and calls the routine.

For a device that shares its controller with several other devices, MBA$INT transfers control to the driver's interrupt-servicing routine by simulating a direct transfer, through an interrupt vector, to the controller's CRB. The CRB contains code that transfers control to the interrupt-servicing routine.

MBA$INT first pushes the processor status longword (PSL) onto the stack. The routine then calls (with a JSB instruction that leaves an address within MBA$INT on the stack) the code within the CRB. This code contains the following sequence of instructions, where XX$INT is the address of the interrupt-servicing routine and XX$IDB is the address of the controller's IDB:

```
PUSHR     #^M<R2,R3,R4,R5>
JSB       XX$INT
.LONG     XX$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 an address in the IDB.

G.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 dedicated controller devices return to MBA$INT through an RSB instruction, although the RSB can execute as a result of the driver's invoking the IOFORK macro.

Drivers of devices that share a controller return control to MBA$INT by removing the indirect pointer to the IDB from the top of the stack, restoring registers R2 through R5, and executing an REI instruction. This sequence, executed within the driver's interrupt-servicing routine, eliminates the simulated interrupt-frame from the stack before returning to MBA$INT.

G.6.3 Considerations for Interrupt-Servicing Routines

Drivers for dedicated controller devices attached to the MASSBUS do not have interrupt-servicing routines. Instead, MBA$INT handles all the functions that a driver interrupt-servicing routine normally provides.

Drivers of devices that share a controller on the MASSBUS must have their own interrupt-servicing routines. In general, these routines perform the same functions as the interrupt-servicing routines for UNIBUS and Q22 bus devices (discussed in Section 11. However, the two types of drivers diverge in two areas.

One difference between UNIBUS/Q22 bus and MASSBUS drivers concerns the number of registers saved by the interrupt-servicing routine. When the interrupt dispatcher transfers control to a MASSBUS driver interrupt-servicing routine, registers R2 through R5 are pushed onto the stack. UNIBUS/Q22 bus drivers save R0 through R5.

After handling an interrupt, both MASSBUS and UNIBUS/Q22 bus driver interrupt-servicing routines execute an REI instruction. For UNIBUS/Q22 bus devices, the REI dismisses a real interrupt, whereas the MASSBUS driver's REI returns control to MBA$INT.

H Mapping I/O Space and Connecting to an Interrupt Vector

A real-time VAX/VMS process running on a VAX 8600, VAX 8650, VAX–11/785, VAX–11/782, VAX–11/780, VAX–11/750, VAX–11/730, VAX–11/725, MicroVAX II, or MicroVAX I system can bypass most of the I/O subsystem by manipulating device registers and responding to device interrupts directly.[1]

Programs written in VAX MACRO can interface with the I/O system by using VAX RMS, by using the Queue I/O Request ($QIO) system service, or by mapping to I/O space and connecting to a device interrupt vector. Programs written in a high-level language can interface with the I/O subsystem using the same methods as a VAX MACRO program, or they can issue the I/O statements specific to that language. In the latter case, the program interfaces with the I/O subsystem by means of the VAX Common Run-Time Procedure Library.

A user program can interface with the I/O subsystem at one of several levels, depending on its requirements. At each level, the user program makes trade-offs between ease of use and execution speed. As a general rule, the closer to the VAX/VMS executive that a user program interfaces, the less overhead is involved in the I/O operation. The connect-to-interrupt capability offers the least overhead.

H.1 Interrupt-Generated I/O

A process with suitable privileges can connect to a device interrupt vector and/or map the processor's I/O space into process virtual address space. Connecting to a device interrupt vector allows your process to respond to interrupts from the device with minimal overhead. Mapping processor I/O space allows your process to access device registers from the main program or from an AST service routine.

A process normally uses these features for devices that do not have VAX/VMS drivers. These devices must not be direct memory access (DMA) devices, and they must be attached to the UNIBUS or Q22 bus. Examples of such devices are the AD11–K and the KW11–P.

You can use the $QIO system service with an appropriate function code to connect to a device interrupt vector and to specify a user-supplied interrupt-servicing routine that VAX/VMS executes when the designated device interrupts. Connecting to a device interrupt vector allows you to do the following:

- Respond to an interrupt within a very short time
- Preempt other system processing to handle a real-time event, for example, a clock interrupt
- Buffer data from a device in real time and return the data to the process at a later time

[1] The VAX 8800 and VAX 8200 systems do not support the connect-to-interrupt driver facility discussed in this appendix.

- Set an event flag or queue an AST to your process after receiving the interrupt

An interrupt-servicing routine, specified in your process, allows it to perform some of the functions normally performed by a device driver. The connect-to-interrupt facility, with its VAX/VMS-supplied driver (CONINTERR), thus allows you to avoid writing a full device driver and loading it into the operating system.

If you must access device registers from user mode (that is, from the main program or a user-mode AST service routine), you must use the Create and Map Section ($CRMPSC) system service to map I/O space, specifying page frame number (PFN) mapping. The service creates a global or private section that maps the specified I/O pages into your process' virtual address space. The process can then gain access to I/O space using virtual addresses.

You do not need to map I/O space to access device registers from any of the following routines specified in the $QIO call connecting to an interrupt vector:

- Unit-initialization routine
- Start-I/O routine
- Interrupt-servicing routine
- Cancel-I/O routine

These routines execute in system space and thus can access UNIBUS or Q22 bus I/O space, which is mapped as part of system space.

The remainder of this appendix explains how to map the VAX processor's I/O space and how to connect to a device interrupt vector.

H.2 I/O Space

On a VAX processor, I/O space is assigned physical address locations of 20000000_{16} and higher. I/O space contains device registers that a driver or user process can read and write to control a device. Each device controller has an associated control and status register (CSR) in I/O space. Device registers for each device are located at an offset from the device's CSR.

Macros of the format $IO*xxx*DEF (where *xxx* represents a specific VAX processor), contained in SYS$LIBRARY:LIB.MLB, define symbols describing the layout of I/O space. Table H–1 describes these macros and the symbols they define for each VAX processor.

Table H–1 Symbols Defined by the $IO*xxx*DEF Macros

Macro	Processor(s)	Symbol(s)	Meaning	Value (hex)
$IO790DEF	VAX 8600	IO790$AL_IOA0	Start of I/O space for SBI0	20000000
	VAX 8650	IO790$AL_IOA1	Start of I/O space for SBI1	22000000
		IO790$AL_UB0SP	Offset to start of address space for first UNIBUS	24000000 100000
$IO780DEF		IO780$AL_IOBASE	Start of I/O space	20000000
	VAX–11/782 VAX–11/785	IO780$AL_UB0SP	Start of address space for first UNIBUS	20100000
$IO750DEF[1]	VAX–11/750	IO750$AL_IOBASE	Start of I/O space	F20000
		IO750$AL_UBBASE	Start of UBA0 register space	F30000
		IO750$AL_MBBASE	Start of MBA0 register space	F28000
		IO750$AL_UB0SP	Start of address space for first UNIBUS	FC0000
$IO730DEF	VAX–11/730	IO730$AL_IOBASE	Start of I/O space	F20000
	VAX–11/725	IO730$AL_UB0SP	Start of address space for UNIBUS	FC0000
$IOUV2DEF	MicroVAX II	IOUV2$AL_QB0SP	Start of address space for Q22 bus	20000000
$IOUV1DEF	MicroVAX I	IOUV1$AL_QB0SP	Start of address space for Q22 bus	20000000

[1]The VAX–11/750 processor has fixed MASSBUS adapters (UBBASE, MBBASE) in contrast to the VAX–11/780 processor, which has floating MASSBUS adapters, and the VAX–11/730, which does not have MASSBUS adapters.

The number of registers and their locations vary from device to device. The *PDP–11 Peripherals Handbook* provides the necessary information for DIGITAL-supplied devices. The *VAX Hardware Handbook* contains information about the layout of I/O space.

From the symbols defined by the macros described in Table H–1, you can derive the starting physical addresses of UNIBUS or Q22 bus space for the various VAX processors. Table H–2 lists the starting physical addresses for UNIBUS adapters on the VAX–11/780, VAX–11/782, VAX–11/785, VAX–11/750, VAX–11/730, and VAX–11/725 processors, as well as the starting physical addresses for MicroVAX I and MicroVAX II Q22 bus space.

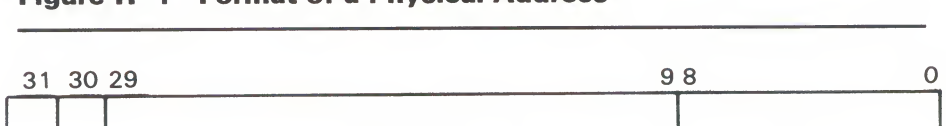
Table H–2 UNIBUS and Q22 Bus Addresses for VAX Processors

UNIBUS adapter number	VAX–11/725 VAX–11/730	VAX–11/750	VAX–11/780 VAX–11/782 VAX–11/785	MicroVAX I MicroVAX II	VAX 8600 SBI0/SBI1 VAX 8650 SBI0/SBI1[1]
0	00FC0000	00FC0000	20100000	20000000	20100000/22100000
1	–	00F80000	20140000	–	20140000/22140000
2	–	–	20180000	–	20180000/22180000
3	–	–	201C0000	–	201C0000/221C0000

[1]The maximum number of UBAs that either a VAX 8650 or VAX 8600 can have is seven. Each SBI can have up to four UBAs, but only one of the SBIs can be expanded into another cabinet. If you expand SBI1, you can put four UBAs on it, plus three on the internal slots of SBI0. Thus, the maximum number of UBAs in this configuration is seven. If you expand SBI0, you can have four UBAs on it (three internal, one external). Because you cannot expand SBI1 in this case, the maximum number of UBAs in this configuration is four.

The *page frame number* (PFN) of a physical page in memory is contained in bits 9 through 29 of its physical address (see Figure H–1). Bit 29 of the address is clear to indicate a physical memory address and set to indicate an address in I/O space. Bits 0 through 8 specify the byte address within the page.

Figure H–1 Format of a Physical Address



ZK-4845-85

H.3 PFN Mapping

For a process to gain access to I/O space or to any page of physical memory, it must map that page into its virtual address space. When your VAX/VMS process maps a page by specifying its page frame number, it completely bypasses VAX/VMS memory management and creates its own window to the page. As a result, the protection functions that VAX/VMS normally performs are not performed for PFN mapping:

- No checks are performed to ensure that no other VAX/VMS processes are mapped to the page and modifying it.
- No reference count is maintained. A process can delete a global section mapped by page frame numbers when other processes are still using it; this is not the case for other types of global sections.

Modifying pages mapped by page frame numbers can have unpredictable results and can adversely affect system operation, especially if the operating system is also using these pages. Because of the unprotected nature of PFN-mapped pages, you must have the PFNMAP privilege to use this capability.

When used for mapping by page frame number, the Create and Map Section ($CRMPSC) system service designates the specified page(s) as a global or private section and maps the section into the requesting process' virtual address space. The pages can be located anywhere in the VAX processor's local memory, in MA780 memory (if a multiport memory unit is connected to the system), or in I/O space.

The format and conventions PFN mapping (that is, mapping a physical page frame section) are similar to those for mapping a disk file section. The $CRMPSC system service has the following general formats:

VAX MACRO Format

```
$CRMPSC  [inadr] [,retadr] [,acmode] [,flags] [,gsdnam] [,ident] -
         [,relpag] [,chan] [,pagcnt] [,vbn] [,prot] [,pfc]
```

High-Level Language Format

```
SYS$CRMPSC([inadr] [,retadr] [,acmode] [,flags] [,gsdnam] [,ident]
          [,relpag] [,chan] [,pagcnt] [,vbn] [,prot] [,pfc])
```

The **relpag**, **chan**, and **pfc** arguments are not applicable to mapping by page frame number. The **inadr**, **retadr**, **acmode**, **gsdnam**, **ident**, and **prot** arguments have the same functions, regardless of whether you specify page frame number mapping; these arguments are described in the *VAX/VMS System Services Reference Manual*.

The following arguments can have values specific to PFN mapping:

Arguments

[flags]

Mask defining the section type and characteristics. This mask is the logical OR of the flag bits you want to set. The $SECDEF macro defines symbolic names for the flag bits in the mask. The SEC$M_PFNMAP flag bit must be set to indicate mapping by page frame number. The SEC$M_PFNMAP flag setting identifies the memory for the section as starting at the page frame number specified in the **vbn** argument and extending for the number of pages specified in the **pagcnt** argument.

If appropriate, the following flags can also be set:

Flag	Description
SEC$M_GBL	Pages form a global section. The default is private section.
SEC$M_EXPREG	Pages are mapped into the first available space. By default, pages are mapped into the range specified by the **inadr** argument.
SEC$M_WRT	Pages form a read/write section. By default, pages form a read-only section.
SEC$M_PERM	Pages are permanent. By default, pages are temporary.
SEC$M_SYSGBL	Pages form a system global section. By default, pages form a group global section.

Neither the SEC$M_CRF (copy-on-reference) nor the SEC$M_DZRO (demand-zero) bit can be set when mapping by page frame number.

[pagcnt]

Number of pages in the section; the value of this argument must not be zero.

[vbn]

Page frame number of the first page to be mapped (as opposed to this argument's normal usage identifying the starting virtual block number (vbn) within a disk file). When you are mapping more than one page with a single $CRMPSC system service request, the pages are physically contiguous starting with the specified page.

H.3.1 Notes on PFN Mapping

The following considerations apply to PFN mapping.

1. An error in mapping UNIBUS or Q22 bus I/O space or a reference to a nonexistent bus address causes a UNIBUS adapter error. However, this error does not cause a system failure (except on VAX–11/750 and VAX–11/730 processors where a machine check will occur). Rather, an entry is made in the system error log file and the user program continues executing (probably with erroneous results). The process is not notified of the UNIBUS adapter error.
2. If a power failure occurs on the UNIBUS or Q22 bus, the system continues to run. However, if a user process accesses UNIBUS or Q22 bus I/O space from user mode during a bus power failure, the process receives a machine check exception. To handle this condition, the process must have a condition handler to deal with machine check exceptions. The *VAX/VMS System Services Reference Manual* discusses condition handlers in detail.
3. During recovery from a UNIBUS adapter or Q22 bus power failure, the processor spends a considerable amount of time (perhaps 10 to 60 milliseconds) at IPL 31. This action blocks user processes from executing during the recovery.
4. When a process requests deletion of a PFN-mapped page, VAX/VMS will wait until there is no direct I/O outstanding for the process before deleting the page. This is because no reference count is maintained for PFN-mapped pages. (For example, VAX/VMS cannot determine whether outstanding direct I/O is for the PFN-mapped page or not.) Applications using devices that have direct I/O perpetually outstanding, such as the DR32, must not delete PFN-mapped pages because this will cause the process to hang in the MWAIT state.

Once you have mapped to I/O space, you can read data from a device data buffer register or enable interrupts by setting a bit in the CSR, because the device registers are now addressable as part of your process' virtual memory. The UNIBUS adapter performs the actual mapping of VAX virtual addresses to 18-bit UNIBUS addresses that correspond to device registers. Likewise, the MicroVAX II or MicroVAX I processor performs the mapping of virtual addresses to 22-bit Q22 bus addresses that correspond top device registers.

See Section 6.2 for a list of restrictions that apply to the use of device register space.

H.4 Connecting to an Interrupt Vector

On a VAX processor, peripheral devices are associated with interrupt vectors. When a device interrupt occurs, the action taken by the processor depends on the device's associated IPL.

Connecting to an interrupt vector differs from the standard method of programming a peripheral device. Programming a peripheral device is normally a 3-step loop:

1. The device driver starts the device and enables interrupts from the device.
2. The device generates an interrupt.

3 The device driver services the interrupt, collects status and data, and clears the interrupt condition.

Under the VAX/VMS operating system, a user program normally requests I/O by means of a $QIO system service call. A device driver, executing as part of the operating system, controls and responds to the device. The driver returns status and data to the requesting user process.

However, real-time application programmers can connect to an interrupt vector to control and respond to a device without writing a full VAX/VMS device driver, and without issuing $QIO calls for each device interaction. Instead, you issue a connect-to-interrupt $QIO call that specifies code to be executed to control the device, and a data area that the program and the device control code can share. You subsequently control and respond to the device without additional $QIO calls.

The timings involved in different system activities associated with connecting to an interrupt vector are as follows:

- The time between when the device generates an interrupt and when the process' interrupt-servicing routine receives control depends upon the IPL of the processor at the time of the interrupt. If the processor is executing at an IPL below that of the device (as is the usual case), the interrupt-servicing routine gains control within a few microseconds. However, if the processor is executing at an IPL above that of the device, the interrupt-servicing routine does not gain control until the executing code lowers the IPL below the device IPL. (Section 3.1 contains a more complete discussion of interrupt priority levels.)
- The time from the user interrupt-servicing routine's exit to the execution of the AST routine specified in the $QIO call depends on the priority of the process and whether a context switch is required.

H.4.1 Performing the Connect-to-Interrupt

Connecting to a device interrupt vector allows your program to receive notification of an interrupt from a designated device by any combination of the following means:

- By execution of a user-supplied interrupt-servicing routine
- By the setting of an event flag
- By execution of an AST service routine that gains control in process context

In addition, you can specify a cancel-I/O routine that is executed when the process disconnects from the interrupt vector or is deleted.

Before your program can run, the system manager must have performed the following actions at system generation time:

- Specify the *REALTIME_SPTS* SYSGEN parameter, reserving system page table entries for use by real-time processes. These system page table entries are used to map process-specified buffers in system space (see the **p1** argument description in Section H.4.3). The *REALTIME_SPTS* parameter value must be greater than or equal to the number of pages in buffers specified by processes connected to interrupt vectors.

- Configure the real-time device by issuing a CONNECT command to the System Generation Utility. This command names the device; its vector, register, and adapter addresses; and a skeletal driver (CONINTERR) for the device. (See the description of the CONNECT command in Section 14.2.2 and in the *VAX/VMS System Generation Utility Reference Manual*.)

At run time the process calls the $ASSIGN system service to associate a channel with the device. The process can also map the page in UNIBUS or Q22 bus I/O space containing the device registers (see Section H.3). To connect to the device interrupt vector, the process issues a $QIO call specifying the IO$_CONINTREAD or IO$_CONINTWRITE function code and as many of the following items as are appropriate:

- An interrupt-servicing routine to be executed when the device generates an interrupt.
- A buffer containing code to be executed in system context and/or data. (This buffer must be contiguous in the process' address space.)
- An AST service routine to execute and/or an event flag to be set after the interrupt-servicing routine (if any) completes. (If an AST service routine is specified, an AST parameter may also be specified.)
- A unit-initialization routine.
- A start-I/O routine.
- A cancel-I/O routine.

A nonprivileged process (that is, lacking the CMKRNL privilege) can also connect to an interrupt vector, but it can only specify an AST service routine to be executed or an event flag to be set (or both) when an interrupt is generated. Section H.4.3 describes the $QIO format for connecting to an interrupt vector.

H.4.2 The Connect-to-Interrupt Driver (CONINTERR.EXE)

The VAX/VMS connect-to-interrupt driver (CONINTERR) provides a driver interface to the system on behalf of the process. CONINTERR connects the process to the device by executing the following steps:

1. Validates the arguments to the $QIO system service call, such as the accessibility of the buffer specified in argument **p1** to the process, and the number of the event flag optionally specified in the **efn** argument.
2. Locks the physical pages of the buffer into physical memory, and maps the pages using system page table entries allocated by the *REALTIME_SPTS* SYSGEN parameter.
3. Constructs argument lists and calling interfaces to the process-specified routines by storing values in the device's unit-control block (UCB).
4. Allocates the specified number of AST control blocks to the process, and inserts each block in a queue in the device's UCB.
5. Transfers control to VAX/VMS to queue the connect to interrupt I/O packet to the CONINTERR start-I/O routine.

When the CONINTERR start-I/O routine gains control, it passes control, by means of a user-specified JSB or CALLS instruction interface, to the process-specified start-I/O routine. This routine usually initializes the device and may also start device activity.

When the device generates an interrupt, the CONINTERR interrupt-servicing routine gains control. This routine transfers control to the process-supplied interrupt-servicing routine.

H.4.3 $QIO System Service for Connect-to-Interrupt

The format of the $QIO system service to connect to an interrupt vector is given below. This explanation is limited to connecting to an interrupt vector. For a detailed description of the $QIO system service, see the *VAX/VMS System Services Reference Manual.*

The **relpag**, **chan**, and **pfc** arguments are not applicable in mapping by page frame number. The **inadr**, **retadr**, **acmode**, **gsdnam**, **ident**, and **prot** arguments have the same functions regardless of whether you specify page frame number mapping; these arguments are described in the *VAX/VMS System Services Reference Manual.*

VAX MACRO Format

```
$QIO  [efn] ,[chan] ,func [,iosb] [,astadr] [,astprm] -
      [,p1] [,p2] [,p3] [,p4] [,p5] [,p6]
```

High-Level Language Format

```
SYS$QIO([efn] ,[chan] ,func [,iosb] [,astadr] [,astprm]
     [,p1] [,p2] [,p3] [,p4] [,p5] [,p6])
```

Arguments

[efn]
[iosb]
[astadr]
[astprm]

These arguments apply to the $QIO system service completion, not to device interrupt actions. For an explanation of these arguments, see the description of the $QIO system service in the *VAX/VMS System Services Reference Manual.*

func

Function code of IO$_CONINTREAD or IO$_CONINTWRITE. The IO$_CONINTWRITE function code allows locations in the buffer pointed to by the **p1** argument to be modified; the IO$_CONINTREAD function code makes the buffer contents read-only.

[p1]

Address of a descriptor for the buffer containing code and/or data. The first longword records the number of bytes in the buffer; the second longword records the address of the buffer. The buffer size must not exceed 65,535 bytes.

[p2]

Address of an entry point list. The list consists of four longwords that contain offsets into the buffer (specified in the **p1** argument) of entry points

of process-specified routines. These longwords and their contents[2] are as follows:

Symbol	Meaning
CIN$L_INIDEV	Offset to unit-initialization routine
CIN$L_START	Offset to start-I/O routine
CIN$L_ISR	Offset to interrupt-servicing routine
CIN$L_CANCEL	Offset to cancel-I/O routine

[p3]
Longword containing flags and an optional event flag number specification.

The low-order word contains the inclusive-OR of flags describing options to the connect-to-interrupt facility. The flags and their meanings are as follows:

Flag	Meaning
CIN$M_EFN	Set event flag on interrupt
CIN$M_USECAL	Use CALL interface to process-specified routines (default is JSB interface)
CIN$M_REPEAT	Leave process connected to the interrupt vector until the connection is canceled
CIN$M_INIDEV	Process-specified unit-initialization routine is in the buffer specified in the **p1** argument
CIN$M_START	Process-specified start-I/O routine is in buffer
CIN$M_ISR	Process-specified interrupt-servicing routine is in buffer
CIN$M_CANCEL	Process-specified cancel-I/O routine is in buffer

The high-order word specifies the number of the event flag to be set when an interrupt occurs. This number is expressed as an offset to CIN$V_EFNUM.

For example, to specify that your interrupt-servicing routine is in the buffer and to set event flag 4, code **p3** as follows:

```
P3 = <CIN$M_ISR!CIN$M_EFN!4@CIN$V_EFNUM>
```

[p4]
Address of the entry mask of an AST service routine to be called as the result of an interrupt (see Section H.4.4).

[p5]
AST parameter to be passed to the AST service routine (used as the AST parameter only if the process-supplied interrupt-servicing routine does not overwrite the value).

[p6]
Number of AST control blocks to preallocate in anticipation of fast, recurrent interrupts from the device.

[2] The listed symbols are defined by the $CINDEF macro located in the library SYS$LIBRARY:LIB.MLB.

Condition Values Returned

SS$_NORMAL	System service successfully completed.
SS$_ACCVIO	The caller does not have the appropriate access to the buffer specified in the **p1** argument or to the entry point list specified in the **p2** argument.
SS$_BADPARAM	The size of the buffer specified in the **p1** argument exceeds 65535 bytes, or the number of preallocated AST control blocks specified in the P6 argument exceeds 65535.
SS$_DISCONNECT	A connection is already outstanding for the device, or a condition described in Note 2b below has occurred.
SS$_EXQUOTA	The process has exceeded its direct I/O limit quota or its AST limit quota.
SS$_ILLEFC	An illegal event flag number was specified.
SS$_INSFMEM	Insufficient system dynamic memory is available to complete the system service.
SS$_INSFSPTS	Insufficient system page table entries are available to double map the process buffer. (The value of the *REALTIME_SPTS* SYSGEN parameter must be increased.)
SS$_NOPRIV	The process does not have the CMKRNL privilege. This privilege is only required if the user specifies a buffer to be used by the process and the process-specified kernel mode routines.
SS$_UNASEFC	The process is not associated with the cluster containing the specified event flag.

See Note 3 below for additional information on these flags.

Privilege Restrictions

The connect-to-interrupt $QIO call does not require privileges if no shared buffer is specified. If the request specifies a buffer descriptor argument (that is, **p1**), the process must have the CMKRNL privilege.

Resources Required/Returned

A connect-to-interrupt request updates the process quota values as follows:

- Subtracts the number of preallocated AST control blocks in the **p6** argument from the number of outstanding ASTs remaining for the process (ASTCNT)
- Subtracts 1 (for the $QIO) from the direct I/O count (DIOCNT)

Notes

1 After the $QIO call is issued, the operation is not completed until the process or the connect-to-interrupt driver cancels I/O on the channel.

2 The connect-to-interrupt driver can cancel I/O on the channel for a number of reasons, including the following:

a The driver cannot set the specified event flag, perhaps because the process disassociated from the common event flag cluster after requesting that a flag in that cluster be set.

b The driver cannot reallocate AST control blocks quickly enough. This condition can occur because not enough AST control blocks (**p6** argument) were specified, because not enough pool space is available for the requested AST control blocks, or because the process ASTCNT quota is exhausted.

c The driver cannot queue the AST to the process.

3 If no event flag setting was requested in the **p3** argument and if no AST service routine was specified in the **p4** argument, **p6** is ignored and no AST control blocks are preallocated. If you requested an event flag be set and/or an AST service routine but did not preallocate any AST control blocks (that is, **p6** is zero), one AST control block is preallocated automatically, because the system needs one control block to set any event flag or to deliver any ASTs.

If you request an event flag and/or an AST service routine and if you preallocate any AST control blocks, the CIN$M_REPEAT bit is set automatically in the longword specified in the **p3** argument. Thus, as long as you preallocate any AST control blocks, your process will automatically remain connected to the interrupt vector to receive repeated interrupts until the process is disconnected from the interrupt vector.

If the CIN$M_REPEAT flag is not set, the process is disconnected from the interrupt vector after the first successful interrupt, and a status code of SS$_NORMAL is returned.

H.4.4 AST Service Routine

The AST service routine that you specify in call to the $QIO system service for the connect-to-interrupt operation, gains control in process context. This routine usually performs one or more of the following steps:

1 Reads or writes device registers if the process mapped I/O space.

2 Interprets data. Use caution, however, because any processing done by the AST service routine can be interrupted by a device interrupt, which might store more data or modify the buffer's contents.

3 Calls the Cancel I/O on Channel ($CANCEL) system service to disconnect the process from the interrupt. Once the process is completely disconnected, the CONINTERR driver clears all interrupts for the driver.

H.4.5 Conventions for Process-Specified Routines

Any routines that the process specifies in the connect-to-interrupt call are double-mapped, once in process space and once in system space. Each routine executes in kernel mode at an appropriate IPL:

Routine	IPL
Unit-initialization routine (after power recovery)	IPL$_POWER (IPL 31)
Start-I/O routine	IPL$_QUEUEAST (IPL 6)
Interrupt-servicing routine	Device IPL (assumed to be IPL 22)
Cancel-I/O routine	IPL$_QUEUEAST (IPL 6)

The process must have the CMKRNL privilege. Each routine must:

- Be position independent.
- Follow the rules for accessing I/O space as described in Section 6.2.
- Access only data within the buffer or nonpageable locations in system space.
- Perform any necessary synchronization of access to data in the shared buffer.
- Save any registers it uses (unless otherwise noted in the remaining sections of this appendix).
- Exit properly.
- Not incur exceptions.
- Not perform lengthy processing.
- Not dispatch to code outside the buffer specified in the **p1** argument to the $QIO system service call.

Later sections in this appendix discuss various programming language constraints and other conventions for the process-specified routines included in a connect-to-interrupt procedure. You can find additional help for writing a start-I/O routine, interrupt-servicing routine, unit-initialization routine, or cancel-I/O routine in Sections 9, 11, 13.1, and 13.2, respectively. Additionally, you may find useful the several program examples of connecting to an interrupt vector with which this appendix concludes.

H.4.6 Programming Language Constraints

Only VAX MACRO or VAX BLISS-32 should be used to code process-specified routines in system space or any references to I/O space. There is no assurance that the code generated by compilers for other languages will satisfy all the constraints described in this section.

The following constraints apply to process-specified routines in system space (that is, in the buffer specified in the **p1** argument to the $QIO call that establishes the connection to the interrupt vector):

- The compiler must generate position-independent code for the routines.
- The generated code and data must be contiguous in virtual space.
- No calls can be made to any procedure outside the buffer. (This restriction includes calls to routines in the VAX Common Run-Time Procedure Library.)

For any references to I/O space, the generated code must follow the rules for accessing I/O space discussed in Section 6.2. Device register access from high-level languages usually requires that the variable equivalent to the register be a 16-bit integer data type. You may need to check the assembly-language code generated by compilers for languages other than VAX MACRO or VAX BLISS-32 to determine whether it follows all necessary conventions.

H.4.7 Process-Specified Unit-Initialization Routine

During recovery from a power failure, VAX/VMS calls the CONINTERR unit-initialization routine. This routine marks the device as on line in the UCB$L_STS field, stores the UCB address in the IDB$L_OWNER field, and then transfers control to the process-specified unit-initialization routine. The process-specified routine executes in system context at IPL$_POWER (IPL 31).

If the process specified a JSB interface, the process unit-initialization routine gains control with the following register settings:

R0	Address of UCB
R4	Address of CSR
R5	Address of IDB
R6	Address of DDB
R8	Address of CRB

If the process specified a CALL interface, the process unit-initialization routine gains control with an argument list pointed to by AP:

0(AP)	Argument count of 5
4(AP)	Address of CSR
8(AP)	Address of IDB
12(AP)	Address of DDB
16(AP)	Address of CRB
20(AP)	Address of UCB

The process-specified unit-initialization routine may initialize device registers. It must follow these conventions:

- Not lower IPL.
- Save and restore all registers it uses, other than R0 through R3.
- Restore the stack to its original state before exiting.
- Exit with an RSB instruction (for a JSB interface) or a RET instruction (for a CALL interface).

For additional information on writing a unit-initialization routine, see Section 13.1.

H.4.8 Process-Specified Start-I/O Routine

The process-specified start-I/O routine executes in system context at IPL$_QUEUEAST (IPL 6). It is entered from the CONINTERR start-I/O routine.

If the process specified a JSB interface, the process start-I/O routine gains control with the following register settings:

R2	Address of counted argument list
R3	Address of IRP
R5	Address of UCB

If the process specified a CALL interface, the process start-I/O routine gains control with an argument list pointed to by AP:

0(AP)	Argument count of 4
4(AP)	System-mapped address of process buffer
8(AP)	Address of IRP
12(AP)	System-mapped address of the device's CSR
16(AP)	Address of UCB

The process-specified start-I/O routine may set up device registers. It must follow these conventions:

- Maintain an IPL equal to or higher than IPL$_QUEUEAST (IPL 6), and exit at IPL 6. If it raises IPL, the routine should first save the current IPL on the stack for later use in restoring IPL.
- Save and restore all registers it uses, other than R0 through R4.
- Restore the stack to its original state before exiting.
- Exit with an RSB instruction (for a JSB interface) or a RET instruction (for a CALL interface).

For additional information on writing a start-I/O routine, see Section 9.

H.4.9 Process-Specified Interrupt-Servicing Routine

A process-specified interrupt-servicing routine is entered when an interrupt from the device occurs. This routine executes in system context at device IPL.

If the process specified a JSB interface, the process interrupt-servicing routine gains control with the following register settings:

R2	Address of counted argument list
R4	Address of IDB
R5	Address of UCB

If the process specified a CALL interface, the process interrupt-servicing routine gains control with an argument list pointed to by AP:

0(AP)	Argument count of 5
4(AP)	System-mapped address of process buffer
8(AP)	Address of AST parameter
12(AP)	System-mapped address of the device's CSR
16(AP)	Address of IDB
20(AP)	Address of UCB

The process-specified interrupt-servicing routine usually performs one or more of the following steps:

1 Copies the contents of device registers into the shared buffer or the AST parameter.

2 Writes to a device register to clear the interrupt condition, if such an operation is required for the device.

3 Restarts the device, or returns an offset, a byte count, or actual data as an AST parameter.

4 Returns an interrupt status to the VAX/VMS connect-to-interrupt driver (CONINTERR).

The process-specified interrupt-servicing routine, like those supplied by VAX/VMS, has the following characteristics:

- It is mapped in system space.
- It executes on the interrupt stack.
- It executes at the IPL of the device that requested the interrupt.

Because of these characteristics, the interrupt-servicing routine executes as part of the VAX/VMS operating system rather than in the context of your user process. As part of the operating system, the interrupt-servicing routine has access to system data bases not available to user processes. However, because an interrupt-servicing routine has these capabilities and executes at a raised IPL, you must code it carefully to avoid disrupting the system.

The routine must follow these conventions:

- Maintain an IPL equal to or higher than device IPL. (If the IPL is raised, the current IPL should first be saved on the stack for later use in restoring IPL.)
- Save and restore all registers it uses, other than R0 through R4.
- Restore the stack to its original state before exiting.
- Set or clear the low bit of R0, as a status value, before exiting. The status values are as follows:

Bit 0 of R0	Meaning
Clear	Dismiss the interrupt. The process is not notified of the interrupt.
Set	Set the event flag if CIN$M_EFN bit is set in the **p3** argument to the $QIO system service call, and queue the AST if **p4** specifies an AST service routine.

- Returns to the CONINTERR interrupt-servicing routine with a RET instruction (for a CALL interface) or RSB instruction (for a JSB interface)

Depending on the interrupt status, the CONINTERR interrupt-servicing routine queues a fork process to run at a lower IPL. Then the interrupt-servicing routine exits from the interrupt with an REI instruction. When the CONINTERR fork process gains control, it queues an AST or posts an event flag to the process (or both).

For additional information on writing an interrupt-servicing routine, see Section 11.

H.4.10 Process-Specified Cancel-I/O Routine

When the user process issues a cancel-I/O request for a device connected to the process, the CONINTERR cancel-I/O routine first checks to determine whether the process can indeed cancel I/O for this device. If it can, the CONINTERR cancel-I/O routine transfers control to the process-specified cancel-I/O routine. This routine executes in system context at IPL 8 (fork IPL).

If the process specified a JSB interface, the process cancel-I/O routine gains control with the following register settings:

R2	Negated value of channel index number
R3	Address of current IRP
R4	Address of PCB for process canceling the I/O
R5	Address of UCB

If the process specified a CALL interface, the process cancel-I/O routine gains control with an argument list pointed to by AP:

0(AP)	Argument list count of 4
4(AP)	Negated value of channel index number
8(AP)	Address of current IRP
12(AP)	Address of PCB for process canceling the I/O
16(AP)	Address of UCB

The process-specified cancel-I/O routine may clear device registers and set the UCB$V_CANCEL bit in UCB$L_STS. It must follow these conventions:

- Maintain an IPL equal to or higher than IPL$_QUEUEAST (IPL 6), and exit at IPL 6. If it raises IPL, the routine should first save the current IPL on the stack for later use in restoring IPL.
- Save and restore all registers it uses, other than R0 through R3.
- Check the UCB$V_BSY bit in UCB$L_STS to validate that the channel index number represents that the process is still connected to the device.
- Place a completion status in R0 and R1. VAX/VMS places the values in these registers in the I/O status block associated with the connect-to-interrupt $QIO call.
- Restore the stack to its original state before exiting.
- Exit with an RSB instruction (for a JSB interface) or a RET instruction (for a CALL interface).

For additional information on writing a cancel-I/O routine, see Section 13.2.

H.5 Real-Time Applications Examples

To understand how the connect-to-interrupt facility is useful for programming real-time devices, consider devices used in three types of real-time applications:

1. Asynchronous event reporting without data—devices that generate an interrupt as the result of an external event not initiated by a programmed request.
2. Program-driven data collection—devices that generate an interrupt as the result of a programmed request, and make the result of the request available as data in a device register at the time of the interrupt.
3. Asynchronous event reporting with data—one device triggers another device by generating an interrupt that causes a programmed request to be sent to the other device, which in turn generates an interrupt.

Examples of these three types of real time applications and models of programs to handle the devices follow.

Note: The configurations described in the examples in this section are not officially supported; DIGITAL does not provide device driver, UETP, or diagnostic support for certain devices mentioned. The examples are provided merely as possible models for users who wish to design real-time applications using unsupported devices or configurations.

The files in the SYS$EXAMPLES directory whose names begin with "LABIO" illustrate an application using the connect to interrupt technique. Included is a program example illustrating data definitions and coding used to connect to a device interrupt vector.

H.5.1 Example 1: KW11–W Watchdog Timer

This type of device reports asynchronous external events: it generates an interrupt as a result of an external event not initiated by a programmed request. The only data of interest to be passed to the user process is the occurrence of the external event. Such devices include contact and/or solid state interrupts, and clocks or counters. The program may need to initiate clock and counter devices by means of a programmed request, but any subsequent interrupts are the result of external events only.

In this example, a dual-processor system uses two KW11–W watchdog timers connected back-to-back to monitor CPU failures. Each processor must arm its timer at regular intervals to prevent the timer from operating a relay that outputs an alarm signal. The alarm output of each timer is connected to the receive input of the other watchdog. If processor A fails and its watchdog times out, the alarm output generates an interrupt on processor B by way of the second watchdog timer.

The watchdog control program on each processor simply addresses the timer at regular intervals. If the interval passes without the timer being addressed, the timer operates an output relay that generates an interrupt to the second CPU. For this example, assume that the interval is 5 seconds (Section H.5.3 contains an example that addresses the problem of a much smaller time interval.)

The watchdog control program on processor A executes as follows:

1. Assigns a channel to the device.

2 Calls the $CRMPSC system service to map to the I/O page in order to address the device registers.

3 Issues a connect-to-interrupt $QIO call to connect the program to the watchdog timer for processor B; specifies the addresses of an interrupt-servicing routine and an AST routine.

4 Writes a value to a device register to start the timer.

5 Calls the $SETIMR system service to request that an event flag be set after a specified interval (for example, 5 seconds).

6 Calls the $WAITFR system service to wait for the event flag.

7 When the event flag is set, writes a value to a device register to reset the timer.

8 Loops to Step 5.

The same control program runs on processor B except that it connects to the watchdog timer for processor A. If either processor fails, the watchdog timer generates an interrupt on the other processor.

The standby processor that receives the interrupt gains control in the VAX/VMS connect-to-interrupt driver (CONINTERR), which calls a process-supplied interrupt-servicing routine (defined in step 3 above) that handles the interrupt as follows:

1 Sets the KW11–W switch relay register to clear the timer interrupt condition.

2 Sets a status flag that will cause an AST to be delivered to the control program that connected to the interrupt.

3 Returns to CONINTERR.

CONINTERR completes the interrupt handling as follows:

1 Schedules a fork process at a lower IPL. This fork process, when it gains control, will queue an AST to the user program.

2 Executes an REI instruction to return from the interrupt.

The timer control program on the standby processor regains control in an AST routine. This routine responds to the other processor's failure by switching over and assuming control of the other processor's tasks (or whatever is appropriate).

H.5.2 Example 2: AD11–K, AM11–K A/D Converter with Multiplexer Connected to the UNIBUS

This type of device provides program-driven data collection: it generates an interrupt as the result of a programmed request to the device, and makes the result of the request available as data in a device register. Typical devices include A/D converters and digital I/O registers.

The data collection operation is usually repetitive for such applications. Therefore, the interrupt service routine must be capable of buffering data from the device in order to ensure that no data is lost because of the high-speed data transfer rate. A typical buffer size for this sampling technique might be 32 16-bit words.

In this example, a user program controls an AD11–K/AM11–K combination that accepts analog data from thermocouples. The AD11–K converts analog data to digital data and returns the data in a device register. Every 10 seconds, the program samples 16 to 32 out of 64 channels at gain settings that may vary based on the thermocouple type and previous samplings.

To collect data efficiently, the program buffers data in a process-specified interrupt-servicing routine, and requests delivery of an AST to the user process when all the requested channels have been sampled. To perform variable sampling, the program passes parameters to the interrupt-servicing routine.

The program establishes a protocol to communicate between the program and the interrupt-servicing routine. The protocol defines a data area shared by the main program, the interrupt-servicing routine, and the AST routine. The data area contains parameters from the program and data from the AD11–K. The data area is a 98-word array used as follows:

1. Elements 1-2 of the data area contain an index to the next buffer location to be filled, and a count indicating the number of samplings still to be taken. The main program initializes these values before starting the device. The interrupt-servicing routine reads and modifies these values in the process of copying data and determining when to stop sampling.
2. Elements 3-66 of the data area are reserved for interrupt service routine parameters. Each pair of elements contains the number of a channel and a gain value. The main program loads these parameters before starting the device.
3. Elements 67-98 of the data area receive the data that the interrupt-servicing routine reads from the AD11–K data buffer register. The AST routine later reads data from this part of the buffer.

The program sets up for the sampling as follows:

1. Assigns a channel to the device.
2. Calls the $CRMPSC system service to map to the I/O page in order to address the device registers.
3. Initializes the data area by writing a 67 (the index to the next buffer location to be filled) into element 1, and the number of samples to take into element 2 of the data area; clears elements 3 through 98 of the data area.
4. Writes channel numbers and gain values into the parameter section of the data area.
5. Issues a connect-to-interrupt $QIO call to connect the process to the A/D converter; specifies the addresses of the area to be double mapped, an offset to the interrupt-servicing routine, and an AST routine.
6. Sets the start and interrupt-enable bits in the AD11–K status register to start the A/D converter.
7. Calls the $HIBER system service to place the process in a wait state.

As soon as the AD11–K has converted the first sample, the device generates an interrupt. The VAX/VMS CONINTERR routine calls the process-specified interrupt-servicing routine. This process-specified routine executes as follows:

1. Computes the next location to be written in the buffer by reading the first element in the data area.

2 Reads 12 bits of data from the A/D buffer register into the next location in the buffer.

3 Updates the buffer offset and count elements at the beginning of the data area.

4 If all requested samples have been collected, writes the address of the data area into the AST parameter, sets a status flag that will cause an AST to be delivered to the control program, and returns to the CONINTERR routine.

5 Otherwise, sets the start bit in a device register to restart the device and returns to the CONINTERR routine with a status flag requesting no AST delivery or event flag setting.

Based on the interrupt status from the process-specified interrupt-servicing routine, the CONINTERR routine completes the interrupt processing by queuing a fork process that will queue an AST to the user process. When the process gains control in the AST service routine, this routine processes the samples in the following steps:

1 Clears the interrupt-enable bit in the device status register.

2 Examines the data collected in order to adjust channel selection and/or gain values for the next sampling.

3 Copies the data to a file.

4 Reinitializes the data area.

5 Calls the $SCHDWK system service to wake the process after a short interval (for example, 10 seconds).

6 Returns.

When the time interval elapses, the process regains control. The program can then restart the sampling process by again setting the start and interrupt-enable bits in the AD11–K status register.

H.5.3 Example 3: KW11–P Real-Time Clock and AD11–K Converter Connected to the UNIBUS

This type of device reports asynchronous external events by collecting data: one device triggers another device by generating an interrupt that causes a programmed request to be sent to the other device, which in turn generates an interrupt. A typical example is a clock-driven A/D operation for precise time sampling as required in signal processing. This processing technique is often used in laboratories. The amount of data collected in such a timed sampling might typically be 200 to 1000 16-bit words.

In this example, the main program sets up the real-time clock to generate interrupts periodically. At regular intervals, the clock interrupt triggers a programmed request for an A/D conversion operation. The AD11–K collects a sample, and interrupts the CPU with a "done" interrupt and 12 bits of data. The AD11–K interrupt-servicing routine buffers the data and, if the buffer is full, causes an AST to be delivered to the process. The process, gaining control in an AST routine, copies the buffered data to another buffer or to disk.

Programming these device functions is slightly more complicated than the previous example. The main program must specify a large buffer to be used in ring fashion to guarantee that data is not lost between clock-driven samplings. In addition, the program must connect to two device interrupts—one for the clock and one for the A/D converter.

The protocol used by the main program, the interrupt-servicing routine, and the AST routine is similar to the previous example. The data area is larger: 4K words of buffer area follow the parameter area. The A/D converter interrupt-servicing routine and the AST routine treat the 4K-word buffer as four buffer sections of 1K words per section. The first element in each 1K buffer section is a flag indicating whether the section is in use. The AST resets the flag value after copying the contents of the buffer. The interrupt-servicing routine uses a buffer section only if the section's flag value indicates that the buffer has been emptied.

The main program starts the sampling with the following steps:

1. Assigns channels to the clock and to the A/D converter.
2. Calls the $CRMPSC system service to map to the I/O page in order to address the device registers.
3. Initializes the data buffer by writing a 67 (the index to the next buffer location to be filled) into element 1, and the number of samples to take into element 2 of the data area; clears elements 3 through 4096 of the data area; flags each page of the buffer as available.
4. Writes channel numbers and gain values into the parameter segments of the data area.
5. Issues a connect-to-interrupt $QIO call to connect the process to the clock, and specifies the address of an interrupt-servicing routine.
6. Issues a connect-to-interrupt $QIO call to connect the process to the A/D converter; and specifies the addresses of the area to be double mapped, an offset to the interrupt-servicing routine and an AST routine.
7. Sets the sampling interval by writing a 16-bit value into the KW11–P count set buffer register.
8. Starts the clock by setting the run, mode, rate selection, and interrupt-enable bits in the KW11–P control and status register. Setting the mode bit causes repeated interrupts generated at a rate specified in the time interval.
9. Calls the $HIBER system service to place the process in a wait state.

The clock interrupts when zero (underflow) occurs during a countdown from the preset interval count. The VAX/VMS CONINTERR routine calls the process-specified clock interrupt-servicing service routine. This process-specified routine starts the A/D conversion as follows:

1. Starts the A/D converter by setting the start and interrupt-enable bits in the AD11–K status register.
2. Sets interrupt status that prevents AST delivery or event flag setting as a result of this interrupt.
3. Returns to CONINTERR.

Starting the A/D converter results in an interrupt from the AD11–K, and control passes, by way of CONINTERR, to the AD11–K interrupt-servicing routine. This routine executes as follows:

1. If this sample is the first sample for a new buffer (indicated by a flag in the data area), the routine moves to the next buffer section (branches to error handling if the buffer is still full), and sets up the first two elements of the data area to indicate the buffer section to be written next. Then it sets the flag at the start of the new buffer section and sets a flag in the data area to indicate that sampling is occurring.
2. The routine computes the next location to be written in the buffer by reading the first location in the data area.
3. The routine reads 12 bits of data from the A/D buffer register into the next location in the buffer.
4. The routine updates the buffer offset and count values in the data area.
5. If this sample fills the data sector, the routine writes the offset of the filled sector from the start of the 4K-word buffer into the AST parameter, sets a status flag that will cause an AST to be delivered to the control program, and sets a flag indicating that a new data section is to be started.
6. The routine returns to CONINTERR.

The AST routine copies and fills the next buffer section with zeros to indicate that the section is again available to the interrupt-servicing routine. When the next clock interrupt occurs, the data can be written to the next buffer section, even if the AST routine has not yet emptied the previous buffer section.

Glossary

ACF: See *configuration-control block.*

ACP: See *Ancillary Control Process.*

adapter-control block (ADP): A structure in the I/O database that describes an I/O adapter and its resources.

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 by 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 that passes control to a user-defined routine. ASTs enable a user process to be notified of the occurrence of a specific event, asynchronously with respect to the execution of the user process.

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 execution of 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 allows I/O device controllers to communicate with main memory and the central processor. These I/O controllers may reside on the same bus as memory and the central processor (for instance, in a VAX 8200 or MicroVAX I system), or they may be on a separate bus entirely (for instance, in a VAX–11/780 or VAX 8600 system). In the latter case, an I/O adapter enables and controls the communications between the I/O bus and the processor and memory.

The backplane interconnect is called the synchronous backplane interconnect (SBI) on the VAX–11/780 and VAX 8600 processor, the CPU-to-memory interconnect (CMI) on the VAX–11/750 processor, and the VAXBI on the VAX 8800 and VAX 8200 processors. The MicroVAX II and MicroVAX I processors use the Q22 bus as a backplane.

base register: A general register that contains the base address of (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 several bytes of data in a single backplane-interconnect transfer.

buffered I/O: An I/O operation, such as terminal or mailbox I/O, in which an intermediate buffer from the system's buffer pool is used instead of a buffer in process space. See also *direct I/O*.

bugcheck: The operating system's diagnostic that detects and reports internal inconsistencies. If the system can continue running, it declares a *nonfatal bugcheck* and reports it in an error log entry. A serious error results in a *fatal bugcheck*. As a result of a fatal bugcheck, the system shuts itself down in an orderly fashion.

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 database 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 database that describes the activity on a particular controller. The channel-request block for a controller contains pointers to the queue of drivers waiting to access a device through the controller.

configuration-control block: A structure in the I/O database used by the autoconfiguration facility of the System Generation Utility to describe the device it is adding to the system. The information stored in the configuration-control block might be useful to a device driver's unit-delivery routine.

configuration register: A control and status register for an I/O 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 physical pages in the I/O space which contain 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 diagnostic programs.

console terminal: The hard-copy terminal connected to the central processor's console.

context: The environment of an activity. See also *process context, hardware context, and software context.*

controller data channel: A logical path to which the driver of a device that shares a controller must gain access before it can use the controller to activate a device.

control and status register (CSR): A control and status register for a device or controller. It resides in the processor's I/O space.

CRB: See *channel-request block.*

CSR: See *control and status register.*

database: A collection of related data structures; all the occurrences of data described by a database management system.

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 database that identifies the generic device/controller name and driver name for a set of devices that share the same controller.

device driver: The set of instructions and tables that handles physical I/O operations to a device.

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 controller logic used to request device functions (such as I/O transfers) and/or report status.

device unit: One device 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 program: A program that tests hardware, firmware, peripherals logic, or memory, and that reports any faults it detects.

direct data-path: A UNIBUS adapter data path that transfers several 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.*

direct-memory-access (DMA) transfer: The type of I/O transfer by which a device controller accesses memory directly and, as a result, can transfer a large amount of data without requesting a processor interrupt after each of the smaller amounts. Contrast with *programmed-I/O (PIO) transfer.*

DPT: See *driver-prologue table.*

drive: The electromechanical unit of a mass storage device on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

driver-dispatch table (DDT): A table in a driver that lists the addresses of the entry points of standard driver routines and the sizes of diagnostic and error-logging buffers for the device.

driver-prologue table (DPT): A table in a driver that describes the driver and the type of device it drives to the VAX/VMS procedure that loads drivers into the system.

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 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 that is independent of the current instruction).

There are three types of hardware exceptions: traps, faults, and abortions. Examples are: attempts to execute a privileged or reserved instruction, trace traps, page faults, compatibility-mode faults, execution of breakpoint instructions, and arithmetic traps.

executive: The software that provides the basic control and monitoring functions of the operating system.

FDT: See *function-decision table.*

FDT routines: Driver routines called by the $QIO 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-servicing routine that is activated by a software interrupt at a fork-interrupt priority level. Once activated, it dispatches driver fork processes from a fork queue until no processes remain in the queue for that IPL.

fork process: A process with a minimal context that executes instructions under a set 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 the system's virtual address space; it can refer to and modify static storage that is never modified by procedures that execute at a higher IPL. VAX/VMS uses software interrupts and fork processes to synchronize executive operations.

fork queue: A queue of 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 preprocessing routines associated with each valid function of the device.

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 access mode in which the processor is executing
- The contents to be loaded in the SP for every access mode other than the current access mode

When a process is executing, its hardware context is continually being updated by the processor. When 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' hardware PCB resides in its process header (PHD).

IDB: See *interrupt-dispatch block.*

interrupt: An event other than an exception or a 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 database that describes the characteristics of a particular controller and points to devices attached to that controller.

interrupt priority level (IPL): The level at which a software or hardware interrupt is generated. There are 32 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-servicing routine.

interrupt-servicing routine (ISR): A routine executed when a device interrupt occurs.

interrupt stack (IS): The system-wide stack used when executing instructions in interrupt context. In the VAX/VMS operating system, all hardware interrupts (and all software interrupts above IPL 3) are serviced on the system-wide interrupt stack and not one of the per-process stacks.

interrupt-stack pointer (ISP): The pointer to the top of 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 database: A collection of data structures that describe 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-dispatch 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 $QIO 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 $QIO 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 database that describes an individual I/O request. The $QIO 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 and status register and data registers. These regions are physically discontiguous.

I/O-status block (IOSB): A data structure associated with the $QIO 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 is not returned from the system 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-servicing routine*.

limit: The size or number of 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 *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-logical-block 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 interprocess communication (for example, the error logger and OPCOM read from system-wide mailboxes). 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.

machine check: An exception that is reported when the processor or an external adapter detects an internal error. If the machine check is recoverable, the machine check handler the condition in an error log entry. If an unrecoverable machine check occurs while the processor is in supervisor or user mode, the machine check handler reports the exception to that mode. However, if an unrecoverable machine check occurs in kernel or executive mode, a fatal bugcheck results. See also *exception* and *bugcheck*.

mapping register: See *scatter-gather map*.

MASSBUS adapter (MBA): An interface device between the backplane interconnect and the MASSBUS.

memory interconnect: The internal processor bus for the VAX–11/750.

nexus: A physical connection to the synchronous backplane interconnect (SBI). For example, when connected to the SBI, the central processor, memory subsystem, and I/O controllers are known as nexuses. See also *Synchronous Backplane Interconnect*.

node: A VAXBI interface—such as a central processor, controller, or memory subsystem—that occupies one of 16 logical locations on a VAXBI bus. See also *VAXBI*.

offset: A displacement from the beginning of a data structure to the beginning of a field within that data structure. Offsets for items within a data structure usually have an associated symbol. The name of the symbol is used to refer to the field; its value is the offset.

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 disks. 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 a 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 operations.

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, hardware context, 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 value that uniquely identifies a process. Each process has a PID and a 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 is 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 the process itself might). Levels 0 through 15 are used for normal processes. The system can temporarily increase the priority of a normal process based on the activity of the process.

Contrast with *interrupt priority level.*

programmed-I/O (PIO) transfer: The type of I/O transfer, largely conducted by the driver program, that requires a processor interrupt after each byte or word is transferred. Drivers for relatively slow devices, such as printers, card readers, terminals, and some disk and tape drives use PIO data transfers. Contrast with *direct-memory-access (DMA) transfer.*

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.*

Q22 bus: The hardware interconnect by which MicroVAX II and MicroVAX I peripheral devices communicate with main memory and the processor.

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. See also *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 *limit.*

return status code: See *status code.*

SBI: See *Synchronous Backplane Interconnect.*

scatter-gather map: A technique by which a set of physically discontiguous pages are made to seem contiguous to an I/O controller performing a direct-memory-access transfer. It is I/O adapter hardware that generally provides this means of mapping physical pages to I/O adapter address space.

small process: A system process that has no control region in its virtual address space and has an abbreviated context. Examples are the 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; it cannot be swapped.

software context: The context maintained by VAX/VMS to describe a process. See also *software process-control block (PCB).*

software process-control block (software PCB): The data structure used to contain a process' 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 the process 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 needed resources and for activating the device unit. An example of a needed resource is the controller's data channel.

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 or VAX 8600 hardware that interconnects the processor, memory controllers, MASSBUS adapters, the UNIBUS adapter.

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 is also alert for device timeouts and performs time-dependent scheduling upon request. The timer's interrupt-servicing 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. On a VAX 8200 or VAX 8800, it is called a BUA.

unit-control block (UCB): A structure in the I/O database that describes the characteristics of a device unit and current activity on it. The unit-control block also holds the fork block for its unit's device driver; the fork block is part of the UCB and 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 failure and during execution of 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 failures.

VAXBI: The part of the VAX 8200 hardware that connects I/O adapters with memory controllers and the processor. In a VAX 8800 system, the part of the hardware that connects I/O adapters with the bus that interfaces with the processor and memory.

vector: A one-dimensional array.

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.

For the purpose of handling exceptions, users can declare up to two software-exception vectors (primary and secondary) for each of the four processor-access modes. Each vector contains the address of a condition handler, and is a longword.

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's fork process to be suspended until the device requests an interrupt or the device times out.

XDELTA: A software tool for debugging operating systems and drivers.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

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

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

— — Do Not Tear - Fold Here —

Cut Along Dotted Line

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

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

— — Do Not Tear - Fold Here —

Cut Along Dotted Line