

The word "digital" is written in a lowercase, sans-serif font, with each letter contained within its own white rectangular box. The boxes are arranged in a single horizontal row.

digital

A white rectangular box with a thin black border, centered on the page. It contains the title and order number.

VAX/VMS
Guide to Writing
a Device Driver

Order No. AA-H499B-TE

The word "VAX11" is written in a large, bold, white, sans-serif font. The letters are closely spaced and have a slightly stylized appearance.

VAX11

March 1980

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

VAX/VMS
Guide to Writing
a Device Driver

Order No. AA-H499B-TE

SUPERSESSION/UPDATE INFORMATION: This revised document supersedes the VAX/VMS Guide to Writing a Device Driver (Order No. AA-H499A-TE).

OPERATING SYSTEM AND VERSION: VAX/VMS V02

SOFTWARE VERSION: Not applicable

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

First Printing, February 1979
Revised, March 1980

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

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

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

Copyright © 1979, 1980 by Digital Equipment Corporation

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECTape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
DATATRIEVE	TRAX	

CONTENTS

		Page
PREFACE		xi
PART I		
CHAPTER 1	INTRODUCTION TO DEVICE DRIVERS	1-1
1.1	ASYNCHRONOUS NATURE OF A DEVICE DRIVER	1-1
1.2	FORK PROCESSES	1-3
1.3	PROCESS CONTEXT AND INTERRUPT CONTEXT	1-3
1.4	DEVICE DEPENDENCE AND DEVICE INDEPENDENCE	1-4
1.5	THE I/O DATA BASE	1-5
1.5.1	Control Blocks in the I/O Data Base	1-6
1.5.1.1	Device Data Block	1-6
1.5.1.2	Unit Control Block	1-6
1.5.1.3	Channel Request Block	1-6
1.5.1.4	Interrupt Data Block	1-6
1.5.1.5	Adapter Control Block	1-7
1.5.1.6	Channel Control Block	1-7
1.5.2	I/O Request Packets	1-7
1.6	SYNCHRONIZATION	1-7
1.6.1	Interrupt Priority Levels	1-7
1.6.2	Fork Queues	1-8
1.6.3	Resource Wait Queues	1-8
1.7	FUNCTIONS OF A DEVICE DRIVER	1-9
1.7.1	Initialization Routines	1-10
1.7.2	FDT Routines	1-10
1.7.3	Start I/O Routine	1-10
1.7.4	Interrupt Service Routine	1-11
1.7.5	Device Timeout Handler	1-11
1.7.6	Cancel I/O Routine	1-11
1.7.7	Error-Logging Routine	1-11
1.8	AN EXAMPLE OF A UNIBUS I/O REQUEST	1-11
1.9	THE UNIBUS	1-13
1.10	PROGRAMMED I/O AND DIRECT MEMORY ACCESS I/O	1-15
1.11	BUFFERED I/O AND DIRECT I/O	1-15
1.12	LOADABLE DRIVERS	1-16
CHAPTER 2	DISCUSSION OF A LINE PRINTER QUEUE I/O REQUEST	2-1
2.1	DRIVER CODE FOR THE LP11 WRITE FUNCTION	2-2
2.2	A USER PROCESS'S I/O REQUEST	2-3
2.3	I/O PREPROCESSING BY VAX/VMS	2-3
2.4	I/O PREPROCESSING BY THE DRIVER	2-4
2.5	QUEUING THE I/O PACKET TO THE DRIVER	2-5
2.6	DRIVER DEVICE ACTIVATION	2-6
2.7	WAITING FOR A DEVICE INTERRUPT	2-6
2.8	INTERRUPT HANDLING	2-7
2.9	I/O COMPLETION PROCESSING BY THE DRIVER	2-8
2.10	I/O COMPLETION PROCESSING BY THE VAX/VMS SYSTEM	2-8

CONTENTS

	Page
CHAPTER 3	3-1
3.1	3-1
3.1.1	3-1
3.1.2	3-2
3.1.3	3-2
3.1.4	3-3
3.1.5	3-3
3.1.6	3-4
3.1.7	3-4
3.1.8	3-6
3.1.8.1	3-6
3.1.8.2	3-7
3.1.8.3	3-7
3.1.8.4	3-7
3.1.8.5	3-7
3.1.9	3-8
3.1.9.1	3-8
3.1.9.2	3-8
3.1.9.3	3-8
3.1.9.4	3-9
3.1.9.5	3-9
3.1.10	3-9
3.1.11	3-11
3.1.11.1	3-11
3.1.11.2	3-11
3.1.11.3	3-12
3.1.11.4	3-12
3.2	3-12
3.2.1	3-13
3.3	3-14
3.3.1	3-15
CHAPTER 4	4-1
4.1	4-2
4.2	4-2
4.2.1	4-3
4.2.1.1	4-4
4.2.1.2	4-5
4.2.1.3	4-7
4.2.1.4	4-7
4.2.1.5	4-7
CHAPTER 5	5-1
5.1	5-1
5.1.1	5-3
5.1.2	5-3
5.1.2.1	5-3
5.1.2.2	5-5
5.1.2.3	5-5

CONTENTS

		Page
5.1.3	Validating the I/O Function	5-7
5.1.4	Checking Process I/O Request Quotas	5-7
5.1.5	Validating the I/O Status Block	5-7
5.1.6	Allocating and Setting Up an I/O Request Packet	5-7
5.1.7	Function Decision Table Processing	5-9
5.2	HANDLING DEVICE ACTIVITY	5-10
5.2.1	Creating a Driver Fork Process to Start I/O	5-12
5.2.2	Activating a Device and Waiting for an Interrupt	5-13
5.2.3	Handling a Device Interrupt	5-14
5.2.4	Switching from Interrupt to Fork Process Context	5-14
5.2.5	Activating a Fork Process from a Fork Queue	5-15
5.3	COMPLETION OF AN I/O REQUEST	5-16
5.3.1	I/O Postprocessing	5-17
PART II		
CHAPTER 6	TEMPLATE FOR AN I/O DRIVER	6-1
6.1	CODING CONVENTIONS	6-1
6.2	RESTRICTIONS ON DEVICE REGISTER I/O SPACE USE	6-3
CHAPTER 7	CODING DEVICE DRIVER TABLES	7-1
7.1	DRIVER PROLOGUE TABLE (DPT)	7-1
7.1.1	DPTAB Macro	7-2
7.1.2	DPT_STORE Macro	7-3
7.1.3	Example of DPTAB and DPT_STORE Macro Use	7-5
7.2	DRIVER DISPATCH TABLE (DDT)	7-5
7.2.1	DDTAB Macro	7-6
7.2.2	Example of a DDTAB Macro	7-7
7.3	FUNCTION DECISION TABLE (FDT)	7-7
7.3.1	Defining Device-Specific Function Codes	7-8
7.3.2	Determining Those Functions that are Buffered I/O	7-9
7.3.3	FUNCTAB Macro	7-10
7.3.4	Example of FUNCTAB Macro Use	7-10
CHAPTER 8	CODING FDT ROUTINES	8-1
8.1	CONTEXT FOR FDT ROUTINE EXECUTION	8-1
8.2	REGISTERS PRESET FOR FDT ROUTINE EXECUTION	8-1
8.3	CONVENTIONS FOLLOWED BY FDT ROUTINES	8-2
8.3.1	Register Conventions	8-2
8.3.2	Process Context Conventions	8-3
8.4	TRANSFERRING INTO AND OUT OF AN FDT ROUTINE	8-3
8.4.1	Exit Methods	8-4
8.5	FDT ROUTINES FOR DIRECT I/O	8-4
8.6	FDT ROUTINES FOR BUFFERED I/O	8-6
8.6.1	Checking the User's Buffer	8-6
8.6.2	Allocating the System Buffer	8-6
8.6.3	Completion of Buffered I/O in I/O Postprocessing	8-7
8.7	FDT ROUTINES PROVIDED BY VAX/VMS	8-8
8.7.1	EXE\$ONEPARM	8-9

CONTENTS

		Page
8.7.2	EXE\$READ	8-9
8.7.3	EXE\$SENSEMODE	8-10
8.7.4	EXE\$SETCHAR	8-11
8.7.5	EXE\$SETMODE	8-11
8.7.6	EXE\$WRITE	8-12
8.7.7	EXE\$ZEROPARM	8-13
8.8	EXIT ROUTINES IN THE VAX/VMS SYSTEM	8-13
8.8.1	EXE\$ABORTIO	8-13
8.8.2	EXE\$FINISHIO and EXE\$FINISHIOC	8-14
8.8.3	EXE\$QIODRVPKT	8-15
8.8.4	EXE\$ALTQUEPKT	8-17
CHAPTER 9	CODING THE START I/O ROUTINE	9-1
9.1	TRANSFERRING CONTROL TO START I/O	9-1
9.2	CONTEXT OF A DRIVER FORK PROCESS	9-1
9.3	ACTIVATING THE DEVICE	9-2
9.3.1	Obtaining Controller Access	9-2
9.3.2	Getting the I/O Function Code and Converting the Code and Modifiers	9-4
9.3.3	Obtaining a Buffered Data Path	9-4
9.3.4	Loading Map Registers	9-4
9.3.5	Computing the Transfer Length	9-5
9.3.6	Computing the Transfer Start Address	9-5
9.3.7	Preparing the Device Activation Bit Mask	9-5
9.3.8	Blocking All Interrupts	9-5
9.3.9	Checking for Power Failure	9-6
9.3.10	Activating the Device	9-6
9.4	WAITING FOR AN INTERRUPT OR TIMEOUT	9-6
9.4.1	WFIKPCH and WFIRLCH Macro Formats	9-7
9.4.2	Expansion of WFIKPCH Macro	9-7
9.4.3	IOCSWFIKPCH Routine	9-7
9.5	RESPONDING TO AN EXPECTED DEVICE INTERRUPT	9-8
CHAPTER 10	CODING FOR UNIBUS DMA TRANSFERS	10-1
10.1	REQUESTING A BUFFERED DATA PATH	10-2
10.1.1	Requesting a Buffered Data Path (with Wait)	10-2
10.1.2	Requesting a Buffered Data Path (No Wait)	10-3
10.1.3	Requesting a Permanent Buffered Data Path	10-3
10.1.4	Requesting the Direct Data Path	10-3
10.1.5	Mixed Direct and Buffered Data Path Transfers	10-3
10.2	REQUESTING UBA MAP REGISTERS	10-4
10.2.1	Allocation of Map Registers	10-4
10.2.2	Permanent Allocation of Map Registers	10-4
10.3	LOADING THE UBA MAP REGISTERS	10-5
10.4	COMPUTING THE STARTING ADDRESS OF A TRANSFER	10-6
10.5	ACTIVATING THE DEVICE	10-6
10.6	COMPLETION OF A DMA TRANSFER	10-7
10.6.1	Purging the Data Path	10-7
10.6.2	Releasing a Buffered Data Path	10-8
10.7	RELEASING UBA MAP REGISTERS	10-8
CHAPTER 11	CODING INTERRUPT SERVICE ROUTINE	11-1
11.1	DELIVERING A DEVICE INTERRUPT TO A DRIVER	11-1
11.2	INTERRUPT CONTEXT	11-3
11.3	SERVICING A SOLICITED INTERRUPT	11-4

CONTENTS

		Page
	11.4 SERVICING AN UNSOLICITED INTERRUPT	11-5
	11.4.1 Examples of Unsolicited Input Handling	11-6
CHAPTER	12 COMPLETING THE I/O REQUEST	12-1
	12.1 I/O POSTPROCESSING	12-1
	12.1.1 EXESIOFORK	12-1
	12.1.2 Completing an I/O Request	12-2
	12.1.2.1 Releasing the Controller	12-2
	12.1.2.2 Saving Status, Count, and Device-Dependent Status	12-3
	12.1.2.3 Returning to the Operating System	12-3
	12.2 TIMEOUT HANDLERS	12-4
	12.2.1 Retrying the I/O Operation	12-4
	12.2.2 Aborting the I/O Request	12-5
	12.2.3 Sending a Message to the Operator	12-6
CHAPTER	13 CODING INITIALIZATION, CANCEL I/O, AND ERROR-LOGGING ROUTINES	13-1
	13.1 INITIALIZATION ROUTINES	13-1
	13.1.1 Initialization During Driver Loading	13-2
	13.1.2 Initialization During Recovery from a Power Failure	13-2
	13.1.3 Initialization Context	13-3
	13.2 CANCEL I/O ROUTINE	13-4
	13.2.1 Context of a Cancel I/O Routine	13-5
	13.2.2 Drivers that Need No Cancel I/O Routine	13-5
	13.2.3 Device-Independent Cancel I/O Routine	13-5
	13.2.4 Device-Dependent Cancel I/O Routines	13-6
	13.3 ERROR LOGGING ROUTINES	13-6
CHAPTER	14 LOADING A DEVICE DRIVER	14-1
	14.1 IN PREPARATION FOR LOADING	14-1
	14.2 LOADING THE DRIVER	14-2
	14.2.1 LOAD Command	14-2
	14.2.2 CONNECT Command	14-3
	14.2.3 RELOAD Command	14-5
	14.2.4 SHOW/DEVICE	14-6
	14.3 AUTOCONFIGURATION	14-7
	14.3.1 SYSGEN's Autoconfiguration	14-8
	14.3.2 The SYSGEN Device Table	14-9
	14.3.3 Floating Vector Address Calculation	14-14
	14.3.4 Floating CSR Address Calculation	14-14
	14.3.5 Rules for Configuration	14-14
	14.3.6 Example of a UNIBUS Configuration	14-15
CHAPTER	15 DEBUGGING A DEVICE DRIVER	15-1
	15.1 BOOTSTRAPPING THE SYSTEM WITH XDELTA	15-1
	15.2 LOADING THE DRIVER	15-2
	15.3 INSERTING BREAKPOINTS IN THE SOURCE CODE	15-3
	15.4 CALCULATING THE BASE OF DRIVER CODE	15-4
	15.5 REQUESTING AN XDELTA SOFTWARE INTERRUPT	15-4
	15.6 LOOKING AT THE VECTOR JUMP TABLE	15-5
	15.7 SETTING AN XDELTA BASE REGISTER	15-5
	15.8 DESTROYING REGISTER CONTENTS	15-5
	15.9 EXAMINING UCB, IRP, and PSL	15-6

CONTENTS

		Page
15.10	XDELTA COMMANDS	15-6
15.10.1	Values and Expressions	15-7
15.10.2	Special Symbols	15-8
15.10.3	Operators	15-8
15.10.4	Open and Display Command	15-8
15.10.5	Close and Display Next Location Command	15-9
15.10.6	Display Range Command	15-9
15.10.7	Indirect Command	15-9
15.10.8	Display Previous Location Command	15-9
15.10.9	Show Value Command	15-10
15.10.10	Step Instruction Command	15-10
15.10.11	Setting Breakpoints	15-10
15.10.12	Clearing Breakpoints	15-10
15.10.13	Displaying Breakpoint List	15-11
15.10.14	Setting Base Registers	15-11
15.10.15	Proceeding from Breakpoints	15-11
15.10.16	Loading PC and Continuing	15-11
15.10.17	Display Mode Control	15-12
15.10.18	The Execute String Command	15-12
15.10.19	Setting Complex Breakpoints	15-13
15.10.20	XDELTA Stored Commands	15-13
15.10.21	Stored Base Registers	15-14
15.11	DELTA	15-14
15.11.1	The EXIT Command	15-14
15.11.2	Examining and Modifying Locations in Process Space	15-14
15.12	DEBUGGING TECHNIQUES	15-14
15.12.1	References to System Addresses	15-15
15.12.2	Opening Device Registers in XDELTA	15-15
15.12.3	Incorrect References to Device Registers	15-15
15.12.4	XDELTA and System Failures	15-15
APPENDIX B	VAX/VMS MACROS INVOKED BY DRIVERS	B-1
APPENDIX C	OPERATING SYSTEM ROUTINES	C-1
APPENDIX D	SAMPLE DRIVER FOR AN A-TO-D CONVERTER	D-1
APPENDIX E	SAMPLE DRIVER FOR DR11s	E-1
APPENDIX F	MASSBUS ADAPTER	F-1
F.1	I/O DATA BASE FOR MASSBUS DEVICES	F-4
F.2	MBA CONSIDERATIONS FOR DRIVERS	F-6
F.2.1	Unit Initialization Routine	F-6
F.2.2	Start I/O Routine	F-7
F.2.2.1	Requesting a Controller Data Channel	F-8
F.2.2.2	Loading Map Registers	F-8
F.2.2.3	Releasing Controller Data Channel(s)	F-9
F.2.3	DPTAB Macro	F-9
F.3	INTERRUPT HANDLING FOR MASSBUS DEVICES	F-9
F.3.1	Looking for Another Request	F-10
F.3.2	Transferring Control to a Subcontroller's Interrupt Service Routine	F-11
F.3.3	Handling Unsolicited Interrupts	F-11
GLOSSARY		Glossary-1
INDEX		Index-1

CONTENTS

			Page
FIGURES			
FIGURE	1-1	VAX/VMS Calls to Driver Routines	1-2
	1-2	The I/O Data Base	1-5
	1-3	Processing a Sample I/O Operation	1-12
	1-4	VAX-11 Hardware Configuration	1-14
	2-1	A Line Printer Write Function	2-2
	2-2	Locating a Function Decision Table	2-4
	3-1	Servicing Hardware Interrupts	3-5
	3-2	IPL Conventions During I/O Processing	3-10
	3-3	IPL conventions During I/O Completion	3-10
	4-1	UNIBUS to SBI Address Mapping	4-3
	5-1	Sequence of Driver Execution	5-2
	5-2	Locating the Target Device	5-4
	5-3	I/O Data Structures for Three Devices on One Controller	5-5
	5-4	I/O Data Base for Two Controllers	5-6
	5-5	Driver Function Decision Table	5-9
	5-6	FDT Routines and I/O Preprocessing	5-11
	5-7	Creating a Fork Process After an Interrupt	5-15
	5-8	Reactivation of a Driver Fork Process	5-16
	6-1	Driver Organization	6-2
	8-1	Queue I/O Request Scan of a Function Decision Table	8-4
	8-2	Format of System Buffer for Buffered I/O Read Operations	8-7
	9-1	Driver Insertion into Channel Wait Queue	9-3
	11-1	Interrupt Handling Flow	11-2
	F-1	MASSBUS Configuration	F-1
	F-2	Mapping of a Virtual Address to a Page Frame Number	F-2
	F-3	Location of MASSBUS Registers	F-3
	F-4	I/O Data Base for MASSBUS Disk Unit	F-4
	F-5	I/O Data Base for MASSBUS Disk and Tape Units	F-5
	F-6	I/O Data Structures Used in Dispatching an Interrupt	F-6

TABLES			
TABLE	3-1	IPLs Defined by VAX/VMS	3-2
	7-1	VAX/VMS I/O Function Codes	7-8
	8-1	Registers Loaded by Queue I/O Request Service	8-2
	8-2	FDT Exit Methods	8-5
	15-1	XDELTA Command Summary	15-7

PREFACE

The VAX/VMS Guide to Writing a Device Driver provides the information needed to write a device driver that runs under VAX/VMS Version 2.0 and to load that driver into the operating system. VAX/VMS makes no guarantee that drivers written for VAX/VMS Versions 1.0, 1.5 and 2.0 will execute without modification on subsequent versions of the operating system. While the intent is to maintain the existing interface, some unavoidable changes may occur as new features are added. The use of internal executive interfaces other than those described in this manual is discouraged.

INTENDED AUDIENCE

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

STRUCTURE OF THIS DOCUMENT

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

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

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

- Chapter 6 contains a template for coding a device driver.
- Chapter 7 details the macros that drivers invoke to create necessary tables.
- Chapter 8 describes the coding of function decision routines.
- Chapter 9 describes the coding of a start I/O routine.
- Chapter 10 describes the UNIBUS considerations for a start I/O routine.
- Chapter 11 describes the coding of an interrupt service routine.
- Chapter 12 describes the coding of I/O completion and device timeout routines.
- Chapter 13 describes the coding of unit and controller initialization routines, I/O cancellation routines, and error-logging routines.
- Chapter 14 describes the loading of a driver into the system.
- Chapter 15 describes the debugging tool XDELTA that you can use to debug a device driver.
- Appendix A describes the I/O data base in detail. This is an important appendix for the programmer of a device driver.
- Appendix B describes the VAX/VMS macros that drivers can invoke.
- Appendix C describes the VAX/VMS routines that device drivers can call.
- Appendix D contains a sample driver for an analog-to-digital converter.
- Appendix E contains a sample driver for two connected DR11s.
- Appendix F contains information needed to write a device driver for a device attached to the MASSBUS.
- The glossary at the end of this manual defines I/O-related and driver-related terms.

ASSOCIATED DOCUMENTS

This document has the following prerequisites:

- VAX-11/780 Hardware Handbook
- VAX/VMS Summary Description and Glossary
- I/O-related portions of the VAX/VMS System Services Reference Manual

- The appendix on naming conventions in the VAX-11 Guide to Creating Modular Library Procedures
- VAX/VMS I/O User's Guide

The following documents are associated with this manual:

- VAX/VMS System Dump Analyzer Reference Manual
- VAX/VMS System Manager's Guide

CONVENTIONS USED IN THIS DOCUMENT

This manual describes code transfer operations in three ways.

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

SUMMARY OF TECHNICAL CHANGES

Please refer to the section on system programming in the VAX/VMS Release Notes (Version 2.0) for a detailed description of the technical changes to VAX/VMS that affect device drivers.

PART I

CHAPTER 1
INTRODUCTION TO DEVICE DRIVERS

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

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

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

1.1 ASYNCHRONOUS NATURE OF A DEVICE DRIVER

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

- An I/O preprocessing routine or routines that validate device-specific parameters of an I/O request, format data, allocate system buffers, and lock pages in memory
- A start I/O routine that activates the device
- An interrupt service routine that responds to interrupts from the device unit
- An error recovery routine that retries I/O operations and performs other error handling
- An error-logging routine that writes the contents of device registers and other data into an error buffer for the system
- A cancel I/O routine that prevents further processing of an I/O request

INTRODUCTION TO DEVICE DRIVERS

- An initialization routine that readies a device or controller for operation when the system is bootstrapped or during power failure recovery
- A driver prologue table that describes the driver and the device type to the VAX/VMS procedure that loads drivers into the system
- A driver dispatch table that lists the entry point addresses of standard driver routines and records the size of diagnostic and error-logging buffers for the device type
- A function decision table that lists all valid function codes for the device and lists the addresses of I/O preprocessing routines associated with each valid function

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

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

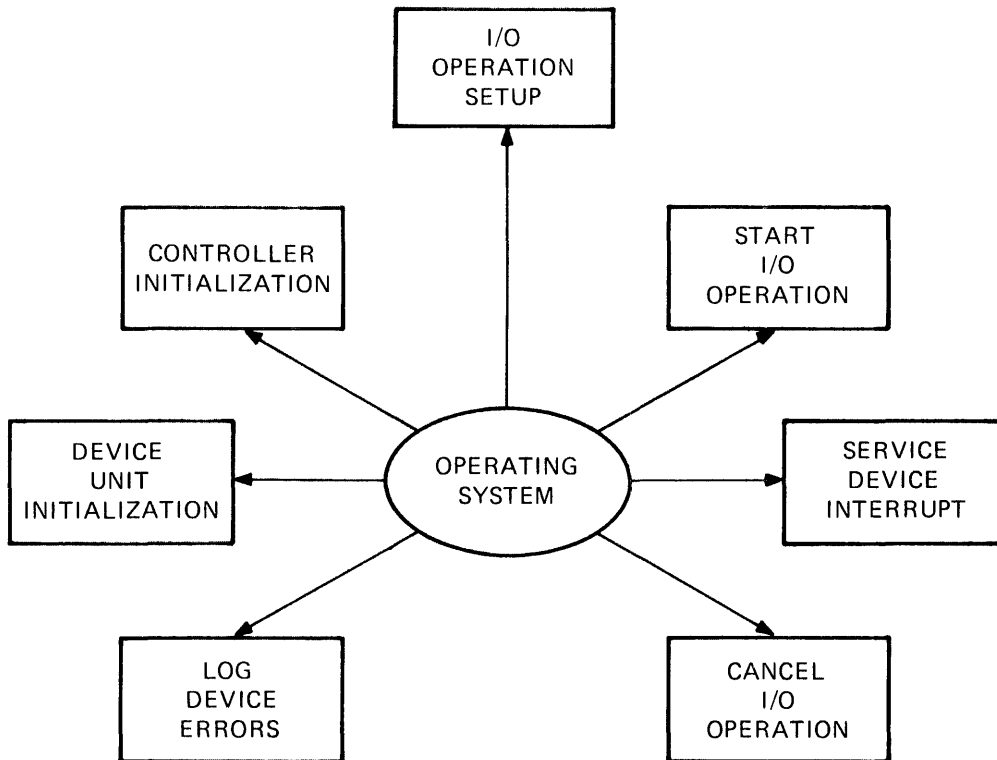


Figure 1-1 VAX/VMS Calls to Driver Routines

MA-2420

INTRODUCTION TO DEVICE DRIVERS

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

1.2 FORK PROCESSES

A fork process is a process that is created dynamically and has minimal context. Fork processes execute entirely within the system address space. The VAX/VMS operating system creates and schedules a fork process by constructing a specialized control block called a fork block, inserting the fork block in a fork queue, and requesting a software interrupt. Fork queues and fork process dispatching are described further in Section 1.6.2.

A driver fork process has the following context:

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

The unit control block also contains the driver's fork block. Section 1.5 describes the unit control block and other control blocks in the I/O data base.

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

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

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

1.3 PROCESS CONTEXT AND INTERRUPT CONTEXT

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

- A user process executing in user mode issues a write Queue I/O Request system service.
- The Queue I/O Request system service gains control in user process context but in kernel mode.

INTRODUCTION TO DEVICE DRIVERS

- The system service uses the driver's function decision table to call the appropriate preprocessing routines. These routines, called FDT routines, execute in full process context in kernel mode.
- When preprocessing is complete, a VAX/VMS routine creates a fork process to execute the driver's start I/O routine in kernel mode.
- The start I/O routine activates the device unit and suspends itself. At this point, VAX/VMS suspends the fork process executing the start I/O routine and saves sufficient context to reactivate the start I/O routine at the point of suspension.
- When the device completes the data transfer, it issues an interrupt. The interrupt causes the system to activate the driver's interrupt service routine.
- The interrupt service routine executes to handle the device interrupt. It then causes the start I/O routine to resume in interrupt context.
- The start I/O routine regains control in interrupt context but almost immediately issues a request to the operating system to transform its context to that of a fork process. This action dismisses the interrupt.
- When reactivated in fork process context, the start I/O routine performs device-specific I/O completion and passes control to the system for additional I/O postprocessing.
- VAX/VMS I/O postprocessing performs processing at a software interrupt priority level and then issues a kernel mode AST for the user process requesting I/O.
- When the kernel mode AST is delivered, the AST routine executes in full process context at kernel mode to deliver data and status to the process. If the original request specified a user mode AST, the kernel mode AST queues it.
- When the user process gains control, the user's AST routine executes in full process context in user mode.

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

1.4 DEVICE DEPENDENCE AND DEVICE INDEPENDENCE

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

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

INTRODUCTION TO DEVICE DRIVERS

- The VAX/VMS operating system includes a number of routines that drivers can call to perform I/O preprocessing, allocate and deallocate resources, and synchronize driver execution.
- VAX/VMS I/O postprocessing performs the device-independent I/O postprocessing for all I/O requests.

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

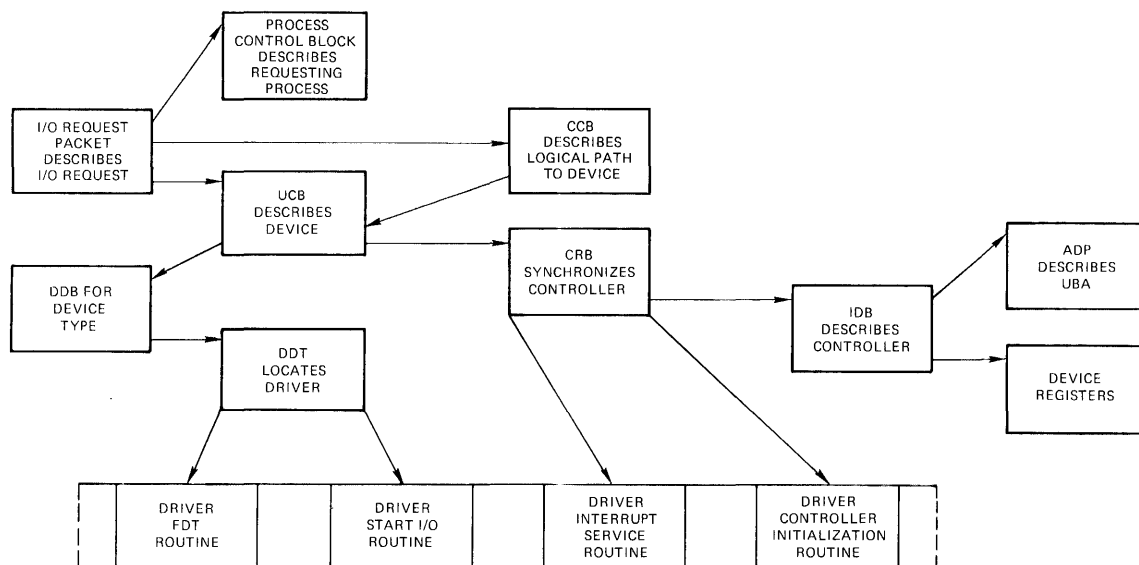
1.5 THE I/O DATA BASE

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

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

The three driver tables are defined in every driver. Section 1.1 lists these tables. Appendix A describes each of the control blocks and the I/O request packet in detail. The use of this information in driver processing is discussed throughout this manual.

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



MA 2429

Figure 1-2 The I/O Data Base

INTRODUCTION TO DEVICE DRIVERS

1.5.1 Control Blocks in the I/O Data Base

Control blocks in the I/O data base permit access to and describe peripheral hardware. The VAX/VMS operating system creates these control blocks either at system start-up or at the time a user-written driver is loaded into the system. Drivers refer to some or all of the following control blocks:

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

1.5.1.1 Device Data Block - A device data block contains information common to all devices of the same type that are connected to a particular controller. It records the generic device name concatenated with the controller designator, and the driver name and location for those devices. In addition, the device data block contains a pointer to the first unit control block for the device units attached to the controller.

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

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

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

INTRODUCTION TO DEVICE DRIVERS

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

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

1.5.2 I/O Request Packets

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

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

1.6 SYNCHRONIZATION

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

1.6.1 Interrupt Priority Levels

The VAX-11 processor defines 32 interrupt priority levels (0 through 31). The higher numbered IPLs are reserved for hardware interrupts, for example, device interrupts. The operating system uses the lower numbered IPLs. A higher IPL always takes precedence over a lower IPL. The VAX-11/780 Hardware Handbook describes the VAX-11 processor's use of IPLs. The following IPLs are of particular interest to drivers:

- Hardware device IPLs (20 through 23); driver interrupt service routines execute at these IPLs.
- Driver fork processing IPLs (8 through 11); driver fork processes execute at these IPLs.

INTRODUCTION TO DEVICE DRIVERS

- I/O completion IPL (IPL 4); VAX/VMS gains control to begin its device-independent I/O postprocessing at this IPL.
- AST delivery IPL (IPL 2); VAX/VMS uses this IPL to coordinate the delivery of an AST to a user process. The Queue I/O Request system service also executes at this IPL.

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

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

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

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

1.6.2 Fork Queues

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

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

1.6.3 Resource Wait Queues

Drivers compete for the following shared resources:

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

INTRODUCTION TO DEVICE DRIVERS

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

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

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

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

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

1.7 FUNCTIONS OF A DEVICE DRIVER

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

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

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

INTRODUCTION TO DEVICE DRIVERS

1.7.1 Initialization Routines

Most device controllers and device units require initialization when the VAX/VMS driver loading procedure loads the driver into memory and during recovery from a power failure. The amount and type of initialization varies from device type to device type. Section 13.1 provides additional initialization information.

1.7.2 FDT Routines

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

The driver contains FDT routines that are device-dependent. VAX/VMS provides additional FDT routines that perform processing common to many I/O functions, as described in Section 8.5. It is advisable for drivers to use FDT routines supplied by the operating system whenever possible.

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

1.7.3 Start I/O Routine

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

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

The start I/O routine may be forced to wait for the controller or UNIBUS resources to become available. In either case, VAX/VMS suspends the routine and reactivates it when the resources are free. Section 1.6.3 describes the context that VAX/VMS saves for the suspended routine.

After activating the device, the start I/O routine invokes the VAX/VMS wait for interrupt macro. The wait for interrupt macro suspends the

INTRODUCTION TO DEVICE DRIVERS

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

1.7.4 Interrupt Service Routine

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

1.7.5 Device Timeout Handler

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

Once every second, the VAX/VMS system timer checks all devices in the system for device timeout. When it locates a device that has timed out, it calls the timeout handler. Like the driver's I/O completion function, the timeout handler gains control in interrupt context.

1.7.6 Cancel I/O Routine

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

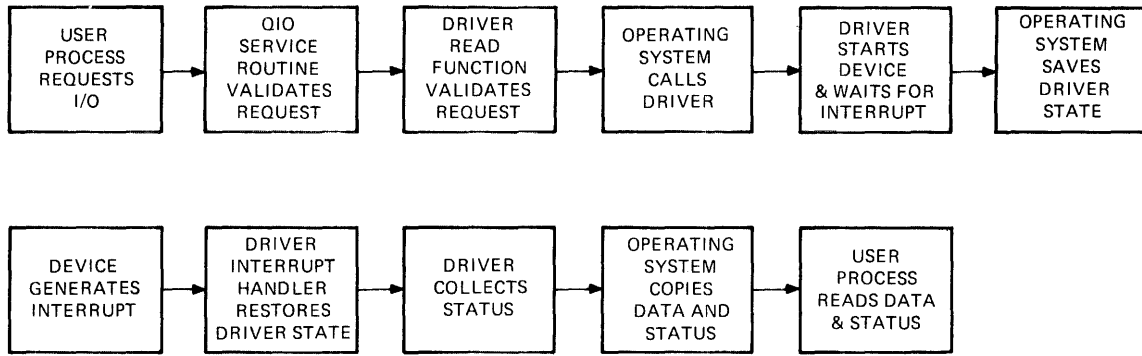
1.7.7 Error-logging Routine

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

1.8 AN EXAMPLE OF A UNIBUS I/O REQUEST

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

INTRODUCTION TO DEVICE DRIVERS



MA-2410

Figure 1-3 Processing a Sample I/O Operation

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

- **A process requests I/O operation.** A user process requests data from the device by issuing either of the following:
 - A VAX-11 RMS get record function call (which results in a Queue I/O request)
 - A Queue I/O Request system service

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

- **The operating system performs I/O preprocessing.** The Queue I/O Request system service validates the request and locates I/O data base control blocks that describe the device and its driver. The system service also allocates and initializes an I/O packet to contain a description of the I/O request. The system service then calls a read function routine in the driver.
- **The driver performs I/O preprocessing.** The driver function decision table routine verifies that the user buffer resides in virtual memory pages that can be modified by the requesting process, locks the buffer pages in memory, and adds details of the I/O operation to the I/O request packet. The read FDT routine then calls the operating system to send the I/O request packet to the driver.
- **VAX/VMS creates a driver fork process.** A VAX/VMS routine creates a fork process in which the device driver can execute. The routine activates the driver fork process by transferring control to the driver's start I/O routine.
- **The driver readies the UNIBUS adapter.** For DMA transfers, the driver fork process calls VAX/VMS routines that control the UNIBUS adapter hardware to map UNIBUS addresses into physical addresses for the transfer.
- **The driver activates the device.** The fork process activates the device by setting bits in device registers.
- **The driver waits for an interrupt.** A VAX/VMS routine saves the context of the driver fork process and relinquishes the processor until an interrupt occurs.

INTRODUCTION TO DEVICE DRIVERS

- **The device requests an interrupt.** When the data transfer is complete, the device requests a hardware interrupt that causes the system to dispatch to the driver's interrupt service routine.
- **The driver services the interrupt.** The driver's interrupt service routine handles the interrupt and reactivates the driver, which reads device registers to obtain status information about the transfer.
- **The operating system inserts the driver in a fork queue.** The driver requests that the process be reactivated at a lower software interrupt priority level.
- **The fork dispatcher reactivates the driver fork process.** When processor priority permits, the VAX/VMS fork dispatcher reactivates the driver as a fork process.
- **The driver completes the I/O operation.** The driver fork process completes device-dependent I/O processing of the I/O request and returns the I/O status to VAX/VMS.
- **VAX/VMS completes the I/O operation.** The VAX/VMS I/O postprocessing routines copy the I/O status into process address space and/or general registers and return control to the user process.

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

The 13-step example condenses and simplifies the processing of an I/O operation by ignoring such issues as the following:

- Association of a device with a process; that is, device assignment
- Simultaneous I/O requests for one device
- Hardware interrupt priority levels
- Driver competition for shared system and UNIBUS adapter resources
- Driver competition for I/O activity through a multiunit controller
- Driver recovery from device errors or power failure

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

1.9 THE UNIBUS

On a VAX-11 system, the internal processor bus (that is, the synchronous back plane interconnect) connects the central processor to memory. The internal processor bus also connects the UNIBUS adapter and MASSBUS adapter (MBA) to memory and to the central processor. Peripheral devices attach to either the UNIBUS, for UNIBUS devices, or the MASSBUS, for MASSBUS devices, as illustrated in Figure 1-4.

INTRODUCTION TO DEVICE DRIVERS

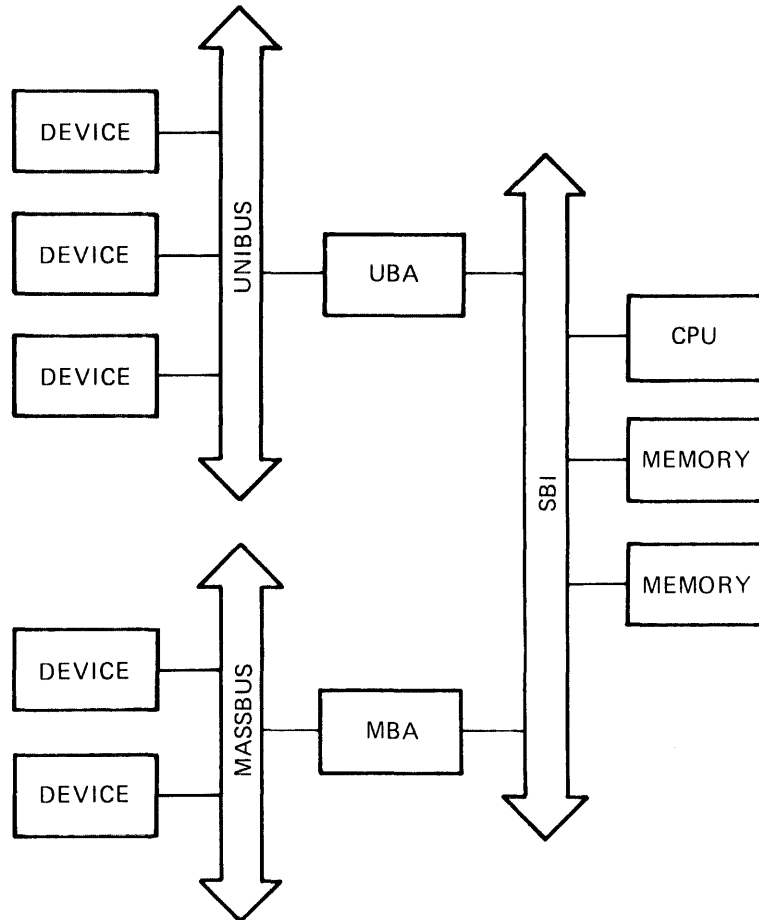


Figure 1-4 VAX-11 Hardware Configuration

The VAX-11/780 Hardware Handbook describes the hardware components diagrammed in Figure 1-4.

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

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

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

The direct data path maps each UNIBUS transfer to an SBI transfer. For each UNIBUS transfer, there is one SBI transfer. Each SBI operation transfers a single word or byte of data depending on the device. A buffered data path, on the other hand, allows a quadword of data to be assembled and transferred in one SBI operation. Up to eight UNIBUS transfers occur for each SBI transfer.

INTRODUCTION TO DEVICE DRIVERS

Drivers performing nonDMA transfers are generally not concerned with UNIBUS adapter operation.¹

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

Devices transfer data using one of the following methods:

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

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

Devices that perform DMA transfers do not require the central processor so frequently. Once the driver activates the device, the device can transfer a large amount of data without requesting an interrupt after each of the smaller amounts. Normally, the driver of a DMA device allocates a UNIBUS buffered data path and UNIBUS map registers for I/O transfers.

1.11 BUFFERED I/O AND DIRECT I/O

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

- Buffered I/O
- Direct I/O

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

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

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

1. Instead of creating a complete device driver for a nonDMA device, you can connect the process to the device interrupt vector to program the device from a user process. For a description of how and when to connect to an interrupt vector, consult the VAX/VMS Real-Time User's Guide.

INTRODUCTION TO DEVICE DRIVERS

1.12 LOADABLE DRIVERS

The VAX/VMS operating system provides a procedure that allows a suitably privileged user to load drivers into a running VAX/VMS system. The SYSGEN utility, described in full in the VAX/VMS System Manager's Guide, supports commands that invoke the driver loading procedure:

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

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

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

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

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

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

CHAPTER 2

DISCUSSION OF A LINE PRINTER QUEUE I/O REQUEST

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

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

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

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

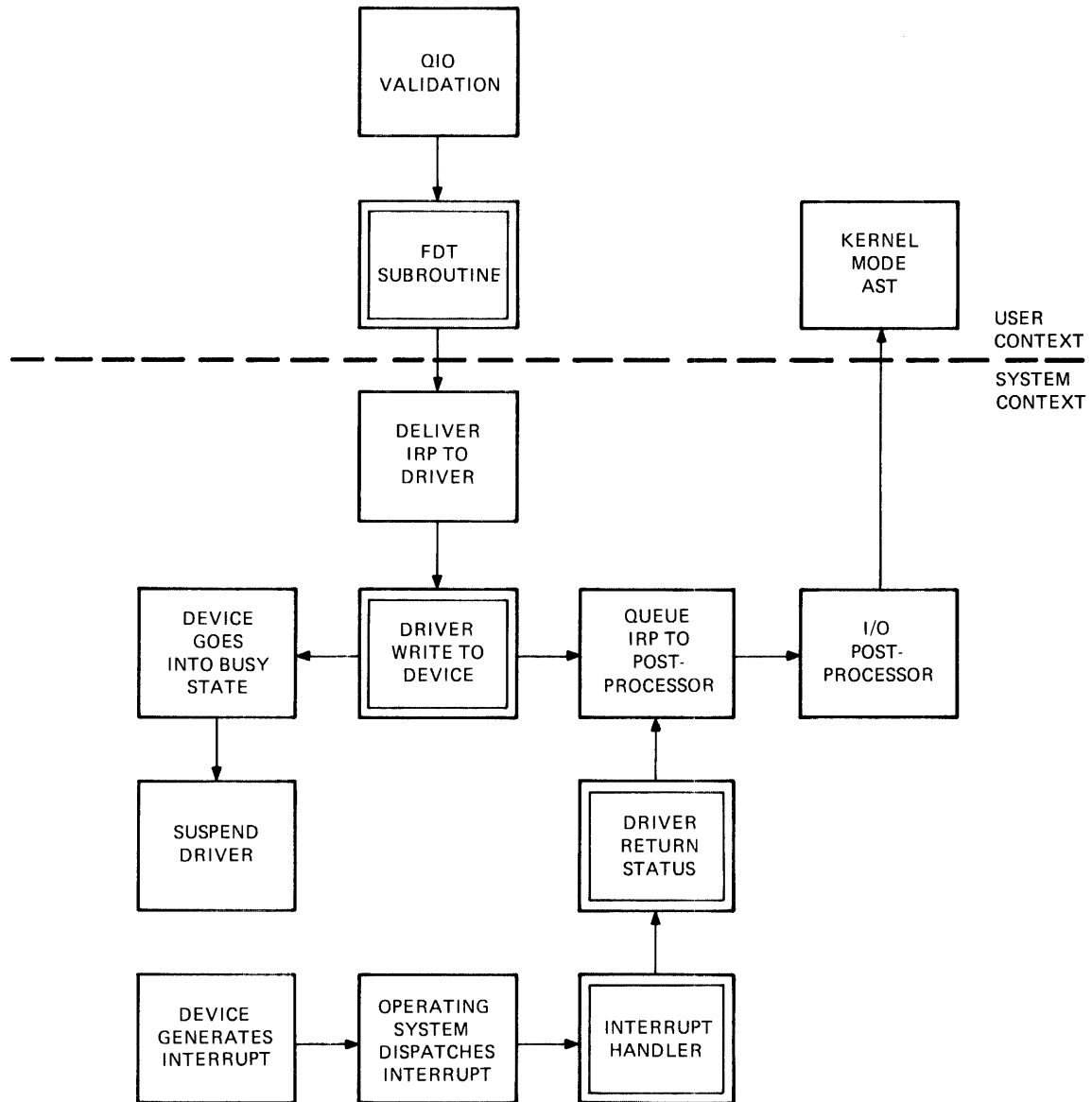
The LP11 was selected for this discussion because it is a simple driver but still illustrates many driver principles. Although the LP11 is usually spooled, for purposes of this discussion, assume that it is not spooled.

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

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

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

DISCUSSION OF A LINE PRINTER QUEUE I/O REQUEST



MA-2421

Figure 2-1 A Line Printer Write Function

2.1 DRIVER CODE FOR THE LP11 WRITE FUNCTION

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

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

DISCUSSION OF A LINE PRINTER QUEUE I/O REQUEST

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

2.2 A USER PROCESS'S I/O REQUEST

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

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

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

2.3 I/O PREPROCESSING BY VAX/VMS

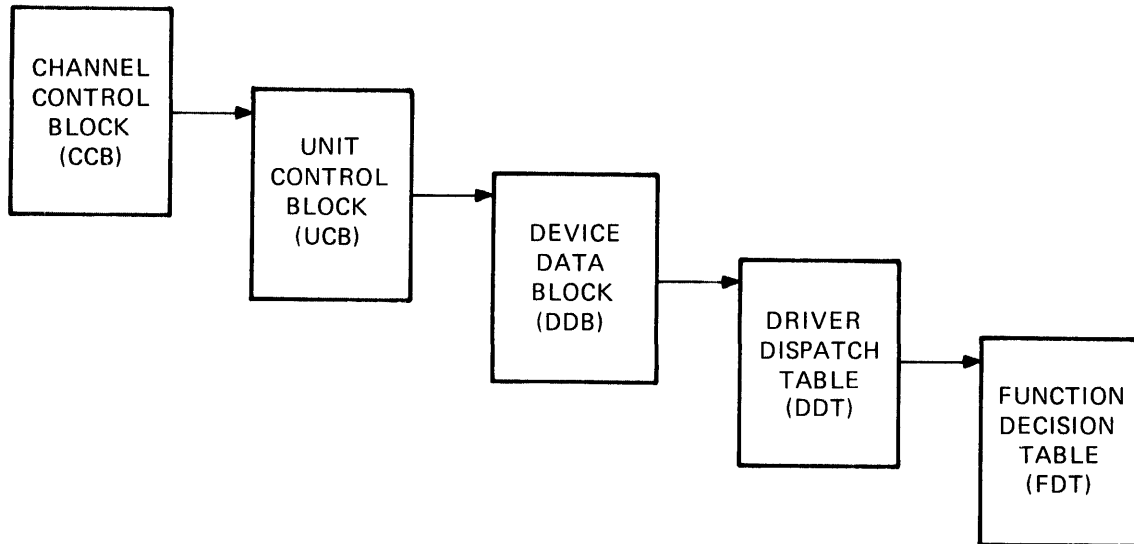
When called, the Queue I/O Request system service first validates that the I/O request is correctly specified; that is, the I/O request must meet the following criteria:

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

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

- The line printer FDT must list IO\$ WRITEVBLK as a valid function for the device.
- The event flag number must be valid.
- The process buffered I/O request quota must permit the Queue I/O Request system service to perform a buffered I/O request without exceeding the process's quotas.
- The process must have write access to the user-specified location to be used as an I/O status block.

DISCUSSION OF A LINE PRINTER QUEUE I/O REQUEST



MA-2419

Figure 2-2 Locating a Function Decision Table

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

If the target device for the I/O request is not file-structured, the Queue I/O Request system service changes any virtual function code to its logical equivalent when it builds the I/O request packet. Thus, for a line printer device, `IO$WRITEVBLK` is translated to `IO$WRITELBLK`. User-written drivers should check that virtual function codes have corresponding logical codes.

2.4 I/O PREPROCESSING BY THE DRIVER

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

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

- It calculates the length of the required system space buffer.
- If the process byte count quota for buffered I/O (`BYTCNT`) permits, the routine allocates a buffer from system address space, stores the address of the buffer in the I/O request packet, and decreases the current process byte count quota.

DISCUSSION OF A LINE PRINTER QUEUE I/O REQUEST

- It then synchronizes with other possible subprocesses¹ to read and write fields of the line printer's unit control block.
- It reads the description of the line printer's current line and page position from the device's unit control block.
- It reformats the data from the process buffer into the system buffer, adding carriage control characters, as specified in the I/O request argument P4, before and after the data.

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

- It rewrites updated line and page positions into the device's UCB. This information indicates what the current location on the page being printed will be where the request completes.
- Finally, the routine transfers control to a VAX/VMS routine that queues the I/O packet to the device driver.

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

2.5 QUEUING THE I/O PACKET TO THE DRIVER

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

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

UCB → DDB → DDT → Entry point to start I/O routine

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

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.

DISCUSSION OF A LINE PRINTER QUEUE I/O REQUEST

2.6 DRIVER DEVICE ACTIVATION

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

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

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

UCB → CRB → IDB → CSR address

In contrast to many other devices, such as disks, the LP11 line printer does not share a controller with other devices. Therefore, no arbitration for ownership of the controller is required. The CSR address is always the address of the line printer control/status register, and all other device registers are at fixed offsets from this address.

2. The driver examines the device's control/status register to see if the device is ready to accept characters.
3. If the device is ready, the driver writes a byte of data into the line printer data buffer and decreases the count of bytes to transfer. It then repeats step 2.
4. If the device is not ready, that is, if the device's internal buffer is full, the driver raises IPL to 31 to block out all interrupts and sets the interrupt enable bit in the device's control/status register.

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

2.7 WAITING FOR A DEVICE INTERRUPT

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

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

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

DISCUSSION OF A LINE PRINTER QUEUE I/O REQUEST

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

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

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

2.8 INTERRUPT HANDLING

When the LP11 line printer requests a hardware interrupt, the UNIBUS adapter interrupt service routine gains control. The service routine reads the device vector from a UNIBUS adapter register and passes the interrupt to the LP11 driver interrupt service routine.

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

- Confirms that the interrupt was expected by examining bits in the device's unit control block
- Restores the saved registers (R3 and R4) from the device's unit control block
- Restores the address of the unit control block in R5
- Transfers control to the driver PC address stored in the device's unit control block

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

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

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

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

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

DISCUSSION OF A LINE PRINTER QUEUE I/O REQUEST

2.9 I/O COMPLETION PROCESSING BY THE DRIVER

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

- It obtains the number of characters left to transfer from the unit control block.
- It transfers characters until the LPl1 again prints its data buffer or all characters have been transferred.
- When all characters have been transferred, the driver code branches to driver I/O completion code.

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

- A success status code
- The number of bytes transferred

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

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

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

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

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

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

CHAPTER 3
SYNCHRONIZATION OF I/O REQUEST PROCESSING

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

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

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

3.1 INTERRUPT PRIORITY LEVELS

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

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

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

3.1.1 IPLs Defined by VAX/VMS

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

SYNCHRONIZATION OF I/O REQUEST PROCESSING

Table 3-1
IPLs Defined by VAX/VMS

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

3.1.2 IPLs Defined for the Hardware

Hardware interrupt levels are used for device interrupts (IPLs 20 through 23) and urgent conditions including power failure and serious errors such as a machine check. The VAX-11/780 Hardware Handbook provides additional information about hardware interrupt levels.

3.1.3 Interrupt Service Routines

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

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

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

SYNCHRONIZATION OF I/O REQUEST PROCESSING

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

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

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

3.1.4 Raising IPL

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

3.1.5 Lowering IPL

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

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

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

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

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

SYNCHRONIZATION OF I/O REQUEST PROCESSING

3.1.6 Servicing Hardware Interrupts

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

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

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

Transferring control to the interrupt service routine requires that the processor first transfer control to the interrupt dispatcher for the adapter (MASSBUS or UNIBUS) to which the device is attached. The interrupt dispatcher is itself an interrupt service routine. The dispatcher locates the channel request block (CRB) for the device and transfers control to the dispatching field. The channel request block contains a JSB instruction that, in turn, transfers control to the driver's interrupt service routine. When the interrupt service routine issues an REI instruction, the code executing prior to the interrupt resumes unless interrupts occurred at levels between that of the executing interrupt service routine and the interrupted code.

Figure 3-1 illustrates the steps performed by the hardware and VAX/VMS to transfer control to a driver interrupt service routine after a UNIBUS device interrupt.

3.1.7 Transferring Control to the Driver Fork Process

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

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

SYNCHRONIZATION OF I/O REQUEST PROCESSING

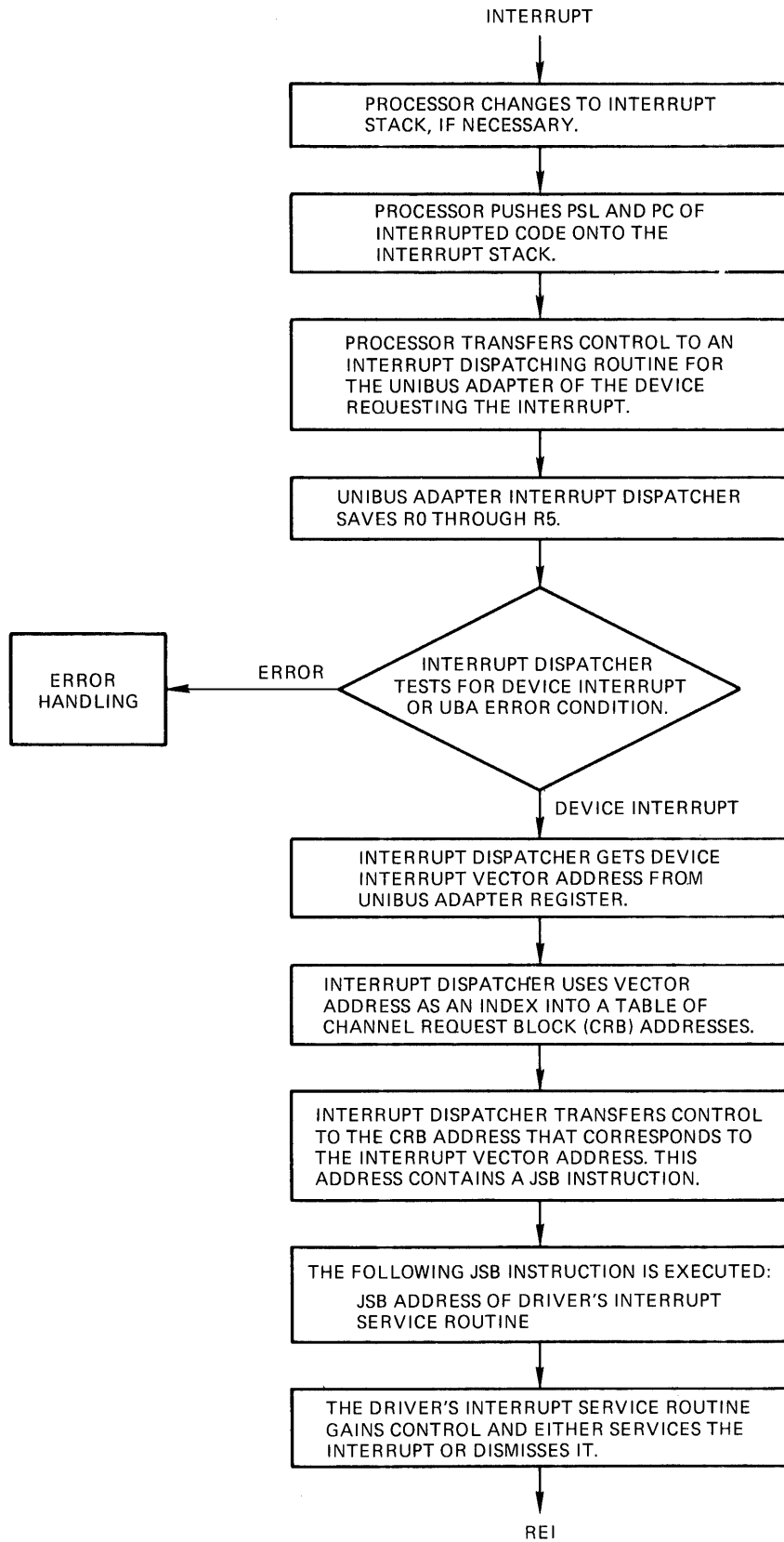


Figure 3-1 Servicing Hardware Interrupts

SYNCHRONIZATION OF I/O REQUEST PROCESSING

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

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

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

3.1.8 IPL Use During I/O Processing

I/O processing occurs mainly at the following IPLs:

- IPL\$ASTDEL (IPL 2)
- IPL\$IOPOST (IPL 4)
- Driver fork processing IPLs (IPLs 8 through 11)
- Hardware device IPLs (IPLs 20 through 23)
- IPL\$POWER (IPL 31)

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

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

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

SYNCHRONIZATION OF I/O REQUEST PROCESSING

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

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

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

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

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

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

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

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

3.1.8.5 IPL\$ POWER - The highest IPL, IPL\$ POWER, locks out all other interrupts. Many VAX/VMS routines and drivers raise IPL to IPL\$ POWER to execute code sequences that cannot tolerate interruption. For example, much of system initialization occurs at IPL\$ POWER.

SYNCHRONIZATION OF I/O REQUEST PROCESSING

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

3.1.9 Additional IPLs

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

- IPL\$ SCHED (IPL 3); never used by drivers
- IPL\$ QUEUEAST (IPL 6); very seldom used by drivers
- IPL\$ SYNCH and IPL\$ TIMER (IPL 7); very seldom used by drivers
- IPL\$ MAILBOX (IPL 11); very seldom used by drivers

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

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

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

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

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

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

3.1.9.3 IPL\$ SYNCH and IPL\$ TIMER - IPL\$ SYNCH is the system data base synchronization level. When a VAX/VMS subroutine or a driver needs to modify or read a dynamic portion of the system data base, the routine always executes at IPL\$ SYNCH to ensure that the data base does not change due to some interrupt service routine or process action. For example, the driver loading procedure invoked by the SYSGEN utility

SYNCHRONIZATION OF I/O REQUEST PROCESSING

raises IPL to IPL\$SYNCH before adding control blocks to the I/O data base.

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

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

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

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

3.1.10 Overview of IPL Use

Figure 3-2 illustrates the normal IPL flow during the processing of an I/O request.

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

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

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

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

SYNCHRONIZATION OF I/O REQUEST PROCESSING

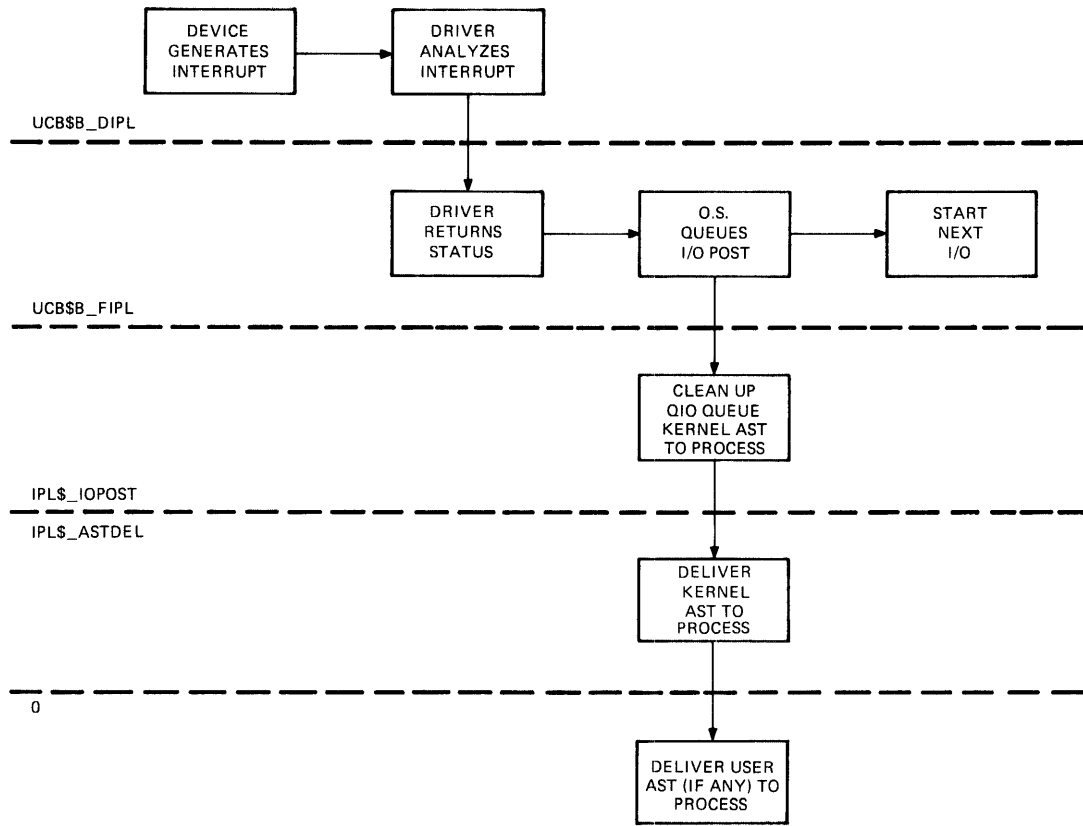


Figure 3-2 IPL Conventions During I/O Processing

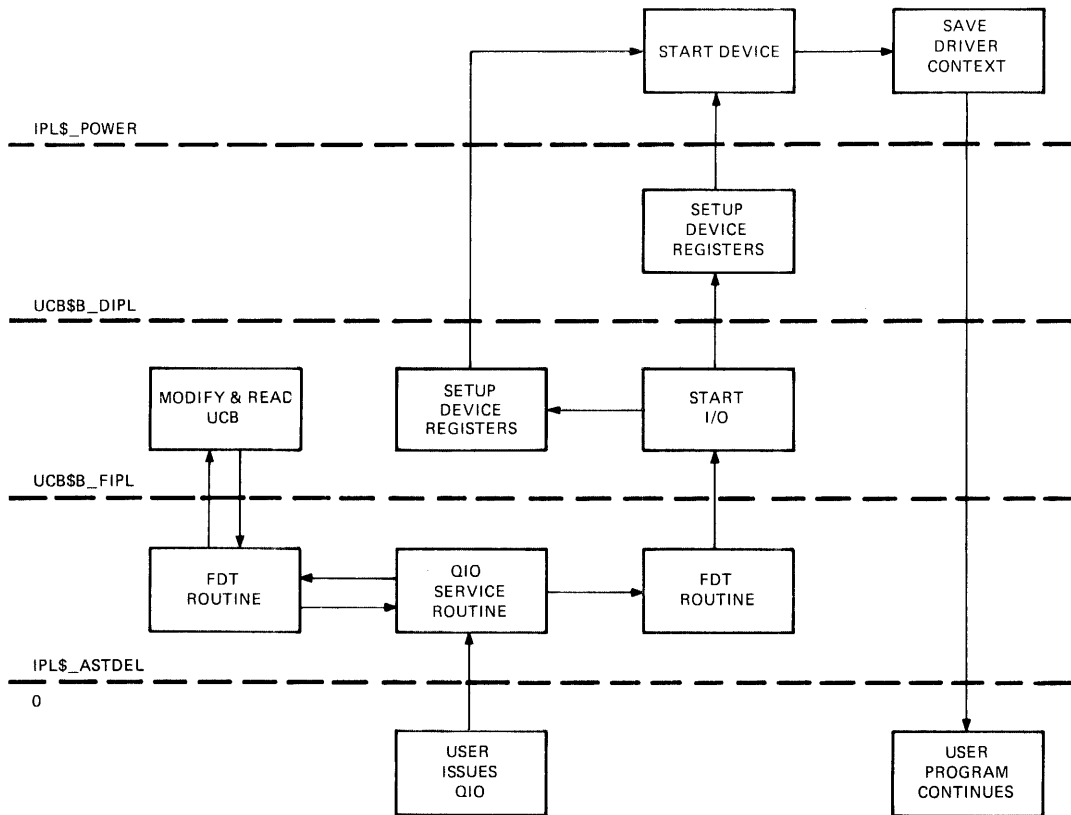


Figure 3-3 IPL Conventions During I/O Completion

SYNCHRONIZATION OF I/O REQUEST PROCESSING

The I/O postprocessing routine adjusts process quota usage and deallocates system buffers for write functions at IPL\$ IOPOST. The routine also calls another VAX/VMS routine that raises IPL to IPL\$ SYNCH to queue a kernel mode AST to the process that issued the original QIO request. The AST routine executes at IPL\$ ASTDEL, and may queue a user AST routine that eventually executes at an IPL of 0. I/O postprocessing continues at IPL\$ IOPOST until all entries in the postprocessing queue have been serviced.

3.1.11 Modifying IPL in Driver Code

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

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

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

- SETIPL
- DSBINT
- ENBINT
- SOFTINT

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

Format

```
SETIPL [ipl]
```

ipl

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

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

Format

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


SYNCHRONIZATION OF I/O REQUEST PROCESSING

ipl

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

dst

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

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

Format

ENBINT [src]

src

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

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

Format

SOFTINT ipl

ipl

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

3.2 FORK BLOCKS AND FORK DISPATCHING

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

SYNCHRONIZATION OF I/O REQUEST PROCESSING

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

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

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

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

3.2.1 Interrupt Service Routine for Fork Dispatching

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

Figure 3-4 illustrates the fork queue structure.

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

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

SYNCHRONIZATION OF I/O REQUEST PROCESSING

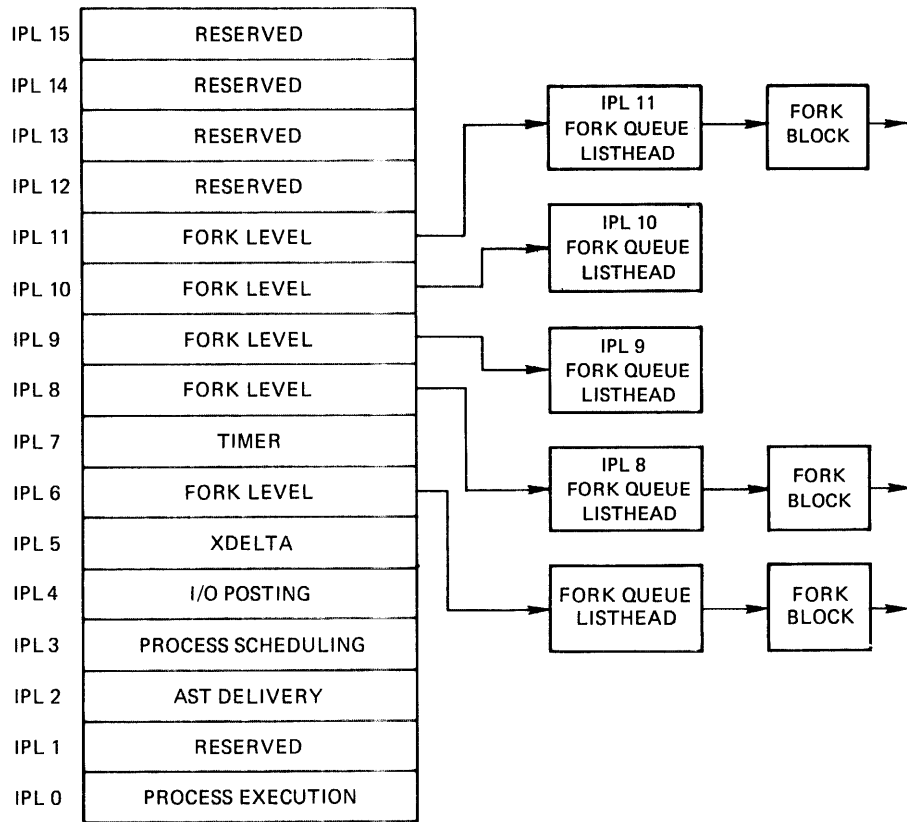


Figure 3-4 Fork Dispatching Data Structure

3.3 RESOURCE WAIT QUEUES

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

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

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

SYNCHRONIZATION OF I/O REQUEST PROCESSING

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

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

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

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

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

3.3.1 Competing for a Controller Data Channel

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

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

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

- Retrieve the virtual address of the data to be written and the number of bytes to transfer from the device's unit control block
- Retrieve the virtual address of the device's control/status register from the interrupt data block

SYNCHRONIZATION OF I/O REQUEST PROCESSING

- Calculate the address of the line printer's data buffer register by adding a constant offset to the control/status register address
- Write data one byte at a time to the line printer's data buffer until all bytes of data have been written

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

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

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

If the channel is busy, the VAX/VMS routine saves the driver fork context in the UCB fork block and inserts the fork block address in the controller channel wait queue.

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

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

CHAPTER 4

THE UNIBUS ADAPTER

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

- Arbitrates priority interrupts from UNIBUS devices
- Delivers interrupts from UNIBUS devices to the processor
- Allows drivers to gain access to UNIBUS device registers using system virtual addresses
- Translates 18-bit UNIBUS addresses to 30-bit SBI byte addresses
- Provides a data transfer path to randomly ordered physical SBI addresses, that is, to discontinuous pages
- Provides buffered data transfer paths to consecutively increasing physical SBI addresses
- Permits byte-aligned buffers for UNIBUS devices requiring word-aligned buffer addresses

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

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

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

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

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

THE UNIBUS ADAPTER

4.1 READING AND WRITING DEVICE REGISTERS

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

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

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

4.2 MAPPING UNIBUS AND SBI ADDRESSES FOR DMA TRANSFERS

The UNIBUS address space consists of 256K bytes of memory, of which 8K bytes are reserved for device control registers. UNIBUS DMA devices read and write data from and to memory locations using 18-bit UNIBUS addresses. The UNIBUS adapter translates the 18-bit UNIBUS addresses into 30-bit SBI addresses. This translation allows the operating system, I/O drivers, and UNIBUS devices to access the same physical address space.

The UNIBUS adapter provides 496 map registers to translate UNIBUS addresses to SBI addresses. Each map register represents one page of the UNIBUS address space. A 21-bit field in the map register identifies the SBI page frame number corresponding to the UNIBUS address that the map register represents.

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

- The 9-bit UNIBUS page address field (bits 9 through 17 of the UNIBUS address) identifies the UBA map register.
- The 21-bit SBI page frame number field (bits 0 through 20) in the map register identifies bits 27 through 7 of the SBI address.
- UNIBUS address bits 2 through 8 map directly to bits 0 through 6 of the SBI address.

The resulting 28-bit SBI address locates the SBI longword that is the target of the transfer. The UNIBUS adapter identifies the byte addressed within the longword by interpreting the low-order two bits of the UNIBUS address.

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

THE UNIBUS ADAPTER

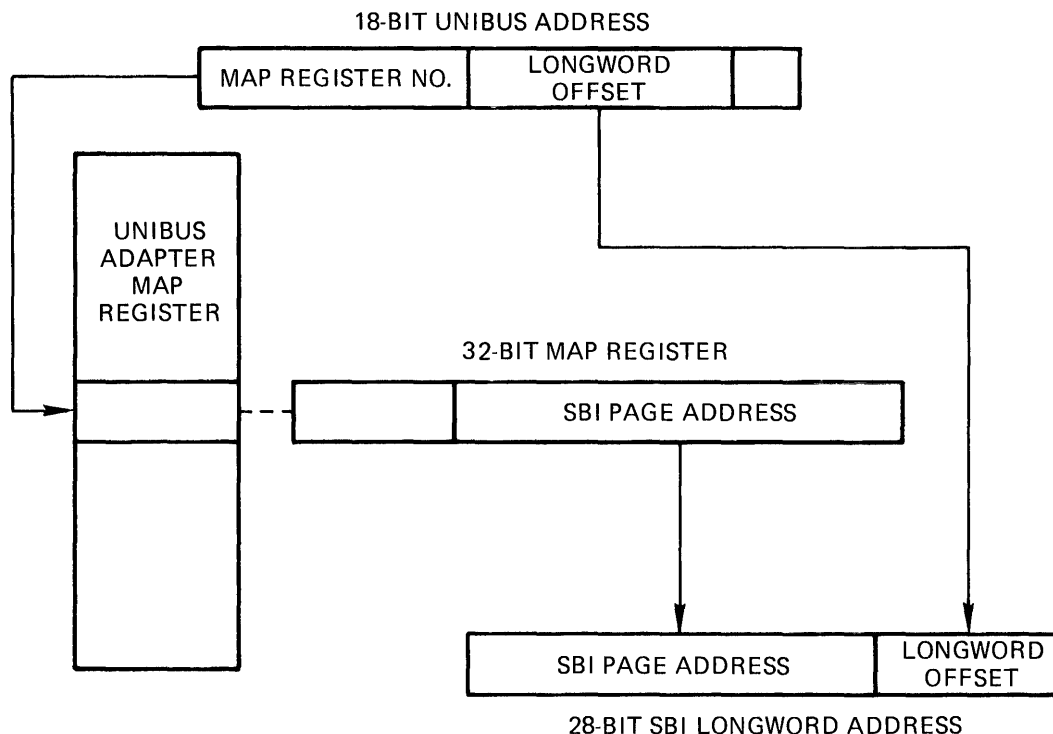


Figure 4-1 UNIBUS to SBI Address Mapping

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

4.2.1 UNIBUS Adapter Data Transfer Paths

The UNIBUS adapter sends data through one of 16 data paths for UNIBUS devices performing DMA transfers. One data path, the direct data path (DDP), allows UNIBUS transfers to randomly ordered SBI addresses. The direct data path maps each UNIBUS transfer to an SBI transfer. Thus, a single word or byte of data is transferred per SBI operation.

The other 15 data paths, the buffered data paths (BDPs), allow sequential access devices on the UNIBUS to transfer to consecutively increasing addresses much faster than through the direct data path. The buffered data paths store data from the UNIBUS until a quadword of data has been assembled. Then the UNIBUS adapter begins an SBI transfer.

Buffered data paths also allow a UNIBUS device to transfer randomly ordered, longword-aligned 32-bit data. The longword-aligned transfer mode is discussed in Section 4.2.1.5.

When a UNIBUS device begins a DMA transfer by placing an address on the UNIBUS, the UNIBUS adapter map register not only performs address mapping but also provides the number of the data path to be used for the transfer. Each UNIBUS adapter map register contains a 4-bit field that describes the data path. Data path 0 is the direct data path, and data paths 1 through 15 are the buffered data paths.

THE UNIBUS ADAPTER

The sequence below describes a UNIBUS device DMA transfer.

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

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

The direct data path is slower than buffered data paths because each UNIBUS transfer cycle corresponds to an SBI cycle. Throughput is one word or byte transferred per SBI cycle, which is approximately a .8 megabyte per second maximum. The direct data path is also unable to transfer a word of data to an odd SBI address. Therefore, an FDT routine for a DMA device that uses the direct data path might check that the specified buffer is on a word boundary.

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

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

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

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

THE UNIBUS ADAPTER

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

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

4.2.1.2 Buffered Data Paths - In contrast to the direct data path, the 15 buffered data paths transfer data much more efficiently between the UNIBUS and the SBI by decoupling the UNIBUS transfer from the SBI transfer. Buffered data paths read or write 32 or 64 bits of data in a transfer, and buffer the unrequested portions of the data in UBA buffers. Thus, as many as four separate UNIBUS read functions can be accommodated with a single SBI transfer.

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

- Fast DMA block transfers to or from consecutively increasing addresses (maximum 1.39 megabyte per second transfer rate)
- Word-oriented block transfers that begin and end on an odd byte of SBI memory; note, however, that these transfers can be quite slow since the UNIBUS adapter may need to transfer two quadwords to complete a 1-word transfer
- 32-bit data transfers from random longword-aligned SBI addresses

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

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

- The driver must request the UNIBUS adapter to purge the buffered data path after each block transfer (except for error-free longword-aligned transfers).
- All addresses in a block transfer must be consecutively increasing addresses.
- All transfers within a block must be of the same function type (DATI or DATO/DATOB).

A buffered data path buffers data from the UNIBUS until a quadword of data has been transferred (except in longword-aligned transfer mode; see below). Then, the UNIBUS adapter transfers the contents of the buffer to the appropriate SBI address in a single SBI operation. The

THE UNIBUS ADAPTER

procedure for a UNIBUS write operation that transfers a quadword of data to memory is broken into individual steps below.

- The UNIBUS device transfers one word of data to the buffered data path.
- The buffered data path stores the word of data and completes the UNIBUS cycle.
- The buffered data path sets its buffer-not-empty (BNE) bit to indicate that the buffer contains valid data.
- The UNIBUS device repeats the first three steps until the UNIBUS address is the last byte or word of a physical quadword.
- When the UNIBUS device addresses the last byte or word in a physical quadword, the UBA recognizes a complete data-gathering cycle.
- The buffered data path requests an SBI extended write function to write a quadword of data from the buffered data path to memory.
- When the SBI transfer is complete, the buffered data path clears its BNE bit to indicate that the buffer no longer contains valid data.

The procedure for a UNIBUS read function using a buffered data path includes a prefetch function. The prefetch automatically reads another quadword of data from the SBI after the contents of a buffered data path is transferred to the UNIBUS. The prefetch speeds up UNIBUS reads from SBI memory. The steps of a UNIBUS read function are listed below.

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

THE UNIBUS ADAPTER

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

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

4.2.1.4 Purging a Buffered Data Path - Since prefetches may read more data from SBI 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 SBI can complete with some data (less than a quadword) 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:

- Sets the BNE bit in the buffered data path register owned by the fork process. For a UNIBUS read function, the adapter simply clears the bit set by the subroutine. For a UNIBUS write function, the adapter transfers any data left in the data path buffer to VAX-11 memory, then clears the bit.
- Notifies the driver fork process of any error that occurs during the purge.

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

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

A longword-aligned transfer over a buffered data path is faster than a direct data path transfer and somewhat slower than a normal buffered data path transfer (maximum 1.17 megabyte per second transfer rate).

This longword-aligned use of the buffered data path disables the prefetch and makes normal purging of the data path unnecessary. If, however, the I/O transfer aborts, the driver must purge the data path.

THE UNIBUS ADAPTER

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

For a UNIBUS read, the function occurs as follows:

- The driver fork process initiates a read function on the UNIBUS device.
- The UNIBUS adapter clears the BNE bit in the assigned buffered data path.
- The UNIBUS adapter issues a read from SBI memory operation.
- The UNIBUS adapter stores the longword of data in the buffered data path and sets the BNE bit.
- The UNIBUS adapter initiates two UNIBUS read operations to transfer two words of data.
- When the two read operations are complete, the UBA clears the BNE bit.

For a UNIBUS write, the function occurs as follows:

- The driver fork process initiates a write function on the UNIBUS device.
- The UNIBUS adapter clears the BNE bit in the assigned buffered data path.
- The UNIBUS adapter issues two write operations to transfer two words of data from the UNIBUS device.
- The UNIBUS adapter stores the longword of data in the buffered data path and sets the BNE bit.
- The UNIBUS adapter initiates an SBI write operation.
- When the SBI write operation is complete, the UNIBUS adapter clears the BNE bit.

CHAPTER 5

OVERVIEW OF I/O PROCESSING

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

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

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

5.1 PREPROCESSING AN I/O REQUEST

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

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

OVERVIEW OF I/O PROCESSING

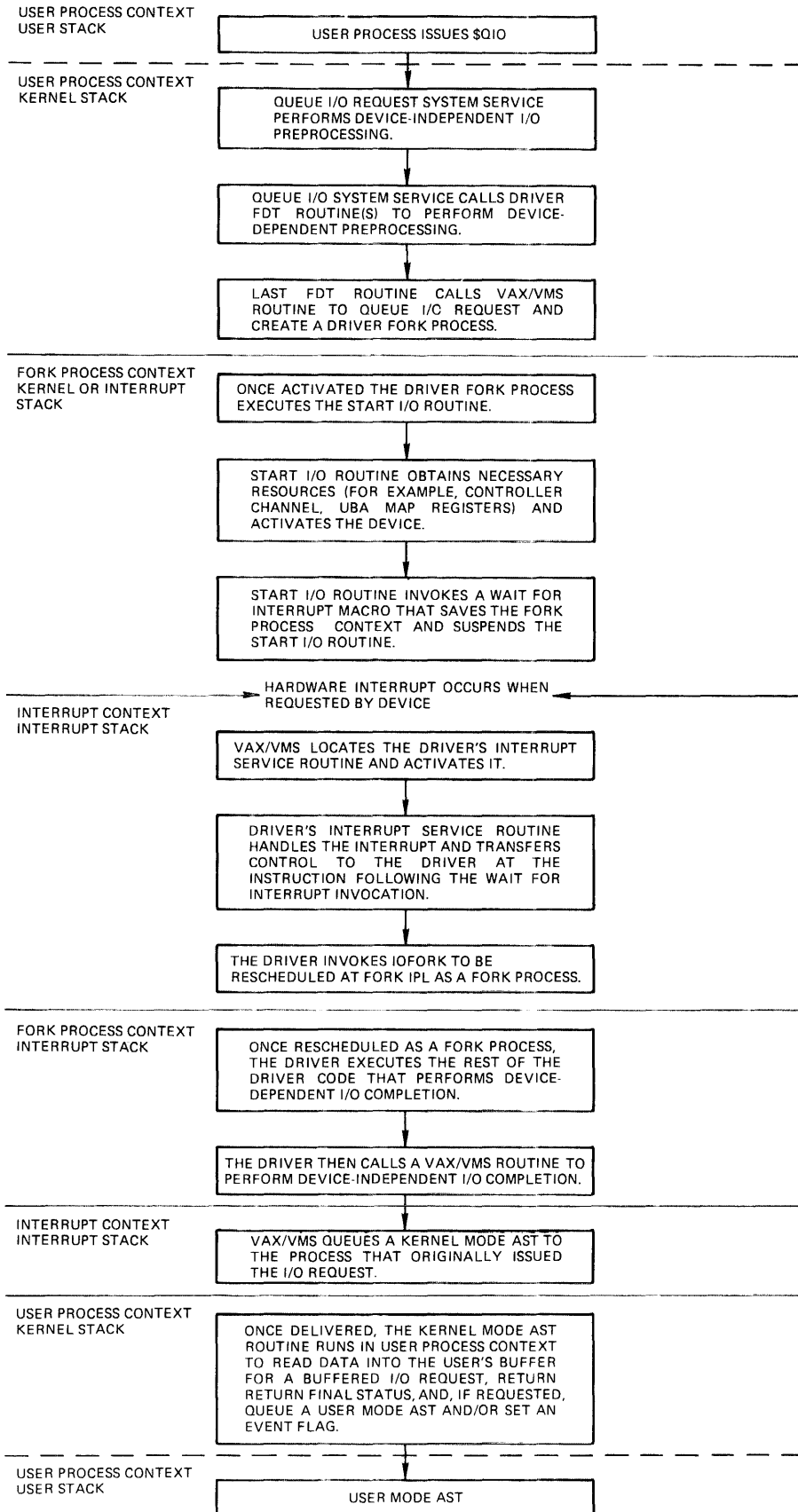


Figure 5-1 Sequence of Driver Execution

OVERVIEW OF 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 system service call. The Assign I/O Channel system service performs the following functions:

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

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

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

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

- Channel request block¹
- Interrupt data block
- Device data block

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

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

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

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

OVERVIEW OF I/O PROCESSING

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

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

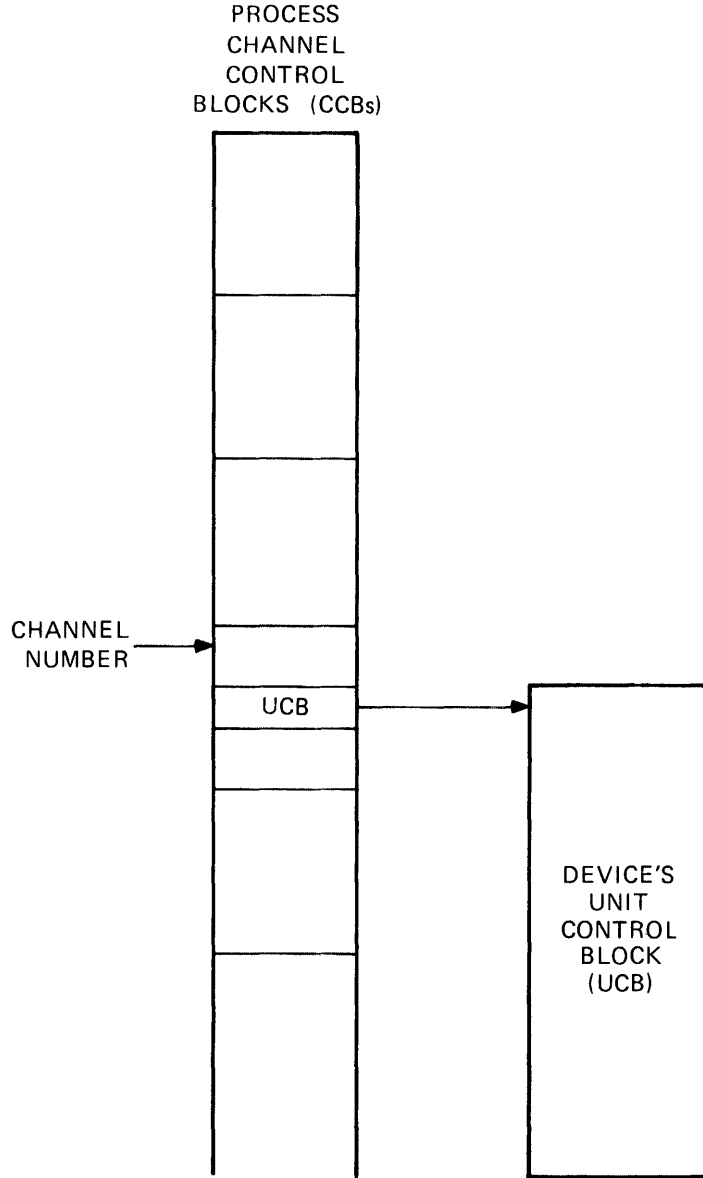


Figure 5-2 Locating the Target Device

OVERVIEW OF I/O PROCESSING

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

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

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

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

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

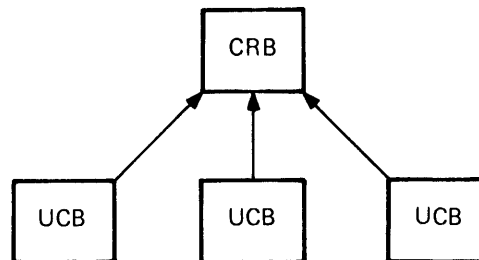


Figure 5-3 I/O Data Structures for Three Devices on One Controller

5.1.2.3 **Interrupt Data Block (IDB) and Device Data Block (DDB)** - The channel request block also points to an interrupt data block. The interrupt data base contains three critical data structure addresses:

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

OVERVIEW OF I/O PROCESSING

Finally, all unit controller blocks describing device units attached to a single controller contain a pointer to a single device data block (DDB). The device data block contains the following fields that identify the device and its driver:

- The generic device/controller name
- The name of the device's driver as obtained from the driver prologue table; see Chapters 7 and 14 for the use of the driver name
- A pointer to a driver dispatch table that lists the addresses of routines in the device driver

Figure 5-4 illustrates a pair of device data blocks describing a group of equivalent devices on two separate controllers.

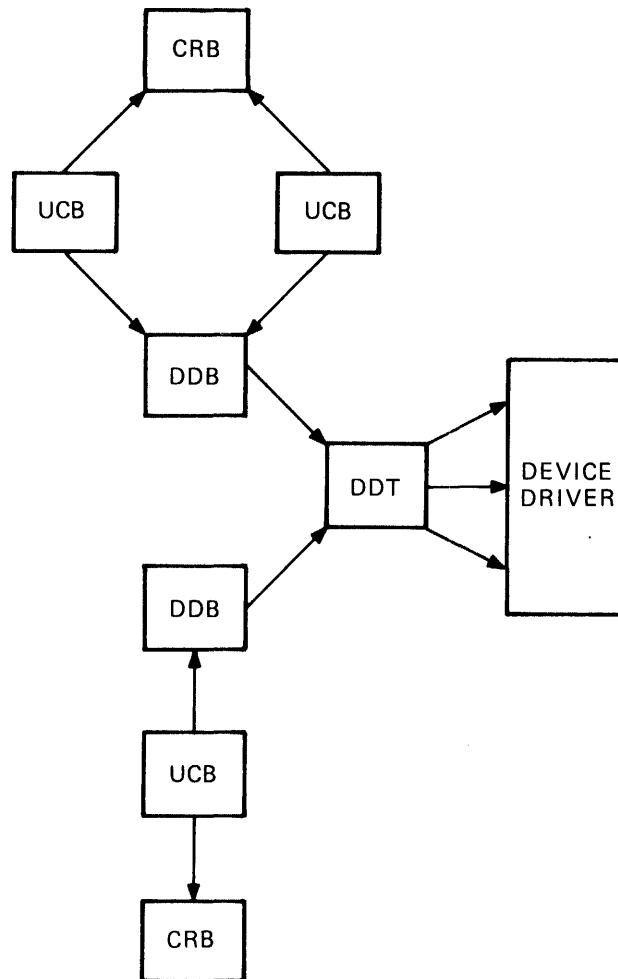


Figure 5-4 I/O Data Base for Two Controllers

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

OVERVIEW OF I/O PROCESSING

5.1.3 Validating the I/O Function

Using the I/O data structures described above, the Queue I/O Request system service locates the address of the driver's function decision table by following a chain of pointers beginning in the UCB of the target device for the I/O request, as follows:

UCB → DDB → DDT → FDT

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

5.1.4 Checking Process I/O Request Quotas

The Queue I/O Request system service calls a routine that determines whether the I/O request being readied will cause the process to exceed its quota for outstanding direct or buffered I/O requests. If the process remains under quota, the checking routine returns a success status to the service, allowing it to continue I/O preprocessing.

In the case where quota is exceeded, the routine examines the system service resource wait flag. If the flag is clear, the routine returns a quota exceeded status to the Queue I/O Request system service, which aborts the I/O request.

If the flag is set, the process is placed in a wait state until it drops below quota, at which time the quota checking routine returns success status to the service. Then, depending on the type of function, the system service decreases the process quota of remaining buffered or direct I/O operations.

5.1.5 Validating the I/O Status Block

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

5.1.6 Allocating and Setting Up an I/O Request Packet

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

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

The Queue I/O Request system service attempts to allocate an I/O request packet from a linked list of preallocated I/O request packets. If no preallocated packets exist, the service calls a VAX/VMS routine

OVERVIEW OF I/O PROCESSING

that allocates an I/O request packet from nonpaged pool. This allocating routine synchronizes with the rest of the system so that it can allocate the memory needed.

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

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

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

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

OVERVIEW OF I/O PROCESSING

5.1.7 Function Decision Table Processing

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

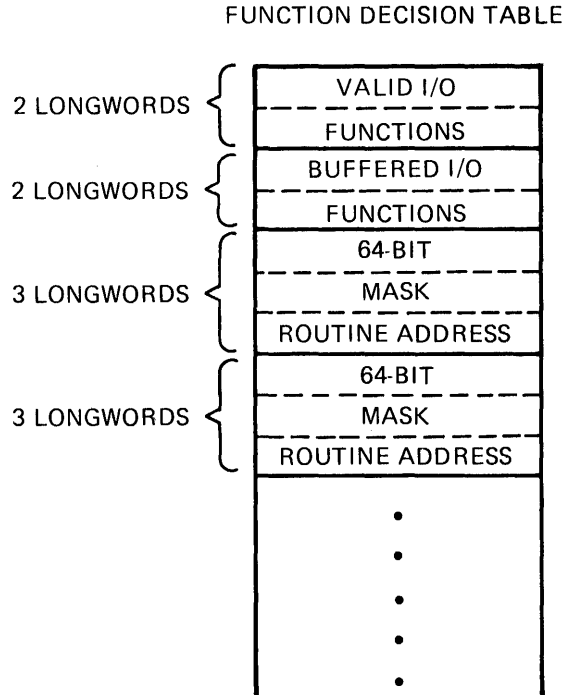


Figure 5-5 Driver Function Decision Table

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

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

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

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

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

OVERVIEW OF I/O PROCESSING

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

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

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

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

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

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

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

5.2 HANDLING DEVICE ACTIVITY

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

OVERVIEW OF I/O PROCESSING

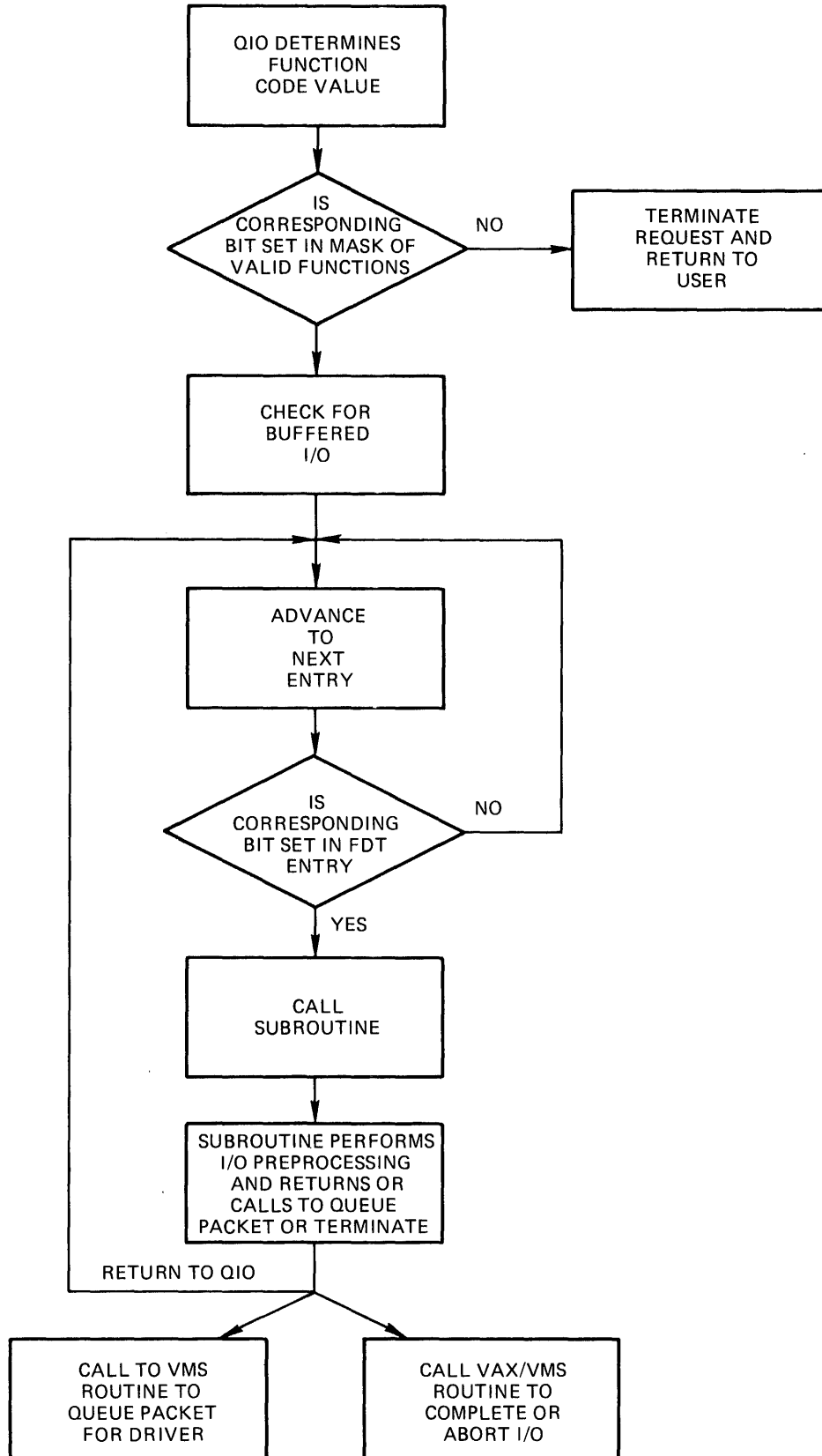


Figure 5-6 FDT Routines and I/O Preprocessing

OVERVIEW OF I/O PROCESSING

5.2.1 Creating a Driver Fork Process to Start I/O

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

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

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

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

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

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

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

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

OVERVIEW OF I/O PROCESSING

5.2.2 Activating a Device and Waiting for an Interrupt

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

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

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

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

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

OVERVIEW OF I/O PROCESSING

5.2.3 Handling a Device Interrupt

Once the driver's start I/O routine initiates the transfer, the driver invokes a VAX/VMS routine to wait for an interrupt. When the device requests an interrupt, the processor dispatches the interrupt to the UNIBUS adapter interrupt service routine. This routine determines whether a device requested the interrupt or the UNIBUS adapter requested the interrupt because of a UBA error condition. If a UNIBUS adapter error caused the interrupt, the system handles the error.

If a device requested the interrupt, the UNIBUS adapter interrupt service routine transfers control to the driver interrupt service routine. The driver's interrupt service routine runs at a high interrupt priority level so that the routine can service interrupts quickly. A driver interrupt service routine usually performs the following processing.

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

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

5.2.4 Switching from Interrupt to Fork Process Context

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

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

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

OVERVIEW OF I/O PROCESSING

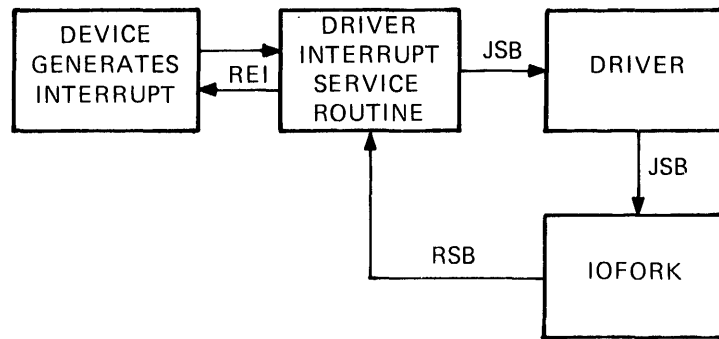


Figure 5-7 Creating a Fork Process After an Interrupt

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. One interrupt service routine handles all interrupts for fork process dispatching. When the processor grants an interrupt at a fork IPL, the fork dispatcher processes the fork queue that corresponds to the IPL of the interrupt. To do so, the dispatcher performs the following steps:

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

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

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

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

OVERVIEW OF I/O PROCESSING

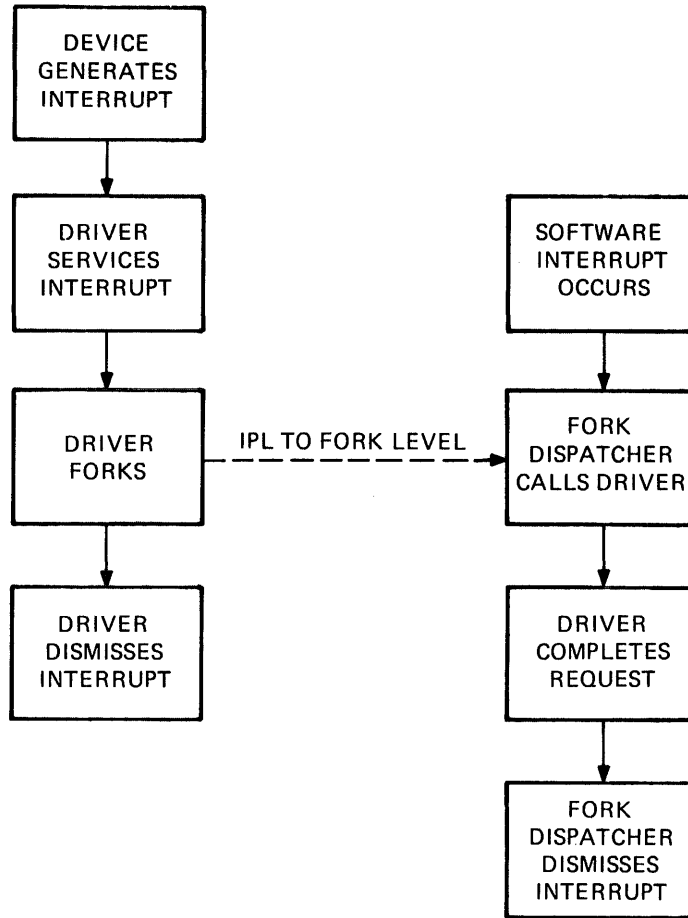


Figure 5-8 Reactivation of a Driver Fork Process

5.3 COMPLETION OF AN I/O REQUEST

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

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

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

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

OVERVIEW OF I/O PROCESSING

5.3.1 I/O Postprocessing

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

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

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

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

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

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

PART II

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 code the following sections of a driver:

- Static tables
- Function decision table routines
- Routines that start an I/O operation on the device and complete the I/O operation
- Interrupt service routines
- Routines that request allocation of UNIBUS adapter map registers and data paths
- Routines that initialize devices and controllers
- Routines that cancel an I/O operation
- Error-logging routines

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

NOTE

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

CHAPTER 6

TEMPLATE FOR AN I/O DRIVER

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

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

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

6.1 CODING CONVENTIONS

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

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

TEMPLATE FOR AN I/O DRIVER

DRIVER ORGANIZATION

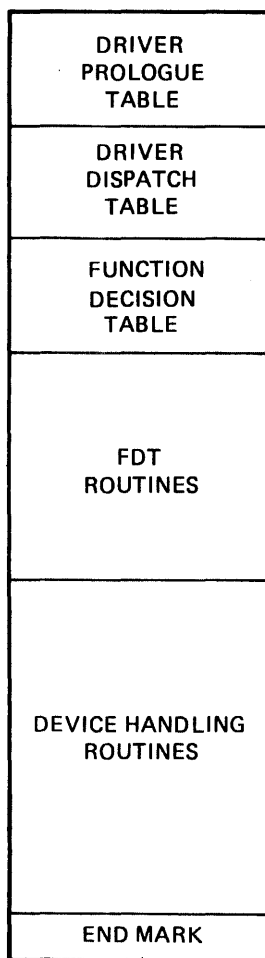


Figure 6-1 Driver Organization

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

- Code can branch only to relative addresses within the driver and to global addresses listed in the VAX/VMS symbol table (SYS.STB in SYS\$SYSTEM:).
- 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 refers to the I/O data base by loading the address of a data structure into a general register and using displacement addressing to the fields of the data structure.

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

TEMPLATE FOR AN I/O DRIVER

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 scratch 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 scratch registers. The routines can use other registers, if necessary, by saving and restoring them but are discouraged from so doing.
- All driver routines can use the stack for temporary storage only if the routines restore the stack to its previous state before calling any VAX/VMS routines or executing RSB instructions.

6.2 RESTRICTIONS ON DEVICE REGISTER I/O SPACE USE

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

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

```
;
; Device register offsets
;

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

.
.
.
MOVL   UCB$L_CRB(R5),R4 ; Get address of CRB
MOVL   CRB$L_INTD+VEC$L_IDB(R4),R4 ; Get the address of
; the device's CSR

.
.
.
TSTW   LP_CSR(R4) ; Is printer online?
```

- Floating, double, field, queue, or quadword operands are not allowed in I/O address space, nor can an instruction obtain the position, size, length, or base of an operand from I/O space. For example, a driver cannot use a field instruction to test a bit in a device register.
- Drivers cannot use string instructions because they are restartable.
- Drivers can use only those instructions with a maximum of one modify or write destination. The destination must be the last operand.

TEMPLATE FOR AN I/O DRIVER

- Registers of devices connected to the SBI (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.

Some UNIBUS devices, such as the LP11, use byte-addressable registers. When a device driver refers to this type of register, it can use byte-context instructions (MOVB, BISB, and so on).

- Unaligned references and references using a length attribute other than the length of the register may produce unpredictable results. For example, if an instruction makes a byte reference to a word-addressable register, the byte addressed may not be modified or supplied. If an instruction makes a word reference to a UNIBUS adapter register, the system signals a machine check exception.
- After an instruction refers to I/O space, it must not handle an exception or interrupt. If the instruction is allowed to restart, it will re-read the device register, which causes undesirable device side-effects or data loss. This rule applies to device driver instructions and instructions within processes that have called the Create and Map Section system service to map part of UNIBUS I/O space.

Also, if an instruction executes above IPL\$ASTDEL, it must not incur an exception or a page fault; if it does, the operating system signals a bugcheck.

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

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

TEMPLATE FOR AN I/O DRIVER

.TITLE TDRIVER - VAX/VMS TEMPLATE DRIVER
.IDENT 'V02-000'

```
;  
;*****  
;*  
;* Copyright (c) 1978,1979,1980 *  
;* by DIGITAL Equipment Corporation, Maynard, Mass. *  
;*  
;* This software is furnished under a license and may be used and copied *  
;* only in accordance with the terms of such license and with the *  
;* inclusion of the above copyright notice. This software or any other *  
;* copies thereof may not be provided or otherwise made available to any *  
;* other person. No title to and ownership of the software is hereby *  
;* transferred. *  
;*  
;* The information in this software is subject to change without notice *  
;* and should not be construed as a commitment by DIGITAL Equipment *  
;* Corporation. *  
;*  
;* DIGITAL assumes no responsibility for the use or reliability of its *  
;* software on equipment which is not supplied by DIGITAL. *  
;*  
;*****  
;  
;+  
;  
; FACILITY:  
;  
; VAX/VMS Template driver  
;  
; ABSTRACT:  
;  
; This module contains the outline of a driver:  
;  
; Models of driver tables  
; Controller and unit initialization routines  
; An FDT routine  
; The start I/O routine  
; The interrupt service routine  
; The cancel I/O routine  
; The device register dump routine  
;  
; AUTHOR:  
;  
; S. Programmer 11-NOV-1979  
;  
; REVISION HISTORY:  
;  
; V02 JHP001 J. Programmer 2-Aug-1979 11:27  
; Remove BLBC instruction from CANCEL routine.  
;  
;--
```

TEMPLATE FOR AN I/O DRIVER

.SBTTL External and local symbol definitions

```

;
; External symbols
;

        $CRBDEF                ; Channel request block
        $DCDEF                 ; Device classes and types
        $DDBDEF                ; Device data block
        $DEVDEF                 ; Device characteristics
        $IDBDEF                 ; Interrupt data 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
;

```

TEMPLATE FOR AN I/O DRIVER

```

$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
        <G0,,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_WRCNT        ; 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.

```


TEMPLATE FOR AN I/O DRIVER

.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_FUNCNABLE,- ; FDT address
            CANCEL=TD_CANCEL,- ; Cancel I/O routine
            REGDMP=TD_REG_DUMP ; Register dump routine

```

;
; Function decision table
;

```

TD_FUNCNABLE:                ; 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

```

TEMPLATE FOR AN I/O DRIVER

```

                                <READVBLK,-
                                READLBLK,-
                                READPBLK>
FUNCTAB +EXE$WRITE,-          ; read virtual,
                                <WRITEVBLK,-
                                WRITELBLK,-
                                WRITEPBLK> ; read logical,
                                +EXE$SETMODE,- ; and read physical.
                                <SETCHAR,-    ; FDT write routine for
                                SETMODE>      ; write virtual,
                                ; write logical,
                                ; and write physical.
FUNCTAB +EXE$SETMODE,-        ; FDT set mode routine
                                <SETCHAR,-    ; for set chars. and
                                SETMODE>      ; set mode.
```

TEMPLATE FOR AN I/O DRIVER

```
.SBTTL TD_CONTROL_INIT, Controller initialization routine

; ++
; TD_CONTROL_INIT, Readies controller for I/O operations
;
; Functional description:
;
;     The operating system calls this routine in 3 places:
;
;         at system startup
;         during driver loading and reloading
;         during recovery from a power failure
;
; Inputs:
;
;     R4     - address of the CSR (controller status register)
;     R5     - address of the IDB (interrupt data 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
```

TEMPLATE FOR AN I/O DRIVER

```
.SBTTL TD_UNIT_INIT, Unit initialization routine

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

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

TEMPLATE FOR AN I/O DRIVER

.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
```

TEMPLATE FOR AN I/O DRIVER

```

.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.

```

TEMPLATE FOR AN I/O DRIVER

```

.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 REQPCCHANL 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 data
;           block)
;     4(SP) - saved R0
;     8(SP) - saved R1
;     12(SP) - saved R2
;     16(SP) - saved R3
;     20(SP) - saved R4
;     24(SP) - saved R5
;     28(SP) - saved PSL (program status longword)
;     32(SP) - saved PC
;
;     The IDB contains the CSR address and the UCB address.
;
; Outputs:
;
;     The routine must preserve all registers except R0-R5.
;
; --

TD_INTERRUPT:
    MOVL    @(SP)+,R4                ; Service device interrupt
                                           ; 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)

;

```

TEMPLATE FOR AN I/O DRIVER

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

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


TEMPLATE FOR AN I/O DRIVER

```

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

;
; Device-dependent cancel operations go next.
;

;
; Finally, the return.
;

10$:
    RSB                               ; Return

```

TEMPLATE FOR AN I/O DRIVER

```

.SBTTL TD_REG_DUMP, Device register dump routine

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

```

TEMPLATE FOR AN I/O DRIVER

```
.SBTTL TD_END, End of driver  
; ++  
; Label that marks the end of the driver  
; --  
  
TD_END: ; Last location in driver  
      .END
```

CHAPTER 7

CODING DEVICE DRIVER TABLES

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

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

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

PARAMETER=value

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

7.1 DRIVER PROLOGUE TABLE (DPT)

The driver prologue table is the first generated code in every device driver. This table, along with parameters to the SYSGEN command that requests driver loading, describes the driver to the driver loading procedure. This procedure computes the size of the driver and loads it into nonpaged system memory. The procedure also creates control blocks in the I/O data base for new devices and writes values from the driver prologue table into fields of these control blocks, as described in Chapter 14.

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

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

CODING DEVICE DRIVER TABLES

prologue table into a list of all driver prologue tables known to the system.

Most device drivers need to have certain other fields of the I/O data base initialized when the driver and its hardware control blocks are loaded. The driver lists these fields in DPT_STORE macro invocations immediately after the DPTAB macro invocation. The DPT_STORE macro accepts two lists of fields:

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

The DPTAB macro stores the relative addresses of the two lists in the driver prologue table. Section 7.1.2 describes the format of the DPT_STORE macro.

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

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

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

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

7.1.1 DPTAB Macro

The DPTAB macro creates a driver prologue table.

Format

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

end

The address of the end of the driver module.

CODING DEVICE DRIVER TABLES

adapter

The adapter type.

UBA = UNIBUS adapter
MBA = MASSBUS adapter

flags

The driver loader flags.

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

DPT\$M_NOUNLOAD Indicates, when set, that the driver cannot be reloaded. A system bootstrap must occur before the driver can be reloaded.

ucbsize

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

unload

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

maxunits

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

name

The name of the device driver module. By convention, a driver name is formed by appending the string DRIVER to the 2-alphabetic character generic device name, for example, DBDRIVER.

7.1.2 DPT_STORE Macro

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

Format 1

DPT_STORE label-name

CODING DEVICE DRIVER TABLES

label-name

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

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

Format 2

DPT_STORE struct type, struct offset, operation, expression,
 [position], [size]

struct type

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

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

struct offset

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

operation

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

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

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

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

expression

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

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

CODING DEVICE DRIVER TABLES

position

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

size

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

7.1.3 Example of DPTAB and DPT_STORE Macro Use

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

```
DPTAB - ; Define DPT
  END=XX END,- ; End of driver
  ADAPTER=UBA,- ; Adapter type
  UCBSIZE=140,- ; Size of UCB
  NAME=XXDRIVER ; Name of driver module
DPT_STORE INIT ; Start of control block
; initialization values
DPT_STORE UCB,UCB$B_FIPL,B,8 ; Driver fork IPL
DPT_STORE UCB,UCB$L_DEVCHAR,L,- ; Device characteristics:
  <DEV$M_REC- ; record-oriented
  !DEV$M_AVL- ; available
  !DEV$M_ODV> ; output device
DPT_STORE UCB,UCB$B_DEVCLASS,B,- ; Device class
  DC$ XX
DPT_STORE UCB,UCB$B_DEVTYPE,B,- ; Device type
  XX$ XL78
DPT_STORE UCB,UCB$W_DEVBUSIZ,W,- ; Default buffer size
  132
DPT_STORE UCB,UCB$B_DIPL,B,22 ; Device IPL

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

7.2 DRIVER DISPATCH TABLE (DDT)

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

CODING DEVICE DRIVER TABLES

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

7.2.1 DDTAB Macro

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

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

Format

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

devnam

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

start

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

unsolic

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

functb

The address of the function decision table for this driver.

cancel

The address of the cancel I/O operation routine.

regdmp

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

diagbf

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

erlgbf

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

unitinit

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

altstart

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

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

CODING DEVICE DRIVER TABLES

7.2.2 Example of a DDTAB Macro

A sample invocation of the DDTAB macro follows.

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

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 bug in device drivers.

7.3 FUNCTION DECISION TABLE (FDT)

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

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

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

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

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

Table 7-1 lists the physical, logical, and virtual I/O function codes that a function decision table can specify.

CODING DEVICE DRIVER TABLES

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

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

7.3.1 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.

CODING DEVICE DRIVER TABLES

```
IO$ STARTCLOCK=IO$ ERASETAPE      ; Start hardware clock
IO$ STOPCLOCK=IO$ OFFSET          ; Stop hardware clock
IO$ STARTDATA=IO$ SPACEFILE      ; Start data acquisition
```

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

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

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

7.3.2 Determining Those Functions that are Buffered I/O

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

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

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

CODING DEVICE DRIVER TABLES

7.3.3 FUNCTAB Macro

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

Format

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

action

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

codes

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

7.3.4 Example of FUNCTAB Macro Use

A sample function decision table follows:

```
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
>
```

In the example above, 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.

CHAPTER 8

CODING FDT ROUTINES

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

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

8.1 CONTEXT FOR FDT ROUTINE EXECUTION

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

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

8.2 REGISTERS PRESET FOR FDT ROUTINE EXECUTION

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

CODING FDT ROUTINES

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

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

8.3 CONVENTIONS FOLLOWED BY FDT ROUTINES

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

8.3.1 Register Conventions

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

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

CODING FDT ROUTINES

8.3.2 Process Context Conventions

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

- It does not lower IPL below IPL\$ASTDEL.
- If a routine raises IPL, it must lower IPL to IPL\$ASTDEL before exiting.
- It does not alter the stack without restoring its original state before exiting.
- It must not call system services or VAX-11 RMS services
- It must observe the register conventions described in the previous section.
- It exits either by an RSB instruction to return control to the system service, or it issues a JMP instruction to one of the VAX/VMS routines described in Section 8.4.

8.4 TRANSFERRING INTO AND OUT OF AN FDT ROUTINE

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

```
JSB    (R0)
```

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

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

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

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

CODING FDT ROUTINES

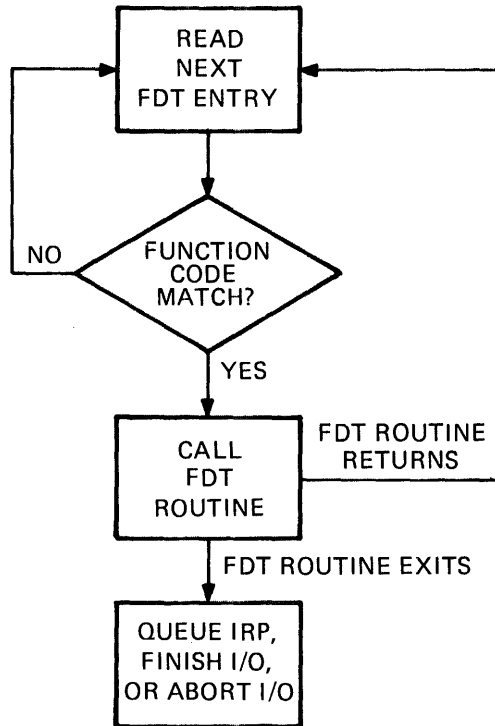


Figure 8-1 Queue I/O Request Scan of a Function Decision Table

8.4.1 Exit Methods

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

8.5 FDT ROUTINES FOR DIRECT I/O

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

When called by the driver, these routines completely prepare a direct I/O read or write request. Thus, a driver that uses these routines eliminates the need for its own device-specific FDT routines.

EXE\$READ and EXE\$WRITE are described in 8.7.

CODING FDT ROUTINES

Table 8-2
FDT Exit Methods

Exit Method	Result
RSB	<p>Returns to the Queue I/O Request system service. The FDT routine returns to the system service because the routine knows that the function decision table contains a subsequent entry with the same function code bit set. As a result, the system service calls another FDT routine.</p>
<p>JMP G^EXE\$QIODRVPKT or JMP G^EXE\$ALTQUEPKT</p>	<p>Transfers control to a VAX/VMS routine that queues an I/O packet to a driver. The FDT routine uses this exit method if all preprocessing is complete, if no fatal errors are found in the specification of an I/O request, and if device activity is required to complete the I/O request.</p> <p>Once an FDT routine transfers control to either of these routines, no driver code that further processes the I/O request can refer to the process virtual address space.</p> <p>EXE\$QIODRVPKT is the standard method used to queue an I/O request for device activity. This routine initiates driver action only if the device unit is currently idle; that is, there is no I/O request being processed. If the device unit is busy, EXE\$QIODRVPKT queues the request to the unit so that the driver will process it when the unit becomes available.</p> <p>In contrast, EXE\$ALTQUEPKT initiates driver action at a special driver entry point without regard for the device unit's activity status. This routine is called by drivers that can handle two or more I/O requests simultaneously.</p>
JMP G^EXE\$FINISHIO	<p>Transfers control to a VAX/VMS routine that writes a quadword of final I/O status from R0 and R1 into the I/O status field of the I/O request packet (IRP\$<u>L</u> MEDIA and IRP\$<u>L</u> MEDIA+4). The routine then inserts the I/O request packet in the I/O postprocessing queue.</p> <p>An FDT routine that discovers a device-dependent error should always return status using EXE\$FINISHIO or EXE\$FINISHIOC. The routine returns to the Queue I/O Request system service the two longwords of status contained in the I/O status block (if any) specified in the Queue I/O Request.</p>

(continued on next page)

CODING FDT ROUTINES

Table 8-2 (Cont.)
FDT Exit Methods

Exit Method	Result
JMP G^EXE\$FINISHIOC	Transfers control to a routine that performs the same functions as EXE\$FINISHIO except that this routine always clears the second longword of the final I/O status.
JMP G^EXE\$ABORTIO	Transfers control to a VAX/VMS routine that aborts an I/O request. An FDT routine that discovers a device-independent error in an I/O request should always use this method of exit. The routine stores a longword of status in R0 and returns this to the system service. Inability to gain access to a data buffer is an example of a device-independent error.

Section 8.7 details the VAX/VMS routines summarized above.

8.6 FDT ROUTINES FOR BUFFERED I/O

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

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

8.6.1 Checking the User's Buffer

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

8.6.2 Allocating the System Buffer

Next, the FDT routine allocates a system buffer. First, it adds 12 bytes for a buffer header to the byte count passed in the P2 parameter of the user's I/O request. This is the total system buffer size. The FDT routine then calls EXE\$BUFFERQUOTA to ensure that the user has sufficient remaining resources. If EXE\$BUFFERQUOTA 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.

CODING FDT ROUTINES

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

- Loads the address of the system buffer into IRP\$SVAPTE
- Loads the total size of the system buffer into IRP\$W_BOFF
- Subtracts the system buffer size from JIB\$L_BYTCNT. A longword in the PCB points to the location of the Job Information Block (JIB).
- Stores the starting address of the system buffer data area in the first longword of the buffer header
- Stores the user's buffer address in the second longword of the header
- Copies data from the user buffer to the system buffer if the I/O request is a read operation

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

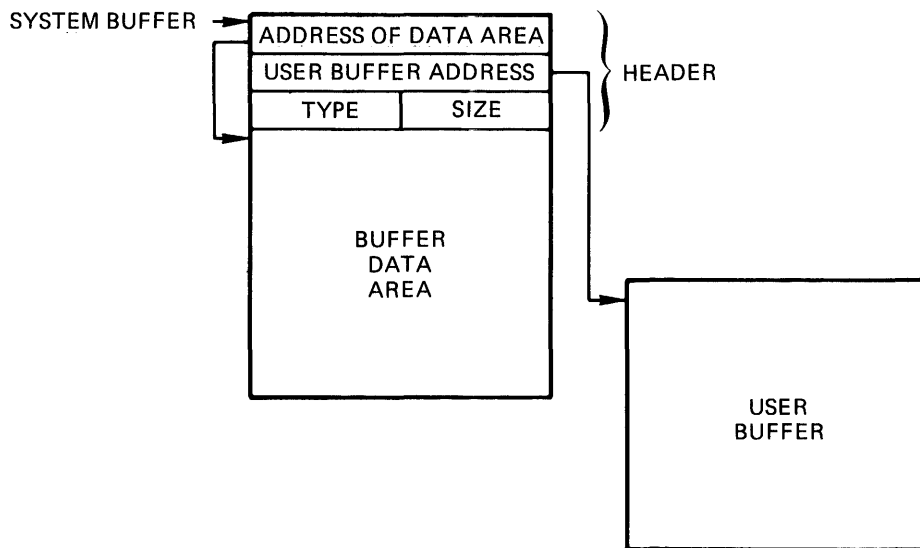


Figure 8-2 Format of System Buffer for Buffered I/O Read Operations

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

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

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

CODING FDT ROUTINES

accuracy. When VAX/VMS I/O postprocessing gains control, it performs the following steps:

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

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

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

8.7 FDT ROUTINES PROVIDED BY VAX/VMS

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

- EXE\$QIODRVPKT
- EXE\$ALTQUEPKT
- EXE\$FINISHIO
- EXE\$FINISHIOC
- EXE\$ABORTIO

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

For information about additional FDT routines, see Appendix C.

CODING FDT ROUTINES

8.7.1 EXE\$ONEPARM

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

Exit Method

Queues the I/O request packet to the driver.

Description

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

8.7.2 EXE\$READ

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

Exit Method

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

Description

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

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

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

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

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

If EXE\$READCHKR finds that the buffer is accessible, it updates the I/O request packet by writing the size in bytes of the transfer to IRP\$W_BCNT and setting the read status bit in IRP\$W_STS (IRP\$V_FUNC). The maximum number of bytes that EXE\$READ can transfer is 65536 (128 pages).

CODING FDT ROUTINES

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

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

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

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

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

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

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

8.7.3 EXE\$SENSEMODE

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

Exit Method

Transfers control to EXE\$FINISHIO.

Description

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

CODING FDT ROUTINES

8.7.4 EXE\$SETCHAR

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

Exit Method

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

Description

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

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

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

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

8.7.5 EXE\$SETMODE

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

Exit Method

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

Description

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

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

CODING FDT ROUTINES

8.7.6 EXE\$WRITE

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

Exit Method

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

Description

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

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

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

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

If EXE\$WRITECHKR finds that the buffer is accessible, it updates the I/O request packet by writing the size in bytes of the transfer to IRP\$W_BCNT. EXE\$WRITE can transfer a maximum of 65536 bytes (128 pages).

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

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

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

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

However, if MMG\$IOLOCK reports a page fault, EXE\$WRITELOCKR adjusts direct I/O count and AST count to the values they held before the I/O request packet and restarts the request procedure at the Queue I/O system service. The routine carries out this procedure so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the system service will resubmit the queue I/O request.

CODING FDT ROUTINES

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

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

8.7.7 EXE\$ZEROPARM

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

Exit Method

Queues the I/O request packet to the driver.

Description

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

8.8 EXIT ROUTINES IN THE VAX/VMS SYSTEM

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

8.8.1 EXE\$ABORTIO

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

Required Register Contents

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

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

CODING FDT ROUTINES

Description

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

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

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

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

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

8.8.2 EXE\$FINISHIO and EXE\$FINISHIOC

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

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

Required Register Content

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

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

CODING FDT ROUTINES

Description

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

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

The software interrupt occurs at IPL 3 so that postprocessing interrupts EXE\$FINISHIO or EXE\$FINISHIOC. When EXE\$FINISHIO or EXE\$FINISHIOC regains control, it completes processing in the following steps:

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

8.8.3 EXE\$QIODRVPKT

Some I/O functions require device activity, or at least access to device registers, for the I/O operation to be completed. Common examples are read and write functions. The FDT routines can perform extensive preprocessing, such as determining whether user buffers are accessible and reformatting data into buffers in the system address space, but they should not access device registers because the device might be active. By convention, FDT routines do not modify the unit control block or device register contents for reasons of synchronization. FDT routines do not execute at the proper IPL (fork IPL) to make such modifications. As a result, they could crash the system or cause their driver to execute incorrectly.

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

Required Register Contents

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

CODING FDT ROUTINES

Description

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

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

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

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

UCB → DDB → DDT → start I/O entry point
- Transfers control to the driver start I/O routine using a JMP instruction

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

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

CODING FDT ROUTINES

EXE\$INSIOQ calls the VAX/VMS routine EXE\$INSERTIRP to insert the I/O request packet in the unit's I/O request packet queue. Then, EXE\$INSIOQ reduces the interrupt priority level to the level at the beginning of its execution; that is, to IPL\$ASTDEL. EXE\$INSIOQ returns control to EXE\$QIODRVPKT. Finally, EXE\$QIODRVPKT returns from the Queue I/O Request system service in the following steps:

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

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

When a device driver's start I/O routine gains control, the process that queued the I/O request may no longer be the mapped process. Therefore, the driver must assume that all information regarding the I/O request is in the unit control block or the I/O request packet and that all buffer addresses in the unit control block are either system addresses or page frame numbers that can be interpreted in any process context. For direct I/O operations, FDT routines also must have locked all user buffer pages in physical memory since paging cannot occur at driver fork level and higher interrupt priority levels. The process virtual address space is not guaranteed to be mapped again until VAX/VMS delivers a kernel mode AST to the requesting process as part of I/O postprocessing.

8.8.4 EXE\$ALTQUEPKT

Special purpose drivers may want to use their own internal I/O queues as well as the device unit I/O queue (UCB\$L_IQQFL) provided by VAX/VMS. These internal queues allow the driver to handle I/O requests even if the device is busy with another I/O operation.

EXE\$ALTQUEPKT permits the driver to ignore unit I/O queue synchronization. When called by an FDT routine, EXE\$ALTQUEPKT gains access to the driver at the alternate start I/O entry point specified in the driver dispatch table (offset DDT\$L_ALTSTART). This entry point bypasses the unit I/O queue and the device busy flag; thus, the driver is activated regardless of whether the device unit is busy.

A driver that uses EXE\$ALTQUEPKT becomes responsible not only for its internal queues but also for any synchronization between those queues and the unit I/O queue maintained by the operating system.

Drivers complete I/O request packets obtained from EXE\$ALTQUEPKT by calling the routine COM\$POST. This routine places the I/O request packet in a postprocessing queue and returns control to the driver. The driver may then fetch another packet from an internal queue.

If a driver processes more than one I/O request packet at the same time, separate fork blocks must be used.

Be aware that programming a device driver to process simultaneous I/O requests requires detailed knowledge of VAX/VMS internal design.

CODING FDT ROUTINES

Required Register Contents

R3 Address of the I/O request packet
R5 Address of the unit control block

You must assume that the contents of R0 through R5 are destroyed upon return to the FDT routine.

Description

EXE\$ALTQUEPKT performs the following steps:

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

UCB → DDB → DDT → alternate start I/O address.
- Calls the driver at alternate start I/O address.

When the alternate start I/O routine finishes, it returns control to EXE\$ALTQUEPKT by executing an RSB instruction. EXE\$ALTQUEPKT restores the interrupt priority level saved on the stack and then returns control to the FDT routine that called it. The FDT routine then executes a JMP instruction to the routine EXE\$QIORETURN in order to return control to the user process.

CHAPTER 9

CODING THE START I/O ROUTINE

A driver start I/O routine activates a device and then waits for a device interrupt or timeout. This chapter describes the start I/O routine. Chapter 12 describes the reactivation of the driver routine that performs device-dependent I/O postprocessing. The start I/O routine discussed in the following sections describes a DMA transfer using a single-unit controller.

9.1 TRANSFERRING CONTROL TO START I/O

The start I/O routine of a device driver gains control from either of two VAX/VMS routines: EXE\$QIODRVPKT or IOC\$REQCOM.

When FDT processing is complete for an I/O packet, the FDT routine transfers control to EXE\$QIODRVPKT. If the designated device is idle, IOC\$INITIATE is called to create a driver fork process. (This procedure is detailed in Section 8.7.3.) The driver fork process then gains control in the start I/O routine of the appropriate driver. If the device is busy, EXE\$QIODRVPKT queues the packet to a device unit's I/O request packet wait queue.

After a device completes an I/O operation, the driver fork process exits by transferring control to IOC\$REQCOM. IOC\$REQCOM inserts the finished I/O packet in the postprocessing queue. It then dequeues the next I/O request packet from the device unit's I/O request packet wait queue and calls IOC\$INITIATE to create a new driver fork process that gains control at the entry point of the driver's start I/O routine.

9.2 CONTEXT OF A DRIVER FORK PROCESS

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

System mapping	Only system page tables are mapped. Therefore, driver code cannot refer to virtual addresses in process address space.
Kernel mode	Execution occurs in the most privileged access mode and can, therefore, change IPL.
High IPL	The VAX/VMS routine that creates a driver fork process raises IPL to driver fork level before activating the driver. The driver can raise and lower IPL between driver fork level and IPL\$_POWER.

CODING THE START I/O ROUTINE

Kernel or interrupt stack	Execution occurs on the kernel or interrupt stack. The driver must not alter the state of the stack without restoring it to its previous state before relinquishing control.
------------------------------	---

The start I/O routine executes on the kernel stack if the VAX/VMS packet queuing routine activated the start I/O routine. It executes on the interrupt stack if the VAX/VMS request complete routine activates the start I/O routine.

In addition to the context described, the VAX/VMS packet queuing routine sets up R3 and R5 for a driver start I/O routine, as follows:

- R3 contains the address of the I/O request packet.
- R5 contains the address of the unit control block for the device.

All registers must be preserved except for R0, R1, R2 and R4.

The packet queuing routine also copies the following I/O request packet fields into the UCB:

- IRP\$W_BCNT
- IRP\$W_BOFF
- IRP\$L_SVAPTE

9.3 ACTIVATING THE DEVICE

The processing performed by a start I/O routine is device-specific. A start I/O routine normally contains elements to perform the following functions:

- Analyze the I/O function
- Transfer the details of a transfer from the I/O request packet into the unit control block
- Obtain and initialize the controller and, for DMA transfers, UNIBUS adapter resources
- Modify device registers to activate the device

The start I/O routine elements listed above execute a series of steps to activate the device. The sections that follow describe those steps as performed for a sample DMA device such as a parallel communications link; the details of processing, however, are specific to the particular device. UNIBUS-related details of DMA transfers are described in Chapter 10.

9.3.1 Obtaining Controller Access

If the device is attached to a multiunit controller, the start I/O routine invokes the VAX/VMS macro REQCHAN to assign the controller data channel to the device unit. Single-unit controllers do not require arbitration for the controller data channel. REQCHAN calls the VAX/VMS routine IOC\$REQCHANL that acquires ownership of the controller data channel.

CODING THE START I/O ROUTINE

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 REQCHAN 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\$REQCHANL 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\$REQCHANL then returns control to the caller of the driver, that is, to IOC\$INSIOQ, as illustrated in Figure 9-1.

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

- Saved R3 containing the address of the I/O request packet
- Implicit saved R5 containing the UCB address
- A return address in the driver

IOC\$REQCHANL does not save R4 since it writes R4 before returning control to the driver.

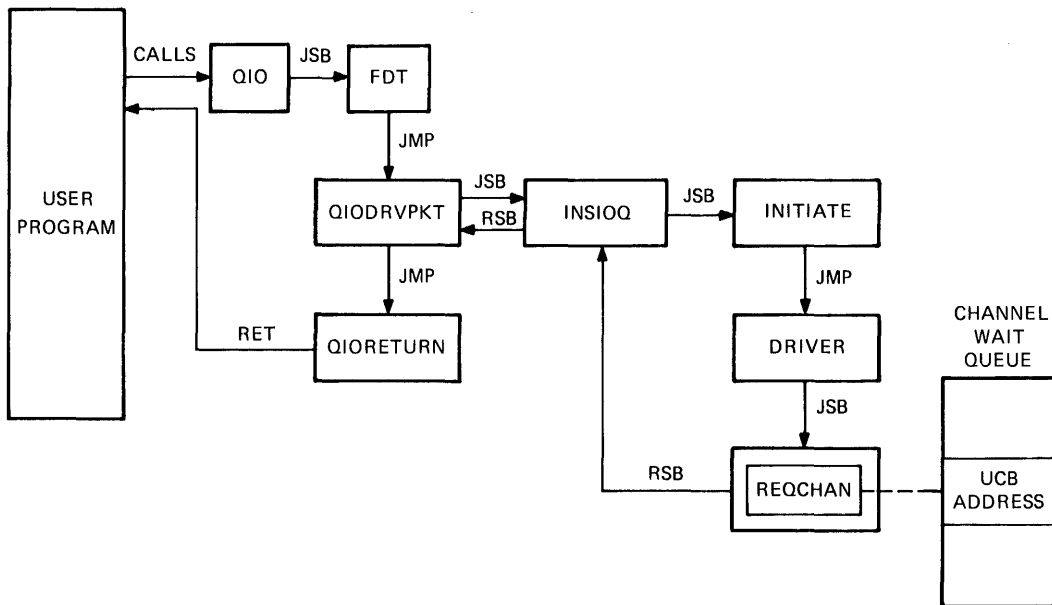


Figure 9-1 Driver Insertion into Channel Wait Queue

If the channel is available, IOC\$REQCHANL locates the interrupt data block for the channel with a pointer in the unit control block:

UCB → CRB → IDB

The interrupt data block contains the address of the control/status register for the channel (IDB\$L CSR). IOC\$REQCHANL returns the control/status register address in R4. The driver for a unit attached to a single-unit controller must contain the code needed to load the control/status address into R4.

CODING THE START I/O ROUTINE

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

The driver must maintain the stack in a known and consistent state for the resource wait queue mechanism to work. When IOC\$REQPCHANL gains control, the top two items on the stack must be two return addresses:

- 0(SP) -- Address of the next instruction to be executed in the driver fork process
- 4(SP) -- Address of the next instruction to be executed in the routine that called the driver start I/O routine

9.3.2 Getting the I/O Function Code and Converting the Code and Modifiers

The start I/O routine extracts the I/O function code and function modifiers from the field IRP\$W_FUNC and translates them into device-specific function codes to be loaded into the device's control/status register or other control registers. The I/O routine being described in this chapter sets up a bit mask that is to be modified further in subsequent instructions and loaded into the control/status register when the driver actually starts the device. That is, the start I/O routine converts the function modifiers contained in IRP\$W_FUNC into device-specific bit settings in the general register (R3 in this case).

9.3.3 Obtaining a Buffered Data Path

If the device uses a buffered data path, the start I/O routine invokes the VAX/VMS macro REQDPR to allocate the data path; Chapter 10 provides the details of interfacing with the UNIBUS adapter, including a description of the REQDPR macro. REQDPR calls the VAX/VMS routine IOC\$REQDATAP, which allocates a data path if one is available.

If no buffered data path is available, IOC\$REQDATAP suspends driver processing by saving the driver's context in the UCB fork block and inserting the fork block address in the data path wait queue. IOC\$REQDATAP then returns control to the caller of the driver.

If a buffered data path is available, IOC\$REQDATAP writes the number of the data path allocated to the driver into the channel request block at CRB\$L_INTD+VEC\$B_DATAPATH.

If the device uses a direct data path, no data path allocation is required. The direct data path is the default data path as long as no code has ever written a nonzero value into the CRB data path field.

9.3.4 Loading Map Registers

The driver's start I/O routine invokes the VAX/VMS macro LOADUBA to load the page frame numbers of the physical pages involved in the transfer into the allocated map registers. The macro calls the VAX/VMS routine IOC\$LOADUBAMAP, as described in Chapter 10. Using the byte offset and byte count fields of the unit control block

CODING THE START I/O ROUTINE

(UCB\$W_BOFF and UCB\$W_BCNT, respectively), IOC\$LOADUBAMAP computes the number of map registers to load. It then obtains the number of the first page frame number from the page table entry field of the unit control block (UCB\$L_SVAPTE).

In each map register, IOC\$LOADUBAMAP sets the valid bit and fills in the information needed for the transfer, that is, data path number, physical page frame number, and an indication of whether the transfer is word aligned. For further information, refer to Section 10.3.

9.3.5 Computing the Transfer Length

Because the device driven by this particular driver expects the transfer as a word count, the start I/O routine computes the length of the transfer in words by dividing the byte count field of the unit control block (UCB\$W_BCNT) by 2. The routine loads the computed value into the word count device register. One of the FDT routines that processes the I/O request must ensure that the byte count for the transfer is even. An odd byte count results in the user's not receiving the last byte of data.

9.3.6 Computing the Transfer Start Address

The start I/O routine calculates the address of the transfer using the byte offset field of the unit control block (UCB\$W_BOFF) and the number of the starting map register (CRB\$L_INTD+VEC\$W_MAPREG). The result is an 18-bit value representing an address in UNIBUS address space. Section 10.4 details the calculation of the starting address for a UNIBUS transfer.

The start I/O routine stores the low-order 16 bits of the computed value in the buffer address device register. It stores the two high-order bits of the computed value in the memory extension bits of the bit mask set up in Section 9.3.2 to contain the device control/status register data, in this case, R3.

9.3.7 Preparing the Device Activation Bit Mask

The start I/O routine prepares the device activation bit mask by setting the interrupt enable and go bits in the general register used previously (in this discussion, R3). The general register contains a complete command to start the transfer at this point. When the start I/O routine copies the contents of the register into the device's control/status register, the device starts the transfer. However, before activating the device, the start I/O routine should perform the steps described in Sections 9.3.8 and 9.3.9.

9.3.8 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, that is, driver fork IPL, on the top of the stack.

CODING THE START I/O ROUTINE

9.3.9 Checking for Power Failure

The start I/O routine examines the powerfail bits in the UCB status word (UCB\$V_POWER in UCB\$W_STS) to determine whether a power failure has occurred since the start I/O routine gained control. If the bit is not set, the transfer can proceed.

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

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

9.3.10 Activating the Device

If no power failure has occurred, the start I/O routine copies the contents of the control mask (in this case, R3) into the device control/status register. When the device notices the new contents of the device register, the actual transfer begins.

9.4 WAITING FOR AN INTERRUPT OR TIMEOUT

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

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

Still executing at IPL\$ POWER, the driver's start I/O routine asks VAX/VMS to suspend the driver fork process by invoking one of the following VAX/VMS macros:

```
WFIKPCH -- Wait for an interrupt or timeout and keep the
           controller data channel
```

```
WFIRLCH -- Wait for an interrupt or timeout and release the
           controller data channel
```

Both of these macros invoke routines that return IPL to the previous level when they exit. These routines expect to find the return IPL on the stack. Original IPL is normally saved on the stack by the DSBINT macro, which the start I/O routine invokes before it checks for power failure, as described in Section 9.3.9.

Drivers generally keep the controller data channel while waiting for the interrupt or timeout. Some I/O operations, however, do not need the controller after the operation is started, for example, a disk seek operation.

Waiting for an interrupt or device timeout is the approach normally taken by drivers.

CODING THE START I/O ROUTINE

9.4.1 WFIKPCH and WFIRLCH Macro Formats

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

Formats

WFIKPCH excpt, [time]

WFIRLCH excpt, [time]

excpt

The address of the timeout routine for this device.

time

The number of seconds to wait before signaling a device timeout. The number must be greater than or equal to 2. A minimum value of 2 is required because the timeout mechanism is accurate only to within one second. If no number is specified, the macro uses the value 65536 by default.

9.4.2 Expansion of WFIKPCH Macro

Because the WFIKPCH and WFIRLCH macros are similar, the description that follows analyzes the expansion of WFIKPCH only.

If the driver specifies the time argument in the macro call, the macro pushes the value of the argument into the stack. If the time argument is not specified, the macro pushes the value 65536 onto the stack.

The VAX/VMS timer routine uses the time value to calculate the length of time to wait before transferring control to a device timeout handler.

WFIKPCH completes its expansion with the following two lines of code:

```
JSB     G^IOC$WFIKPCH
        .WORD   EXCPT-
```

The execution of the JSB instruction pushes the address following the JSB onto the stack as the address to which the called routine would normally return with an RSB instruction.

9.4.3 IOC\$WFIKPCH Routine

The VAX/VMS routine IOC\$WFIKPCH invoked by the macro WFIKPCH performs the functions necessary for the driver fork process to wait for a device interrupt or timeout. IOC\$WFIKPCH first adds 2 to the address on the top of the stack so that the top of the stack contains the address of the next instruction in the driver after the macro invocation. This address is where the driver processing actually resumes as a result of an interrupt service routine JSB instruction.

IOC\$WFIKPCH then saves the contents of R3, R4, and the driver return address from the top of the stack in the first part of the unit control block; that is, in the UCB fork block. The interrupt service routine must restore R5 to contain the address of the unit control block after an interrupt. The interrupt service routine normally obtains the address of the unit control block from the field IDB\$_OWNER of the interrupt data block.

CODING THE START I/O ROUTINE

The VAX/VMS routine that detects a device timeout calculates the address of the driver timeout routine by subtracting 2 from the saved PC in the UCB fork block and calling indirectly through the result, for example:

```
MOVL   UCB$L FPC(R5),R2      ; Get saved PC
CVTWL  -(R2),-(SP)           ; Get offset to timeout
                                   ; handler
ADDL   (SP)+,R2              ; Add to relative driver
                                   ; address to obtain relative
                                   ; handler address
JSB    (R2)                  ; Call timeout handler
```

IOC\$WFIKPCH sets bits in the unit control block (UCB\$V_INT and UCB\$V_TIM in UCB\$W_STS) to indicate that interrupts and timeouts are expected from the device. IOC\$WFIKPCH also writes the device timeout absolute time in the field UCB\$L DUETIM. The absolute time is the number of seconds since the operating system was bootstrapped plus the number of seconds specified in the time argument to the macro.

Finally, IOC\$WFIKPCH reenables interrupts by lowering IPL to its previous level in the driver, that is, to driver fork level, and returns control to the caller of the driver.

9.5 RESPONDING TO AN EXPECTED DEVICE INTERRUPT

The only context saved for the driver is now in the unit control block. It contains the following information:

- A description of the I/O request and the state of the device
- The contents of R3 and R4
- The implicit contents of R5, that is, the address of the UCB fork block
- A driver return address
- The implicit address of a device timeout routine

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

When a device interrupts, the driver interrupt service routine analyzes the interrupt, as detailed in Chapter 11 and summarized below:

- Identifies the UCB address of the device that generated the interrupt
- Obtains device or controller status from the device registers, if necessary, and stores the status in the unit control block
- Restores the driver fork process registers from the UCB fork block, restores R5 with the UCB address, and reactivates the suspended driver at the PC stored in the UCB fork block

If, instead of requesting an interrupt, the device times out, a VAX/VMS timer routine reactivates the suspended driver fork process at the address of the timeout routine. Section 12.2 discusses device timeout handling in detail.

CHAPTER 10

CODING FOR UNIBUS DMA TRANSFERS

A driver performing DMA transfers over the UNIBUS must take UNIBUS operation into consideration. The VAX/VMS operating system and the I/O data base handle most UNIBUS map register and data path resource management for the device drivers. You must choose the type of data path (either direct or buffered) appropriate to the device and ensure that UCB fields are written to describe the virtual memory locations to be read or written. Once these actions have been taken, the driver fork process calls VAX/VMS routines to take care of the detailed operation of the UNIBUS adapter.

The I/O data base contains an adapter control block (ADP) that describes the UNIBUS adapter. This block contains allocation bit maps for the UNIBUS adapter data paths and map registers. Each bit represents one data path or one map register. When the bit is clear, the data path or register is allocated to a device driver.

The adapter control block also contains the virtual address of the UNIBUS adapter configuration register. All other adapter registers are located at fixed offsets from the configuration register. The VAX/VMS UNIBUS adapter-handling routines modify the UNIBUS adapter data path and map registers according to request from driver fork processes.

In general, driver fork processes do not access the UNIBUS adapter control blocks. Instead, drivers call VAX/VMS routines that perform adapter-related services, such as the following:

- Allocate a buffered data path
- Allocate map registers
- Load map registers
- Deallocate map registers
- Purge a buffered data path
- Deallocate a buffered data path

The system creates a driver fork process by calling the start I/O routine in a device driver. The fork process takes some or all the following steps to initiate an I/O transfer on a UNIBUS device:

- Requests buffered data path
- Requests map registers
- Loads map registers

CODING FOR UNIBUS DMA TRANSFERS

- Calculates starting UNIBUS address
- Activates device
- Waits for interrupt

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

- Purges the buffered data path
- Releases the data path
- Releases the map registers

All of the steps above involve the UNIBUS adapter. VAX/VMS, however, hides most of the UNIBUS interfacing from the driver.

10.1 REQUESTING A BUFFERED DATA PATH

The system provides two macros that a driver can invoke to request a buffered data path:

- REQDPR, which suspends the driver to wait for a buffered data path if one is not available
- REQDATAPNW, which returns an error status if no buffered data path is available

In addition, a driver can request the permanent allocation of a buffered data path, as described in Section 10.1.3.

10.1.1 Requesting a Buffered Data Path (with Wait)

A driver fork process requests a buffered data path by invoking the VAX/VMS macro REQDPR. REQDPR calls a VAX/VMS routine named IOC\$REQDATAP that locates the UNIBUS adapter control block. To do this, IOC\$REQDATAP uses a series of pointers that begin in the current unit control block, as follows:

UCB → CRB → ADP

The ADP data path bit map indicates the buffered data paths that are available. IOC\$REQDATAP allocates a data path to the driver by storing the data path number in the channel request block and indicating in the adapter control block (ADP) that the data path is in use. Then, control returns to the driver fork process. Appendix A describes the adapter control block.

If no data path is available, IOC\$REQDATAP saves driver context (R3, R4, and PC) in the UCB fork block and inserts the address of the fork block, which is also the address of the unit control block and the content of R5, in the ADP data path wait queue. The driver fork block remains in the queue until both of the following conditions are met:

- A data path is available
- The driver fork block is the next entry in the data path wait queue

CODING FOR UNIBUS DMA TRANSFERS

Then, the VAX/VMS routine IOC\$RELDATAP allocates the data path to the suspended driver and reactivates the driver fork process.

10.1.2 Requesting a Buffered Data Path (No Wait)

Instead of invoking REQ DPR, the driver fork process can call the subroutine IOC\$REQDATAPNW by invoking the macro REQDATAPNW.

This routine immediately returns control to the driver fork process if no data path is available. The low-order bit of R0 is clear, indicating that the request for allocation was unsuccessful.

If the data path is available, IOC\$REQDATAPNW allocates the data path and returns a normal status code (SS\$NORMAL) to the driver fork process in R0.

10.1.3 Requesting A Permanent Buffered Data Path

A device driver also can permanently allocate a buffered data path with code in a device unit initialization routine. The following steps permanently allocate a buffered data path:

- Invoke the REQ DPRNW macro or the REQ DPR macro to allocate a data path
- Set the path lock bit in the data path number field of the channel request block (VECSV_PATHLOCK in CRB\$L_INTD+VECSB_DATAPATH)

When the driver loading procedure loads or reloads the driver, the procedure calls the unit initialization routine for each device unit associated with the driver. At that time, the unit initialization routine permanently allocates a buffered data path for each device unit if the code described above has been included.

10.1.4 Requesting the Direct Data Path

Because the UNIBUS adapter arbitrates among devices that wish to use the direct data path and because the CRB is initialized to 0 (0 = direct data path), drivers are not required to invoke the REQ DPR macro to request the direct data path.

When a word-aligned UNIBUS device uses the direct data path, the driver must ensure that the specified buffer is on a word boundary, since byte offset is not implemented on the direct data path.

10.1.5 Mixed Direct and Buffered Data Path Transfers

A device driver can use the buffered data path for certain operations, then use the direct data path for other operations. To accomplish this task, the driver should allocate a buffered data path for buffered I/O. When the operation completes, the driver should then purge and release the data path. The release automatically resets the data path number to zero, which signifies a direct data path. However, the driver should take care not to release the direct data path, although it can purge the path if desired. (A purge of the direct data path is a NOP and always yields success.)

CODING FOR UNIBUS DMA TRANSFERS

10.2 REQUESTING UBA MAP REGISTERS

The operating system allows a driver to allocate map registers as needed or to allocate them permanently.

10.2.1 Allocation of Map Registers

A driver fork process requests a set of UNIBUS adapter map registers by invoking the VAX/VMS macro REQMPR. This macro calls the routine IOC\$REQMAPREG. IOC\$REQMAPREG calculates the number of map registers needed for a transfer. The calculation is based on the transfer byte count field and the byte offset field of the device's unit control block (UCB\$W_BCNT and UCB\$W_BOFF).

The procedure for allocating map registers is similar to that used to allocate a buffered data path. First, IOC\$REQMAPREG locates the adapter control block from a series of pointers that begin with the current unit control block, as follows:

UCB → CRB → ADP

Then, the routine examines the map register allocation bit map to locate the required number of contiguous map registers. If the registers are not currently available, IOC\$REQMAPREG saves the driver context (R3, R4, and PC) in the UCB fork block and inserts the fork block address (same as UCB address and the contents of R5) in the map register wait queue.

When the map registers are available, IOC\$REQMAPREG allocates them by clearing the appropriate bits in the map register bit map of the ADP. IOC\$REQMAPREG then writes the number of the starting map register and the number of map registers allocated into the channel request block and returns control to the driver fork process.

10.2.2 Permanent Allocation of Map Registers

A device driver also can permanently allocate a set of UNIBUS adapter map registers with code in a unit or controller initialization routine. You must ensure that the number of map registers permanently allocated is sufficient for the longest possible transfer. The following steps permanently allocate a set of map registers:

- Load the number of map registers required into R3.
- Call the VAX/VMS routine IOC\$ALOUBAMAPN with a JSB instruction:

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

If IOC\$ALOUBAMAPN successfully allocates the map registers, it stores the number of map registers allocated and the starting map register's number in the channel request block at CRB\$L_INTD+VEC\$B_NUMREG and CRB\$L_INTD+VEC\$W_MAPREG, respectively, and returns with the low-order bit set in R0.

Otherwise, it returns with the low-order bit of R0 clear.

- Set the map lock bit in the channel request block (VEC\$C_MAPLOCK in CRB\$L_INTD+VEC\$W_MAPREG).

CODING FOR UNIBUS DMA TRANSFERS

When the driver loading procedure loads or reloads the driver, the procedure calls the unit or controller initialization routine for each device unit or controller associated with the driver. At that time, the initialization routines can permanently allocate map registers.

10.3 LOADING THE UBA MAP REGISTERS

Once a driver fork process has assigned a data path and allocated a set of map registers, it can request VAX/VMS to load the map registers with physical page frame numbers by invoking the VAX/VMS macro LOADUBA. LOADUBA calls a VAX/VMS routine IOC\$LOADUBAMAP that loads each allocated map register with five data items:

- A bit setting to indicate whether the map register is valid.
- A bit setting to indicate whether the transfer is to start on the odd or even byte within a word; this bit is set if the low-order bit of UCB\$W_BOFF is a 1.
- The number of the data path to use for the transfer.
- The page frame number of a page in memory.
- A bit setting to indicate that the transfer operates in longword-aligned random access mode; This bit is set when VEC\$V_LWAE is specified in VEC\$B_DATAPATH.

IOC\$LOADUBAMAP loads the page frame number of the first page of the transfer into the first allocated map register, the page frame number of the second page of the transfer into the second map register, and so forth.

IOC\$LOADUBAMAP sets the valid bit in every allocated map register except the last. It clears the valid bit in the final map register to stop a prefetch from an invalid page frame number.

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

- UCB\$W_BOFF -- byte offset in the first page of the transfer
- UCB\$W_BCNT -- number of bytes to transfer
- UCB\$L_SVAPTE -- virtual address of the page table entry that contains the page frame number of the first page of the transfer

IOC\$LOADUBAMAP determines the data path number, the number of the first map register, the address of the first map register, and the number of map registers from the channel request block and the UNIBUS adapter control block, as follows:

UCB → CRB → data path number

UCB → CRB → number of first map register

UCB → CRB → ADP → virtual address of first map register

UCB → CRB → number of map registers

CODING FOR UNIBUS DMA TRANSFERS

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

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

10.4 COMPUTING THE STARTING ADDRESS OF A TRANSFER

The driver fork process must calculate the starting address of a UNIBUS transfer and load this address into the appropriate device register. The driver takes the following steps to make the calculation:

- Writes the byte offset in page field of the UCB (UCB\$W_BOFF) into bits 0 through 8 of a register
- Gets the number of the starting map register for the transfer from the channel request block; the number is a 9-bit value
- Writes bits 0 through 6 of the map register number into bits 9 through 15 of the register containing the byte offset field
- Writes bits 0 through 15 of the register into the buffer address register for the device
- Writes bits 7 and 8 of the map register number into the extended memory bits of the appropriate device register (usually the control/status register)

10.5 ACTIVATING THE DEVICE

Because a driver fork process can address device registers as though they were any other virtual address, the loading of the UNIBUS buffer address register and control/status register both are simple procedures. The driver locates the CSR address of the device in the interrupt data block, as follows:

UCB → CRB → IDB → CSR address

The CSR address is the virtual address of a device register. All other device registers are located at constant offsets from the CSR address. If, for example, the control/status register is the first device register and the device word count is the third device register, the device driver can load the word count register with the following sequence of instructions:

- Move the CSR address into R4
- Move the number of words to transfer with a MOVW instruction that addresses 4(R4)

CODING FOR UNIBUS DMA TRANSFERS

10.6 COMPLETION OF A DMA TRANSFER

After a driver fork process activates a DMA UNIBUS device, the driver waits for a device interrupt by invoking a VAX/VMS macro that suspends the driver. When the UNIBUS device requests a hardware interrupt, a VAX/VMS interrupt dispatcher gains control. The dispatcher saves R0 through R5 and transfers control to code in the channel request block.

The CRB code calls a driver interrupt service routine. If the service routine can match the interrupt with a suspended driver fork process, the interrupt service routine reactivates the driver fork process at the point that execution was suspended. The driver almost immediately invokes the VAX/VMS macro IOFORK.

IOFORK calls the VAX/VMS routine EXE\$IOFORK. EXE\$IOFORK saves the driver context (R3, R4, and PC) in the UCB fork block and inserts the address of the fork block (R5) in the device's fork queue. EXE\$IOFORK then returns control to the driver's interrupt service routine, which dismisses the interrupt.

When the fork dispatcher reactivates the driver fork process, the driver performs any necessary UNIBUS adapter clean-up operations, such as data path purging and deallocation of UNIBUS adapter resources used in the DMA transfer.

10.6.1 Purging the Data Path

Driver fork processes that use buffered data paths must purge the data path after the DMA transfer is complete. The driver invokes the macro PURDPR, which in turn calls the VAX/VMS routine IOC\$PURGDATAP. This routine takes the following steps to purge the data path:

- Saves the contents of R4 on the stack
- Locates the channel request block as follows:
$$R5 \longrightarrow UCB \longrightarrow CRB$$
- Obtains the starting address of UNIBUS adapter register space and stores it in R2
- Extracts the number of the data path to be purged from the channel request block and loads it into R1
- Stores the address of the data path in R4
- Purges the data path (performed by the UNIBUS adapter). The routine then modifies R0 through R2 to contain the following information:

R0 Success/failure status. If the purge completes without error, the routine sets SS\$NORMAL in this register. If a data path error does occur, R0 is clear and the hardware is reset.

R1 Contents of the data path register

R2 Address of the first UNIBUS adapter map register

The address of the channel request block remains in R3. This address, along with the information in R1 and R2, is used as input to the error-logging routine in the event of a data path error.

CODING FOR UNIBUS DMA TRANSFERS

- Restores the information stored on the stack to R4 and returns to PURDPR.

If a data path error occurs during a data path purge, the driver should retry the entire DMA transfer.

10.6.2 Releasing a Buffered Data Path

A driver fork process releases a buffered data path by invoking the VAX/VMS macro RELDPR. RELDPR calls a VAX/VMS routine IOC\$RELDATAP that determines which data path was assigned to the driver fork process and releases the data path to a waiting driver. The driver must be executing at fork IPL.

The data path number is stored in the channel request block. IOC\$RELDATAP locates it as follows:

UCB → CRB → data path number

If the data path is permanently assigned to a device, IOC\$RELDATAP does not release the data path. Otherwise, the data path number in the channel request block (CRB\$L_INTD + VEC\$B_DATAP) is zeroed. The IOC\$RELDATAP routine attempts to dequeue a waiting driver fork process from the data path wait queue stored in the adapter control block as follows:

UCB → CRB → ADP → data path wait queue

If another driver is waiting for a buffered data path, IOC\$RELDATAP grants that driver fork process the data path, restores its driver context from its UCB fork block, and transfers control to the saved driver PC. When IOC\$RELDATAP can allocate no more data paths, the routine returns to the driver that released the data path. This diversion of driver processing is transparent to the driver fork process.

If the data path wait queue is empty, IOC\$RELDATAP marks the data path as available in the I/O data base and returns control to the driver.

10.7 RELEASING UBA MAP REGISTERS

A driver fork process releases a set of UNIBUS adapter map registers by invoking the VAX/VMS macro RELMPR. RELMPR calls the VAX/VMS routine IOC\$RELMAPREG that releases map registers in a manner similar to that in which data paths are released. The channel request block records the map register numbers assigned to the device. The number of the first map register and the number of map registers are located as follows. The driver must be executing at fork IPL.

UCB → CRB → number of the first map register

UCB → CRB → number of map registers allocated

IOC\$RELMAPREG releases the map registers by setting the corresponding bits in the map register allocation bit map, which it locates as follows:

UCB → CRB → ADP → map register bit map

CODING FOR UNIBUS DMA TRANSFERS

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

- Dequeues the fork block and restores driver context
- Fills the map register request, if possible
- Reactivates the driver fork process at the instruction following the driver's request for map registers
- Returns to the driver fork process

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

CHAPTER 11

CODING INTERRUPT SERVICE ROUTINES

The driver prologue table of most device drivers contains, in the reinitialization section established using the DPT_STORE macro, the address of one or more interrupt service routines. Each interrupt service routine corresponds to an interrupt vector on the UNIBUS. You specify the UNIBUS vector address in the CONNECT command to the SYSGEN utility, as described in Chapter 14.

Most interrupt service routines in device drivers perform the following functions:

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

Figure 11-1 illustrates the general flow of interrupt handling. The remaining sections of this chapter describe the handling of solicited and unsolicited interrupts in further detail.

11.1 DELIVERING A DEVICE INTERRUPT TO A DRIVER

When a UNIBUS device requests a hardware interrupt, the device puts a vector address on UNIBUS lines and the vector address is loaded into a UNIBUS adapter register. When the processor executes at an interrupt priority level below the hardware interrupt level of the device, the following sequence occurs:

- The processor saves the PC and PSL of the currently executing code on the interrupt stack and transfers control to the VAX/VMS UNIBUS adapter interrupt service routine.
- The UNIBUS adapter service routine then saves R0 through R5 on the stack and, using a JMP instruction, transfers control to executable code in the channel request block that the driver loading procedure has associated with the interrupting vector.

CODING INTERRUPT SERVICE ROUTINES

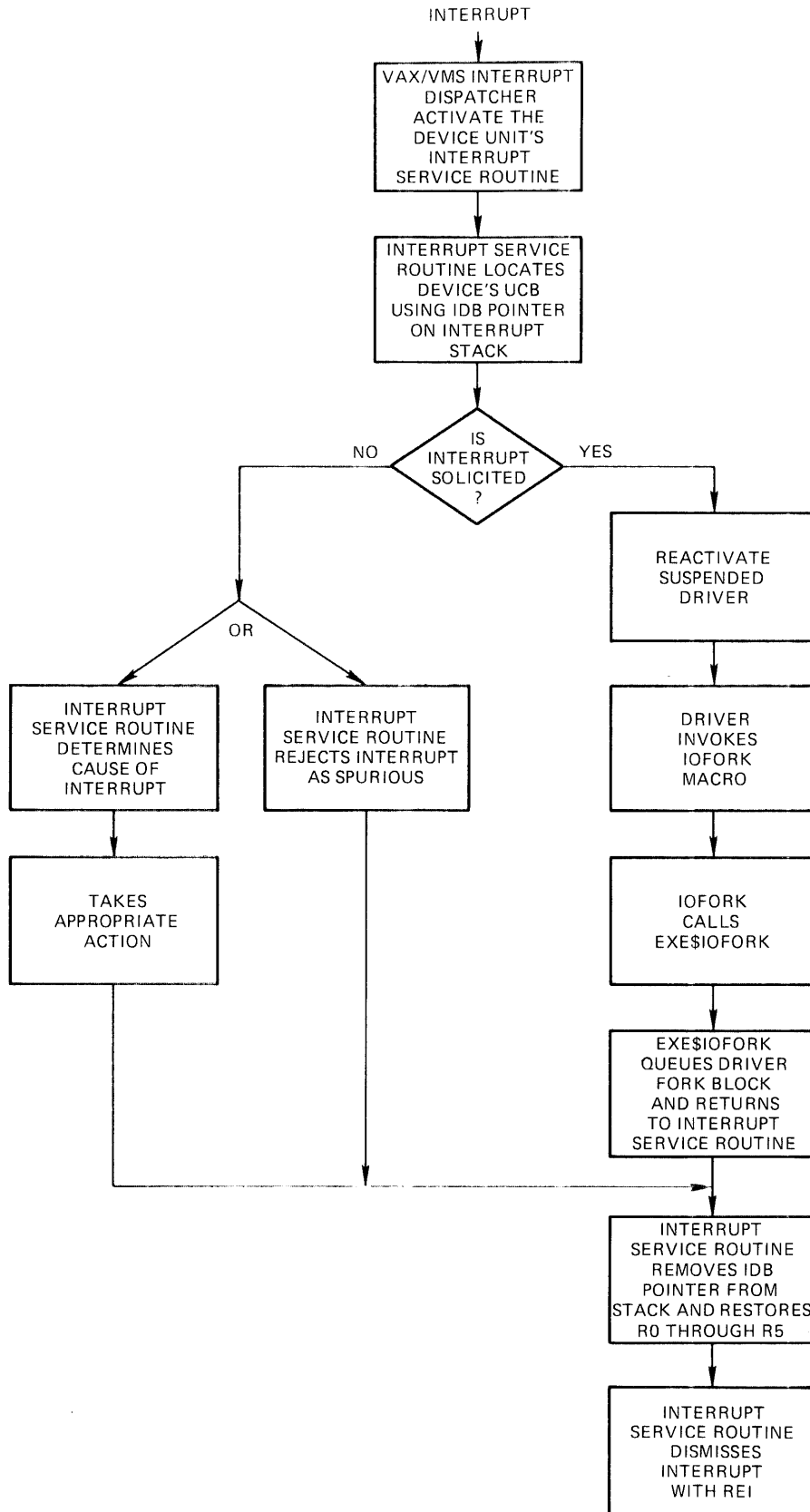


Figure 11-1 Interrupt Handling Flow

CODING INTERRUPT SERVICE ROUTINES

The CRB interrupt dispatching field (CRB\$L_INTD+2) contains the following executable instruction:

```
JSB @#address-of-driver-isr
```

The driver loading procedure writes these instructions into each channel request block as the procedure creates the control blocks. The driver loading procedure obtains the address of the interrupt service routine from the reinitialization portion of the driver prologue table. If the device has two interrupt vectors, for example, and its driver specifies two interrupt service routine addresses in the DPT reinitialization section, the driver loading procedure creates a channel request block with two interrupt dispatching fields.

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

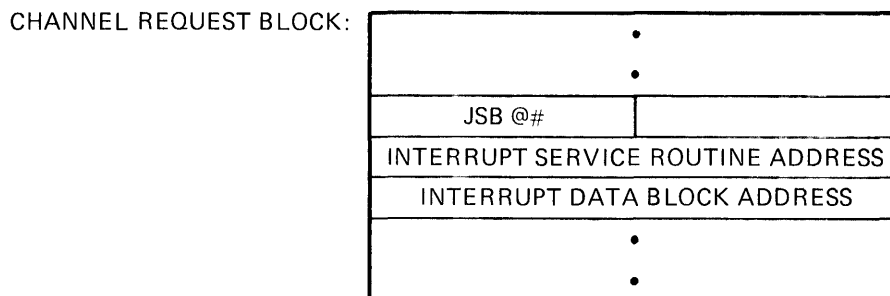


Figure 11-2 Channel Request Block
Containing an Interrupt Service Routine Address

11.2 INTERRUPT CONTEXT

When the UNIBUS adapter interrupt service routine calls a driver interrupt service routine, execution context is as follows:

- R0 through R5 are saved on the stack.
- System address space is mapped. The service routine can gain access to appropriate control blocks in the I/O data base.
- IPL is at hardware device interrupt level.
- The processor is running in kernel mode.
- The processor is running on the interrupt stack.

CODING INTERRUPT SERVICE ROUTINES

The UNIBUS adapter interrupt service routine does not load any registers for the driver interrupt service routine; however, the stack contains the following information:

Stack Location	Content
0(SP)	Pointer to the address of the interrupt data block
4(SP) through 24(SP)	Saved R0 through R5
28(SP)	PC at the time of the interrupt
32(SP)	PSL at the time of the interrupt

11.3 SERVICING A SOLICITED INTERRUPT

When a driver fork process activates a device and expects to service a device interrupt as a result, the driver suspends fork processing and waits for an interrupt to occur. The suspended driver is represented only by the contents of the device's unit control block, which contains a description of the I/O request and the driver fork process. When the driver regains control from the interrupt service routine, only R3, R4, R5, and the PC address are restored to their previous state by the interrupt service routine.

In the sequence below, a driver interrupt service routine returns control to the waiting driver:

- First the interrupt service routine obtains the address of the device's unit control block from the interrupt data block, as follows:
0(SP) → CRB → IDB\$L_OWNER → UCB for the device
- The service routine then tests the software interrupt expected bit in the UCB status word (UCB\$V_INT in UCB\$W_STS). If the bit is set, the driver is waiting for an interrupt from this device.
- The interrupt service routine restores R5 of the driver fork process with the address of the UCB fork block. It restores R3 and R4 of the driver process using two fields from the UCB fork block, UCB\$L_FR3 and UCB\$L_FR4, respectively.
- Finally the interrupt service routine transfers control to the driver PC address saved in the UCB fork block at UCB\$L_FPC by issuing a JSB instruction.

The restored driver can execute a few instructions in the context of the interrupt, such as copying device status information from the device registers into the device's UCB. Before completing the I/O operation, however, the driver routine creates a fork process to lower its execution IPL to driver fork level instead of continuing execution at hardware device interrupt IPL. The driver routine creates a fork process by invoking the VAX/VMS macro IOFORK, as described in Section 12.1.1.

CODING INTERRUPT SERVICE ROUTINES

IOFORK calls the VAX/VMS routine EXE\$IOFORK. EXE\$IOFORK inserts the UCB fork block describing the driver process in the appropriate fork queue and returns control to the driver interrupt service routine. The interrupt service routine then performs the following steps:

- Removes the IDB pointer from the stack
- Restores R0 through R5
- Dismisses the interrupt with an REI instruction

11.4 SERVICING AN UNSOLICITED INTERRUPT

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

Other changes of device status occur when the device has not been activated by a device driver. The device reports these changes by requesting unsolicited interrupts. For example, when a user types on a terminal that is not attached to a process, the terminal requests an interrupt that is fielded by the terminal driver. As a result of the interrupt, the terminal driver causes the login procedure to be invoked for the user at the terminal.

Another example of an unsolicited interrupt is one that the unit requests when an operator changes the volume on a disk drive. The disk driver services the interrupt by altering volume and unit status bits in the disk device's unit control block.

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

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

The driver interrupt service routine decides whether an interrupt is solicited or not by examining the software interrupt expected bit in the UCB status word (UCB\$V_INT in UCB\$W_STS). If the interrupt is unsolicited, the driver can reject the interrupt with the following code sequence:

- Remove the IDB pointer from the stack
- Restore R0 through R5
- Dismiss the interrupt with an REI instruction

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

Drivers should avoid creating a fork process to handle unsolicited interrupts from busy devices. The unit control block of a busy device may contain the active fork block of a previously created driver fork process. If an unsolicited interrupt service routine should create a fork process to handle its request, it may destroy the driver fork

CODING INTERRUPT SERVICE ROUTINES

context currently stored there. Drivers should always handle this type of unsolicited interrupt at hardware device IPL.

11.4.1 Examples Of Unsolicited Input Handling

A card reader device requests an unsolicited interrupt if any user turns the reader online. Once the card reader driver interrupt service routine determines that the interrupt is unsolicited, the routine analyzes the interrupt, as in the following example:

- It obtains the address of the control/status register using the interrupt data block pointed to by the address on the top of the interrupt stack, as follows:

0(SP) → CRB → IDB → CSR address

- It confirms that the reader has just been placed online by examining the online bit in the control/status register.
- It examines the reference count field of the device's unit control block (UCB\$W_REFC) to determine whether a process has assigned or allocated the device. If the count is nonzero, the interrupt service routine removes the IDB pointer from the stack, restores R0 through R5, and dismisses the interrupt with an REI instruction.
- If the reference count is zero, the interrupt service routine clears the status bit in the control/status register.
- It confirms that the job controller has not received a message about the device's being online by testing the job-attached bit in the UCB status word (UCB\$V_JOB in UCB\$W_STS).
- If the job-attached bit is not set, it sets the job-attached bit and creates a fork process that is to send a message to the job controller. The VAX/VMS routine that creates the fork process returns to the driver's interrupt service routine.
- Finally, the interrupt service routine removes the IDB pointer from the stack, restores R0 through R5, and dismisses the interrupt with an REI instruction.

Another example of unsolicited interrupt processing occurs in a device driver for a multiunit controller. When the operator removes a disk volume, the disk drive requests an interrupt. The driver interrupt service routine must determine what drive unit requested the interrupt, obtain drive status from the drive's control/status register, and then decide whether the interrupt was solicited. If the interrupt is unsolicited, the driver service routine calls its unsolicited interrupt routine. The routine checks the status of the volume, as described in the following steps:

- It sets a bit in the unit control block to indicate that the unit is online (UCB\$V_ONLINE in UCB\$W_STS).
- If the UCB volume valid bit is set (UCB\$V_VALID in UCB\$W_STS), the routine tests the volume valid status bit in a device register to determine whether the volume status has changed. If the volume is no longer valid, the routine clears the UCB volume valid bit.
- Finally, the routine returns to the normal driver interrupt service routine.

CODING INTERRUPT SERVICE ROUTINES

The driver interrupt service routine then polls the other device units on the controller to determine whether any other units requested interrupts while the first interrupt was being processed. When no unit requires interrupt servicing, the routine removes the IDB pointer from the stack, restores registers R0 through R5, and dismisses the interrupt with an REI instruction.

CHAPTER 12

COMPLETING THE I/O REQUEST

Once a driver has activated the device and invoked the wait for interrupt macro, the driver remains suspended until one of the following events occurs:

- The device requests an interrupt
- The device times out

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

If the device does not request an interrupt within the designated time interval, the system transfers control to the driver's timeout handler. The address of the timeout handler is specified as an argument to the wait for interrupt macro invocation.

12.1 I/O POSTPROCESSING

Once the driver interrupt service routine has handled an interrupt, it transfers control to the driver by issuing a JSB instruction. At this point, the driver is executing in interrupt context. If the driver were to continue executing in interrupt context, it would lock out most other processing on the processor including the handling of hardware interrupts. To restore the driver to the context of a driver fork process, the driver invokes the VAX/VMS macro IOFORK. Once the fork process has been created and dispatched for execution, it executes the driver code that completes the processing of the I/O request.

12.1.1 EXE\$IOFORK

IOFORK is a macro that generates a call to the VAX/VMS routine EXE\$IOFORK. EXE\$IOFORK converts the driver context from that of an interrupt service routine to the context of a driver fork process in the following steps:

- It disables software timeouts by clearing the timeout enable bit in the UCB status word (UCB\$V_TIM in UCB\$W_STS).
- It saves R3 and R4 of the current driver context in the UCB fork block (UCB\$L_FR3 and UCB\$L_FR4).

COMPLETING THE I/O REQUEST

- EXE\$IOFORK then saves the current driver PC in the UCB fork block (UCB\$L_FPC). The driver PC is the first longword on the stack upon entry to EXE\$IOFORK as a result of the JSB instruction.
- It obtains the fork IPL of the device from the UCB (UCB\$B_FIPL).
- It inserts the address of the UCB fork block (R5) into the fork queue corresponding to the driver fork IPL.
- Finally, if the fork block is the first entry in the fork queue, EXE\$IOFORK requests a software interrupt at driver fork IPL.

The steps listed above move the critical driver fork process context into the UCB fork block; that is, they save R3 through R5 and the driver PC address. The driver fork process resumes processing when the VAX/VMS fork dispatcher dequeues the UCB fork block from the fork queue and reactivates the driver at driver fork IPL.

12.1.2 Completing an I/O Request

When VAX/VMS reactivates a driver fork process by dequeuing the fork block, the driver resumes processing of the I/O operation. If the device has completed the I/O operation without errors, the driver fork process for a DMA device proceeds as follows:

- Purges the buffered data path
- Releases the buffered data path
- Releases map registers
- Releases the controller
- Saves the status code, transfer count, and device-dependent status that is to be returned to the user process in an I/O status block
- Returns control to the operating system

Chapter 10 discusses the first three steps listed above because they relate to UNIBUS DMA transfers. The sections that follow describe the remaining three steps.

12.1.2.1 Releasing the Controller - To release the controller channel, the driver code invokes the VAX/VMS macro RELCHAN. RELCHAN calls the VAX/VMS routine IOC\$RELCHAN. If another driver is waiting for the controller channel, IOC\$RELCHAN grants that driver fork process the channel, restores its driver fork context from its UCB fork block, and transfers control to the saved PC. When no more drivers are awaiting the channel, IOC\$RELCHAN returns control to the driver fork process that released the channel. (The driver must be running at fork IPL.)

COMPLETING THE I/O REQUEST

12.1.2.2 **Saving Status, Count, and Device-Dependent Status** - To save the status code, transfer count, and device-dependent status, the driver performs the following steps:

- It loads a success status code (SS\$_NORMAL) into bits 0 through 15 of R0.
- If the I/O operation performed by the device is a transfer function, the driver loads the number of bytes transferred into the high-order 16 bits of R0, that is, into bits 16 through 31.
- The driver then loads device-dependent status information, if any, into R1. R0 and R1 are the status values that VAX/VMS returns to the user process in the I/O status block specified in the original Queue I/O Request system service. If the user specifies no I/O status block, VAX/VMS makes no use of R0 and R1.

12.1.2.3 **Returning to the Operating System** - Finally, the driver returns to the system by invoking the VAX/VMS macro REQCOM to complete the I/O request. REQCOM calls the VAX/VMS routine IOC\$REQCOM. IOC\$REQCOM locates the address of the I/O request packet corresponding to the I/O operation in the device's UCB (UCB\$_IRP). It then writes the two longwords of completion status contained in R0 and R1 into the media field of the I/O request packet (IRP\$_MEDIA and IRP\$_MEDIA+4).

IOC\$REQCOM then inserts the I/O request packet in the I/O postprocessing queue. If the packet is the only entry in the postprocessing queue, IOC\$REQCOM requests a software interrupt at IPL\$_IOPOST so the postprocessing begins when IPL drops below IPL\$_IOPOST.

If the error-logging bit is set in the device's unit control block (UCB\$_ERLOGIP in UCB\$_STS), IOC\$REQCOM obtains the address of the error message buffer from the unit control block (UCB\$_EMB). It then writes the following information into the error buffer:

- Final device status (UCB\$_DEVSTS)
- Final error count (UCB\$_ERTCNT)
- Two longwords of completion status (R0 and R1)

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

If any I/O request packets are awaiting driver processing, IOC\$REQCOM performs the following steps:

- Dequeues a packet
- Creates a new driver fork process
- Activates the driver at the driver's start I/O routine

Otherwise, IOC\$REQCOM clears the unit busy bit in the device's UCB status word (UCB\$_BSY in UCB\$_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.

COMPLETING THE I/O REQUEST

12.2 TIMEOUT HANDLERS

VAX/VMS transfers control to the driver's timeout handler if a device unit does not request an interrupt within the time limit specified in the wait for interrupt macro. The VAX/VMS timer routine scans device unit control blocks once every second to determine whether a device has timed out.

When the timer routine locates a device that has timed out, the routine calls the device's timeout handler by performing the following steps:

- It disables expected interrupt and timeout on the device by clearing bits in the device's UCB status field (UCB\$V_INT and UCB\$V_TIM in UCB\$W_STS).
- It sets the device timeout bit in the UCB status field (UCB\$V_TIMEOUT in UCB\$W_STS).
- It sets IPL to hardware device interrupt IPL (UCB\$B_DIPL).
- It restores the saved R3 and R4 of the driver fork process from the UCB fork block (UCB\$L_FR3 and UCB\$L_FR4).
- It restores R5 (address of the UCB fork block).
- It computes the address of the driver's device timeout routine from the saved PC in the UCB fork block (UCB\$L_FPC).
- It calls the device timeout routine with a JSB instruction.

During power failure recovery, VAX/VMS forces a device timeout by altering the timeout field (UCB\$L_DUETIM) of a unit control block if the device UCB records that the unit is waiting for an interrupt or timeout (UCB\$V_INT and UCB\$V_TIM set in UCB\$W_STS). The timeout handler can perceive that a power failure recovery is occurring by examining the power bit (UCB\$V_POWER in UCB\$W_STS) in the unit control block.

A timeout handler usually performs either of three functions:

- Retries the I/O operation unless a retry count is exhausted
- Aborts the I/O request
- Sends a message to an operator mailbox and resumes waiting for a subsequent interrupt or timeout

12.2.1 Retrying the I/O Operation

Some devices may retry an I/O operation after a timeout. For example, a disk driver might take the following steps after a transfer timeout:

- Set IPL to driver fork level. Note that this IPL must not drop below IPL\$_TIMER, the level at which interval timer interrupts occur.
- Release map registers, data path, and controller data channel.

COMPLETING THE I/O REQUEST

- If a power failure occurred, load the I/O request packet address into R3 and reload the following I/O request packet fields into the corresponding UCB fields and branch to the start I/O routine:

```
UCB$W_BCNT
UCB$W_BOFF
UCB$L_SVAPTE
```

The above steps result in a total retry of the transfer.

- If no power failure has occurred and the device driver supports error-logging, call ERL\$DEVICTMO to log the device timeout
- If the retry count is not exhausted, decrease the count, clear the UCB timeout bit in UCB\$W_STS, and retry the operation.
- If the retry count is exhausted, set the error code, perform a normal abort I/O clean-up operation, and invoke REQCOM.

12.2.2 Aborting the I/O Request

A driver's timeout routine aborts the I/O request when it exhausts its retry count, or when it determines, upon timeout, that a cancel I/O was requested (UCB\$V_CANCEL is set in UCB\$W_STS).

A device driver timeout handler can abort the I/O request using the following sequence of steps:

- If appropriate to the device and controller, the handler clears the device control/status register.
- The handler then invokes the following VAX/VMS macro to lower IPL to device fork level:

```
SETIPL UCB$B_FIPL(R5)
```

The resulting IPL must not drop below the interval timer IPL.

- The handler releases UNIBUS adapter resources and the controller data channel, if necessary.
- It loads an error status code into the low word of R0.
- It clears bits 16 through 31 in R0 to indicate that no data was transferred.
- It invokes the VAX/VMS macro REQCOM, described in Section 12.1.2.3, to complete the I/O request processing.

Since the device can interrupt driver timeout processing at fork IPL, the interrupt service routine should check the interrupt expected bit (UCB\$V_INT) before handling the interrupt. The operating system clears this bit before it calls the driver's timeout routine.

COMPLETING 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 mailbox and then goes back into a wait for interrupt or timeout state:

- It invokes the following VAX/VMS macro to lower IPL to driver fork level:

```
SETIPL UCB$B_FIPL(R5)
```

- It checks the cancel I/O bit in the UCB status word (UCB\$V_CANCEL in UCB\$W_STS).

- If UCB\$V_CANCEL is set, the timeout handler performs the following:

```
-- Loads the abort status code (SS$_ABORT) into the low word of R0
```

```
-- Writes other status information into R0 and/or R1
```

```
-- Invokes the VAX/VMS macro REQCOM to complete the I/O request processing
```

- If UCB\$V_CANCEL is not set, the timeout handler performs the following:

```
-- Saves R3 and R4 on the stack
```

```
-- Loads an OPCOM message code, such as MSG$_DEVOFFLIN, into R4
```

```
-- Loads the address of the operator mailbox (SYS$GL_OPRMBX) into R3
```

```
-- Calls a VAX/VMS routine to place the message in the operator mailbox, as follows:
```

```
JSB G^EXE$SNDEVMSG
```

```
-- Restores R3 and R4
```

- The timeout handler then invokes the VAX/VMS macro DSBINT to raise IPL to IPL\$_POWER, thereby locking out all interrupts from software and hardware.

- Finally, the timeout handler invokes the VAX/VMS macro WFIKPBH to wait for another interrupt or timeout.

CHAPTER 13

CODING INITIALIZATION, CANCEL I/O, AND ERROR-LOGGING ROUTINES

Drivers normally contain initialization, cancel I/O, and error-logging routines. The driver prologue table specifies the address of initialization routines. The driver dispatch table contains the addresses of the cancel I/O and error-logging routines. Whether these routines are required depends on the type of device.

13.1 INITIALIZATION ROUTINES

Most device controllers and device units require initialization under the following circumstances:

- When the driver loading procedure loads a device driver for the controller and device units
- During recovery from a power failure

Initialization routines ready controllers and device units for operation. Depending on the device characteristics, initialization routines perform any of the actions listed below:

- Enable controller interrupts
- Clear error status bits in device registers
- Initiate a device operation such as clearing a drive or acknowledging a pack
- Store values in UCB fields that cannot be addressed with a `DPT_STORE` macro; that is, fields more than 256 bytes from the start of the unit control block
- Permanently allocate UNIBUS adapter resources, as described in Chapter 10
- Set the online bit (`UCB$V_ONLINE` in `UCB$W_STS`) in the unit control block
- Fill in `IDB$L_OWNER` for single-unit devices such as a line printer

13.1.1 Initialization During Driver Loading

The driver loading procedure¹ loads and reloads drivers in the following steps:

- It loads a driver into nonpaged system memory. If the procedure is loading the driver for the first time since the system was bootstrapped, it creates unit control blocks, a channel request block, an interrupt data block, and a device data block.
- According to the DPT_STORE macro invocations in the driver, the loading procedure writes the addresses of initialization routines in the channel request block of the controller and initializes other fields in other control blocks.
- It raises IPL to IPL\$_POWER to block all interrupts.
- It calls the controller initialization routine, if one exists, for every device controller associated with the driver.
- It calls the unit initialization routine, if one exists, for every device unit associated with the driver.

13.1.2 Initialization During Recovery from a Power Failure

During powerfail recovery procedures, the operating system locates every unit control block in the I/O data base. Each unit control block points to a channel request block for the device's controller. The channel request block contains the address of the controller initialization routine, if one was specified. The system uses the following chain of pointers to locate the address of the initialization routine:

DDB → UCB → CRB → controller initialization routine

The operating system calls the initialization routine for each controller if one was specified in a DPT_STORE macro for the CRB\$_INTD+VEC\$_INITIAL of the channel request block.

Next, the system checks for a device unit initialization routine. First, the system examines the unit initialization field in the driver dispatch table (DDT\$_UNITINIT). If the field does not contain an address, the system checks the channel request block using the following chain of pointers:

DDB → UCB → CRB → device unit initialization routine

MASSBUS drivers store unit initialization routines only in the driver dispatch table.

If either the channel request block or the driver dispatch table contains a nonzero address for such a routine, the system calls the routine to initialize the device unit.

1. The SYSGEN commands CONNECT and AUTOCONFIGURE call controller and unit initialization routines once for each controller and device unit. The LOAD command does not call controller or unit initialization routines, whereas the RELOAD command calls only the controller initialization routine.

CODING INITIALIZATION, CANCEL I/O, AND ERROR-LOGGING ROUTINES

13.1.3 Initialization Context

The VAX/VMS operating system always calls controller and unit initialization routines with IPL raised to IPL\$POWER. The high IPL prevents any interrupts from reaching the processor while initialization is occurring. The initialization routines must not lower IPL. The system calls initialization routines with a JSB instruction; the routines return by executing an RSB instruction.

Controller initialization routines are device-dependent. For example, a card reader controller initialization routine might enable interrupts from the device by setting the interrupt enable bit in the device's control/status register. A disk controller initialization routine, on the other hand, might enable interrupts and initialize all unit status registers.

If a device needs permanently allocated UNIBUS adapter resources, a controller initialization routine can call VAX/VMS UNIBUS adapter resource management routines to allocate the resources. Then, the initialization routine can set bits in the CRB UNIBUS adapter resource description fields (for example, VEC\$V_PATHLOCK in CRB\$L_INTD+VEC\$B_DATAPATH).

At the time of a call to a controller initialization routine, the registers contain the following values:

Register	Value
R4	Address of the control/status register
R5	Address of the interrupt data block that describes the controller
R6	Address of the device data block associated with the controller
R8	Address of the channel request block for the controller

Device unit initialization routines are useful for initializing device-dependent fields in the unit control block. For example, disk initialization routines can also set disk drive parameters (such as number of cylinders) in the unit control block and wait for online units to spin up to speed. Unit initialization routines must set the online bit in the unit control block (UCB\$V_ONLINE) to declare the unit to be online.

At the time of a call to a device unit initialization routine, the registers contain the following values:

Register	Value
R3	Address of the primary control/status register
R4	Address of the secondary control/status register; R4 is equal to R3 if there is no secondary CSR
R5	Address of the device's unit control block

If driver initialization routines modify R4 through R11, the routines must save the contents of the registers before use and restore them before returning to the operating system.

13.2 CANCEL I/O ROUTINE

VAX/VMS routines call the cancel I/O routine in a device driver under the following circumstances:

- When a process issues a Cancel I/O on Channel system service
- When a process deallocates a device and no process I/O channels are assigned to the device
- When a process deassigns a channel from a device
- When the command language interpreter performs cleanup operations as part of image termination by canceling all pending I/O requests for the image and closing all image-related files open on process I/O channels

The VAX/VMS routine EXE\$CANCEL locates the unit control block for the device associated with a process I/O channel from a pointer in the channel request block, as follows:

channel index number → CCB → UCB address

EXE\$CANCEL takes the following steps:

- Raises IPL to fork level
- Removes all I/O request packets associated with the process from the device's I/O request packet wait queue
- Sets the status code SS\$_CANCEL in IRP\$L_MEDIA
- For buffered I/O read operation, clears the buffered read function bit (IRP\$V_FUNC) in IRP\$W_STS
- Inserts the I/O packets removed from the packet wait queue into the I/O postprocessing queue
- If the I/O postprocessing queue is empty, requests a software interrupt

Then, EXE\$CANCEL calls the cancel I/O routine specified in the driver dispatch table of the associated device driver. EXE\$CANCEL locates the routine using the following chain of pointers:

UCB → DDB → DDT → address of the cancel I/O routine

The cancel I/O routine gives the driver an opportunity to prevent further device-specific processing of the I/O request currently being processed on the device.

13.2.1 Context of a Cancel I/O Routine

When EXE\$CANCEL calls the cancel I/O routine, IPL is at driver fork IPL so that the routine can read and modify the device's unit control block. Registers at the time of the call contain the following values:

Register	Value
R2	Negated value of the channel index number
R3	Address of the current I/O request packet
R4	Address of the process control block of the process for which the Cancel I/O on Channel system service is being performed
R5	Address of the device's unit control block

13.2.2 Drivers That Need No Cancel I/O Routine

Some devices do not need any device-dependent processing performed for an I/O request; you can omit the CANCEL argument from the DDTAB macro. In this case, the DDTAB macro expansion loads the address of the VAX/VMS routine IOC\$RETURN into the appropriate position in the driver dispatch table. The routine IOC\$RETURN executes a single RSB instruction.

13.2.3 Device-Independent Cancel I/O Routine

Drivers can specify the VAX/VMS routine IOC\$CANCELIO as the value of the CANCEL argument in the DDTAB macro invocation. IOC\$CANCELIO cancels I/O to a device in the following device-independent manner:

- It confirms that the device is busy by examining the device busy bit in the UCB status word (UCB\$V_BSY in UCB\$W_STS).
- It locates the process identification field in the I/O packet currently being processed on the device using the following chain of pointers:

UCB → IRP → process identification field

IOC\$CANCELIO confirms that the field (IRP\$L_PID) contains the same value as the corresponding field in the process control block (PCB\$L_PID).

- It confirms that the specified channel index number is the same as the value stored in the I/O request packet channel index field (IRP\$W_CHAN).
- It sets the cancel I/O bit in the UCB status word (UCB\$V_CANCEL in UCB\$W_STS).

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

13.2.4 Device-Dependent Cancel I/O Routines

Drivers that include their own cancel I/O routines must perform the first three steps of IOC\$CANCELIO listed in Section 13.2.3 to determine whether the I/O request being processed originates from the process canceling I/O on a channel. If the three checks succeed, the cancel routine can proceed in a device-specific manner.

13.3 ERROR LOGGING ROUTINES

The operating system supplies two routines that drivers can call to allocate and fill error-logging buffers after a device error or timeout occurs:

- ERL\$DEVICERR
- ERL\$DEVICTMO

Both routines expect to find the address of the device unit control block in R5. Drivers must call them at fork IPL. Each routine performs the following steps:

- It allocates an error log buffer of the length specified in the device's driver dispatch table. It uses the following chain of pointers to locate the buffer length:

UCB → DDB → DDT → length of error log buffer

- It loads into the buffer fields from the unit control block, the I/O request packet, and the device data block.
- It loads the address of the error message buffer location where device register contents are to be stored.
- It calls a register dump routine in the device driver. It locates the routine using the following chain of pointers:

UCB → DDB → DDT → register dump routine address

Specify the address of a register dump routine with the value of the REGDMP argument to the DDTAB macro invocation.

The register dump routine can expect the following registers to be loaded:

Register	Content
R0	Address of the buffer
R4	Address of the control/status register if the driver used the WFIKPCH macro to wait for an interrupt or timeout
R5	Address of the device's unit control block

The dump routine should save and restore R3 through R11 if the routine requires their use.

CODING INITIALIZATION, CANCEL I/O, AND ERROR-LOGGING ROUTINES

The driver register dump routine should fill the buffer as follows:

- Write a longword value representing the number of device registers to be written into the buffer
- Move device register longword values into the buffer following the register count longword

The routine must store the contents of each device register to be logged in a longword in the buffer. For example, the following instruction stores the contents of the device register:

```
MOVZWL TD_STATUS(R4), (R0)+
```

A driver that supports error-logging must satisfy the following prerequisites:

- It must use the error log extension to the unit control block.
- It must ensure that `DDT$W_ERRORBUF` is large enough to accommodate `EMB$L_DV_REGS $\overline{AV}+4$` plus one longword for each register to be dumped
- Its driver prologue table must set the device characteristic `DEV$V_ELG` in `UCB$_DEVCHAR`.

CHAPTER 14

LOADING A DEVICE DRIVER

You can load a user-written device driver any time after the system is bootstrapped. If the driver contains an error and the error does not crash or corrupt the operating system, you can correct the error and reload a new version of the driver.

14.1 IN PREPARATION FOR LOADING

To prepare a device driver for loading, take the following steps:

- Write the device driver in one or more source files. If the driver comprises multiple source files, you must insert a .PSECT directive before any generated code in all files except the file that contains the DPTAB and DDTAB macro invocations. The following .PSECT must be used:

```
.PSECT $$$115_DRIVER, LONG
```

If a single source file contains the driver, you must not specify any .PSECT directives. The declaration of the DPTAB and DDTAB macros establish driver program sections correctly.

- Assemble the source file(s) with the system macro library (SYS\$LIBRARY:LIB.MLB). For example:

```
$ MACRO MYDRIVER.MAR+SYS$LIBRARY:LIB.MLB/LIBRARY
```

- Link the object file with the VAX/VMS global symbol table, which is located in SYS\$SYSTEM and called SYS.STB. If the driver consists of multiple source files, you must specify the file that contains the driver prologue table as the first file in the list. The linker options file must contain a BASE statement specifying a zero base for the executable image. The following is an example of the creation of the options file and the LINK command used to link a driver:

```
$ CREATE MYDRIVER.OPT  
BASE=0  
(CTRLZ)  
$ LINK /NOTRACE MYDRIVER1C,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.

LOADING A DEVICE DRIVER

14.2 LOADING THE DRIVER

Once the driver has linked correctly, it is ready to be loaded. To load the driver into system virtual memory, run the SYSGEN utility from the system manager's account or from an account having Change Mode to Kernel and Change Mode to Executive privileges using the following command:

```
$ RUN SYS$SYSTEM:SYSGEN
```

SYSGEN responds with a prompt and waits for further input:

```
SYSGEN>
```

The VAX/VMS System Manager's Guide describes the full set of SYSGEN commands. The sections that follow describe those commands SYSGEN uses to load drivers:

- LOAD (requires Change Mode to Kernel (CMKRNL) and Change Mode to Executive (CMEXEC) privileges)
- CONNECT (requires CMKRNL and CMEXEC privileges)
- RELOAD (requires CMKRNL and CMEXEC privileges)
- SHOW/DEVICE (requires CMEXEC privilege)

In addition, you should understand SYSGEN's automatic configuration feature, as described in Section 14.3.

14.2.1 LOAD Command

To load a device driver and its data base, issue the LOAD command. If the controller has only a single unit attached to it, issue the CONNECT command.

Format

```
LOAD driver-file-spec
```

driver-file-spec

The file specification of the image file containing the I/O driver to be loaded. If the driver file specification is the same as the driver name of a loaded driver, the LOAD command has no effect.

SYSSYSTEM is the default device and directory name. EXE is the default file type.

Description

The driver loading procedure compares the name field in the driver prologue table of the driver being loaded with the name field in the driver prologue tables of the drivers already loaded into system memory. If no match is found, the procedure loads the new driver into contiguous pages of nonpaged pool and links the driver prologue table into the DPT linked list. If the procedure finds a match, it takes no further action.

Example

```
SYSGEN> LOAD CRDRIVER
```

This command loads the card reader driver.

LOADING A DEVICE DRIVER

14.2.2 CONNECT Command

The CONNECT command creates I/O data base control blocks for devices. The CONNECT command can also load the driver if it has not been previously loaded into system memory.

Format

```
CONNECT device-name required-quals [optional-quals]
```

Command Qualifiers

```
/ADAPTER=tr-value  
/CSR=csr-address  
/VECTOR=vector-address  
/DRIVERNAME=driver-name (optional)  
/NUMVEC=number (optional)  
/ADPUNIT=unit-number  
/MAXUNITS=number
```

Parameter

device-name

The name of the device for which control blocks are to be added to the I/O data base. Specify the device name in the following format:

```
devcu
```

dev = device code (up to 9 alphabetic characters)

c = controller designation (alphabetic)

u = unit number (in the range of 0 through 7)

For example, LPA0 specifies the line printer (dev) on controller A (c) at unit 0 (u). When specifying the device name, do not follow it with a colon (:).

The device code and controller specification must be a unique and accurate device name and controller combination. If control blocks for the specified device/controller already exist, the driver loading procedure does not create any control blocks. If the device/controller name does not accurately name a device, the procedure will create spurious control blocks.

Required Qualifiers

/ADAPTER=tr-value

The number of the SBI arbitration priority to which the UNIBUS or MASSBUS adapter is attached. The tr-value must be in the range of 0 through 15. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%0 or %X).

/CSR=csr-address

The UNIBUS address of the control/status register for the device. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%0 or %X).

/VECTOR=vector-address

The UNIBUS address of the interrupt vector for the device. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%0 or %X). Section 14.3 provides additional information on vector and CSR assignments.

LOADING A DEVICE DRIVER

Optional Qualifiers

/NUMVEC=number

The number of interrupt vectors for the device. If this qualifier is omitted, the number of vectors defaults to 1. The number specified by the /VECTOR qualifier is the address of the lowest vector. Vectors must be contiguous.

/DRIVERNAME=driver-name

The name of the driver as recorded in the driver prologue table. If the driver is not loaded when the CONNECT command is issued, CONNECT assumes the driver name is also the file name of an executable image that is located in SYS\$SYSTEM and contains a driver for the device type.

Consult the SYSGEN device table in Section 14.3.2 for the driver names of the devices supported by VAX/VMS.

/ADPUNIT=unit-number

The unit number of a device on the MASSBUS adapter. The unit number for a disk drive is the number of the plug on the drive. For magnetic tape drives, the unit number corresponds to the tape controller number.

/MAXUNITS=number

The maximum number of units attached to the system. This number determines the size of the UCB list appended to the interrupt dispatch block. If specified, this value overrides the maximum number of units designated in the driver prologue table. The maximum number of units is stored in the IDB field IDB\$W_UNITS.

Description

The I/O data base contains a linked list of driver prologue tables. The CONNECT command looks for a device driver by comparing the specified or defaulted name with the driver names in the driver prologue tables. If the driver is not present, the driver loading procedure opens the driver image and loads the file contents as it does for the LOAD command; see Section 14.2.1.

Then the loading procedure examines the I/O data base for control blocks that support the specified device. The procedure creates the following control blocks if they do not exist:

- Device data block -- the procedure creates a device data block for the generic device name/controller string specified if such a device data block does not exist.

When the procedure creates a device data block for a UNIBUS device, it also creates a channel request block and an interrupt data block.

- Unit control block -- the procedure creates a unit control block if it has just created a device data block or if a unit control block for the specified device does not exist.

After creating the control blocks, the driver loading procedure initializes fields in the control blocks according to the driver prologue table. Since the control blocks describe devices new to the system, the procedure writes fields specified in both the initialization and reinitialization portions of the driver prologue table.

LOADING A DEVICE DRIVER

If the driver loading procedure just created a channel request block and the driver specifies device unit and/or controller initialization routine addresses in DPT_STORE macro invocations (CRB\$INTD+VEC\$L INITIAL and CRB\$INTD+VEC\$L UNITINIT), the driver loading procedure calls these initialization routines for units and controllers represented by newly created control blocks. The procedure raises IPL to 31 before calling the routines so that initialization is not interrupted.

You should specify CONNECT commands with extreme caution. The driver and data base loading procedure does little error checking. If the user specifies a vector that has already been defined, the procedure rejects the CONNECT command. However, if the CONNECT command specifies an incorrect CSR address, the I/O data base is apt to become corrupted. The result is a system failure.

If the CONNECT command specifies an existing controller and a new device unit, the procedure creates a unit control block for the new unit and calls a unit initialization routine for the unit.

A CONNECT command that specifies a device name with a new controller causes the driver loading procedure to create a device data block, channel request block, interrupt data block, and unit control block and to call controller and unit initialization routines.

Example

```
SYSGEN> CONNECT LPA0 /ADAPTER=3 /CSR=%O777514 /VECTOR=%O200
```

This command loads the driver LPDRIVER, if it is not already loaded, and creates the device data base (DDB, CRB, IDB, and UCB) needed to describe LPA0.

14.2.3 RELOAD Command

The RELOAD command loads a driver and removes a previously-loaded version of that driver. The RELOAD command provides all of the functions of LOAD, except that it loads the driver regardless of whether it is already loaded.

If any of the units associated with the driver are busy, the driver cannot be reloaded; SYSGEN issues an error message.

Format

```
RELOAD driver-file-spec
```

driver-file-spec

The file specification of the image file containing the driver to be loaded.

Description

To reload the driver, the driver loading procedure compares the name field in the driver prologue table of the driver being loaded with the name field in the driver prologue tables of drivers already loaded into system memory. If no match is found, RELOAD loads the driver as described in Section 14.2.1.

LOADING A DEVICE DRIVER

If the procedure finds a match, it first confirms that the current driver can be replaced by the new driver in the following steps:

- Confirms that the DPT\$M_NOUNLOAD flag in the driver prologue table of the current driver is not set
- Calls the current driver's unload routine, if one exists, and confirms that the returned status is a success code
- Ensures that no devices that use the current driver are busy, as indicated by the UCB\$V_BSY bit set in UCB\$W_STS

If these checks succeed, the procedure replaces the current driver with the new driver. The procedure loads the new driver into contiguous pages of nonpaged system memory and searches the I/O data base for references to the driver. If any device data block refers to the driver being reloaded, the procedure reinitializes fields of the device and controller control blocks according to the reinitialization instructions in the new driver's prologue table; Chapter 7 describes the DPT reinitialization fields.

Fields that must be reinitialized when a driver is reloaded include those that contain relative addresses within the driver:

- Addresses of driver interrupt service routines
- Addresses of device unit and controller initialization routines
- Address of the driver dispatch table

Once the loading procedure has reinitialized fields, it calls the driver controller initialization routine. (It does not call the unit initialization routine.) The procedure then removes the newly replaced driver from the DPT list and deallocates the nonpaged system space the old driver occupied. Finally, the loading procedure links the address of the new driver prologue table into the DPT list.

14.2.4 SHOW/DEVICE

The SHOW/DEVICE command displays the location of a driver and the I/O data base describing its devices in system virtual memory. This command requires Change Mode to Executive privilege; Change Mode to Kernel will not suffice.

Format

SHOW/DEVICE [=driver-name]

driver-name

Name of the driver for which the information is to be displayed. If a driver name is not specified, the command displays information about all drivers and devices known to the system.

LOADING A DEVICE DRIVER

Description

The SHOW/DEVICE command displays the following information:

- Name of the driver
- The driver's starting and ending virtual addresses; the starting address is the address of the driver prologue table
- The generic device/controller name associated with the driver
- The addresses of the device data block, channel request block, and interrupt data block for the generic device/controller supported by the driver
- The unit numbers and UCB addresses for each device unit associated with the driver

Example

```
SYSGEN> SHOW/DEVICE=TMDRIVER
```

```
___DRIVER___START_____END_____DEV_____DDB_____CRB_____IDB_____UNIT_____UCB
TMDRIVER 8009DF00 8009F020
                MTA 800BA660 800BA6C0 800BA360
                                0 8009F020
                                1 8009F0C0
```

14.3 AUTOCONFIGURATION

The standard VAX/VMS system start-up file runs the SYSGEN utility to create and initialize an I/O data base that describes all supported DIGITAL peripherals in the configuration. The following command requests the SYSGEN utility to prepare a data base for all supported DIGITAL devices attached to every UNIBUS and MASSBUS:

```
SYSGEN> AUTOCONFIGURE ALL
```

To configure devices attached to the UNIBUS, SYSGEN goes through the steps described in subsequent sections of this chapter.

DIGITAL-supported devices are attached to the UNIBUS according to a table found in Appendix A of the PDP-11 Peripherals Handbook. The basic rules follow:

- A device of type A is always at a fixed and predefined CSR address; the device always interrupts at a fixed and predefined vector address; only one example of device A can be configured in each system.
- A device of type B is identical to type A except that 1 through n examples can be configured in a single system. Examples 2 through n are also located at fixed and predefined CSRs and vector addresses.
- Devices of type C (1 through n of them) are always at fixed and predefined CSR addresses; however, the interrupt vector addresses vary according to what other devices are present on the system.
- Devices of type D (1 through n of them) are at CSR addresses and vector addresses that vary according to what other devices are present on the system.

LOADING A DEVICE DRIVER

The CSR and vector addresses that vary are called floating addresses. The devices must be located in floating CSR and vector space according to the order in which the devices appear in the SYSGEN device table. This table, shown in Section 14.3.2, lists all the type A and type B devices supported by VAX/VMS. It also lists the type C and type D devices that are recognized by SYSGEN's autoconfiguration procedure.

The base of floating vector space is 300 (octal). The base of floating CSR space is 760010 (octal).

14.3.1 SYSGEN's Autoconfiguration

The SYSGEN utility automatically configures a UNIBUS adapter as follows:

- It initializes the base of floating space to 300 (octal) and 760010 (octal) for vectors and CSRs, respectively.
- It tests fixed and floating CSR address space for all known DIGITAL devices.
- When a device is found at a CSR, SYSGEN reserves floating CSR and vector space for that device, if necessary. Then, if the device is supported by VAX/VMS, SYSGEN creates and initializes an I/O data base for that supported device and loads the driver for that device.

The SYSGEN utility uses a table that lists the characteristics of all DIGITAL devices. This table indicates the following information for each device type:

- The device controller name
- The name of the device driver, and whether it is supported
- The name of the device recognized by VAX/VMS
- The interrupt vector
- The number of interrupt vectors per controller
- The address of the first device register for each controller recognized by SYSGEN (the first register is usually, but not always, the CSR)
- The number of registers per controller

14.3.2 The SYSGEN Device Table

Currently, the SYSGEN device table lists the following devices:

Name	Vector	#Vectors	Vector Alignment	CSR/Rank	#Registers	Driver Support	VAX/VMS Device Code
CR11	230			777160		CRDRIVER	CR
RK611	210			777440		DMDRIVER	DM
LP11	200			777514		LPDRIVER	LP
	170			764004			
	174			764014			
	270			764024			
	274			764034			
RL211 (controller A)	160			774400		DLDRIVER	DL
TS11 (controller A)	224			772520		TSDRIVER	MS
RX211	264			777170		DYDRIVER	DY
DC11	float	2	4	774000		no	
				774010			
				774020			
				774030			
				.			
.							
.							
				(maximum of 32 units)			

Name	Vector	#Vectors	Vector Alignment	CSR/Rank	#Registers	Driver Support	VAX/VMS Device Code
KL11 or DL11A/B (controllers B,C,....)	float	2	4	776500		no	
				776510		no	
				776520			
				.			
				(maximum of 16 units)			
DN11	float	1	4	775200		no	
				775210			
				775220			
				.			
				(maximum of 16 units)			
DMB11	float	1	4	770500		no	
				770510			
				770520			
				.			
				(maximum of 16 units)			
DR11A or DR11C	float	2	8	767600		no	
				767570			
				767560			
				.			
				(maximum of 16 units)			

Name	Vector	#Vectors	Vector Alignment	CSR/Rank	#Registers	Driver Support	VAX/VMS Device Code
PR611	float	1	8	772600 772604 772610 . . . (maximum of 8 units)		no	
PP611	float	1	8	772700 772704 772710 . . . (maximum of 8 units)		no	
DT11	float	2	8	774200 774202 774204 . . . (maximum of 8 units)		no	
DX11	float	2	8	776200 776240 776300 776340		no	
DL11C/D/E	float	2	8	775610 775620 775630 . . . (maximum of 28 units)		no	

Name	Vector	#Vectors	Vector Alignment	CSR/Rank	#Registers	Driver Support	VAX/VMS Device Code
DJ11	float	2	8	float	4	no	
DH11	float	2	8	float	8	no	
GT40	float	4	8	772000 772010		no	
LPS11	float	6	8	770400		no	
DQ11	float	2	8	float	4	no	
KW11W	float	2	8	772400		no	
DV11	float	2	8	float	4	no	
DVP11	float	2	8	float	4	no	
DV11	float	3	8	775000 775040 775100 775140		no	
LK11	float	2	8	float	4	no	
DMC11	float	2	8	float	4	XMDRIVER	XM
DZ11	float	2	8	float	4	DZDRIVER	TT
KMC11	float	2	8	float	4	no	
LPP11	float	2	8	float	4	no	
VMV21	float	2	8	float	4	no	
VMV31	float	2	8	float	8	no	
DWR70	float	2	8	float	4	no	

Name	Vector	#Vectors	Vector Alignment	CSR/Rank	#Registers	Driver Support	VAX/VMS Device Code
RLZ11 (controllers B,C,...)	float	1	4	float	4	DLDRIVER	DL
TS11 (controllers B,C,...)	float	1	4	772524 772530 772534		TSDRIVER	MS
LPA11	float	2	8	770460	8	LADRIVER	LA
	float	2	8	float		LADRIVER	LA
KW11C	float	2	8	float	4	no	
CSR Position 17, Reserved, 4 Registers							
RX11 (RX01)	float	1	4	float	4	no	
DR11 DR11B	float	1	4	float	4	no	
	124			772410			
	float	1	8	772430		no	
	float	1	8	float	4	no	

LOADING A DEVICE DRIVER

Devices not listed in the SYSGEN device table include:

- Non-DIGITAL-supplied devices with fixed CSR and vector addresses. These devices have no effect on autoconfiguration. Customer-built devices should be assigned CSR and vector addresses beyond the floating address space reserved for DIGITAL-supplied devices.
- Those DIGITAL-supplied, floating vector devices that the AUTOCONFIGURE command does not recognize. Use the CONNECT command to attach these devices to the system.

14.3.3 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:

$$\text{CFVB} + (4 * \text{number_of_vectors}) \rightarrow \text{CFVB}$$

14.3.4 Floating CSR Address Calculation

To calculate the floating CSR address of a device, SYSGEN rounds the current floating CSR base (CFCB) up to the next valid floating CSR address. Floating CSR addresses must fall on an 8-byte boundary.

SYSGEN tests the CSR address (CFCB) for the next device in the device table by executing a test word (TSTW) instruction on the address and noting whether there is a response at that address.

If the device is present, SYSGEN reserves floating CSR address space for the device by computing a new CFCB:

$$\text{CFCB} + \text{bytes_in_register_set} \rightarrow \text{CFCB}$$

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:

$$\text{CFCB} + 8 \rightarrow \text{CFCB}$$

If the device is not present, SYSGEN reserves an extra block of CSR space to indicate a change to a new device type:

$$\text{CFCB} + 8 \rightarrow \text{CFCB}$$

14.3.5 Rules for Configuration

The equations described in Sections 14.3.3 and 14.3.4 reduce to the following maxims:

- Devices with fixed CSR addresses and fixed vector addresses must be attached according to the SYSGEN device table settings.

LOADING A DEVICE DRIVER

- Devices with floating CSR or vector addresses must be attached in the order in which they are listed in the SYSGEN device table.
- An 8-byte gap must be reserved between each different type of device that is located in floating CSR address space.
- An 8-byte gap must be reserved in floating CSR address space for each device type that has no controller in its configuration.
- An extra 8-byte gap must be reserved between the KW11C and the RX11 in floating CSR address space.

14.3.6 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	350 360	760120 760130
2 TS11s	224 370	772520 772524
3 DR11Bs*	124 400 410	772410 (CSR is third register) 772430 760260
1 customer device	420 (or higher)	760320 (or higher)

When assigning floating vector addresses and registers to devices not supplied by DIGITAL, be sure to leave a generous gap between these addresses and those of DIGITAL devices, since subsequent VAX/VMS maintenance updates may add new devices to the SYSGEN device table.

CHAPTER 15

DEBUGGING A DEVICE DRIVER

DELTA and XDELTA are debugging tools that can be used to monitor the execution of user programs and the VAX/VMS operating system. When you link DELTA with a user image that runs in a nonprivileged process, DELTA is a user-mode debugging tool. When run in a privileged process, however, DELTA acts as a multimode debugger; it allows you to debug in user mode or to change to kernel mode for debugging. DELTA does not support debugging at elevated IPLs.

XDELTA is syntactically identical to DELTA but also allows you to debug code that executes at an elevated IPL. XDELTA is used for stand-alone debugging of driver code and the executive.

In the command syntaxes and dialogues contained in this chapter, red ink indicates the commands typed by the user and black ink indicates the system prompts and responses.

15.1 BOOTSTRAPPING THE SYSTEM WITH XDELTA

Under VAX/VMS, drivers are part of the operating system. You normally bootstrap the system with two boot flags set to allow you to debug with XDELTA. One flag causes the bootstrapping procedure to include XDELTA in the system. The other boot flag indicates a stop at a breakpoint in VAX/VMS initialization. Execution of the breakpoint instruction causes control to transfer to a fault handler located in XDELTA.

In addition to the normal system bootstrap command files, the VAX/VMS console floppy diskette contains two command files that bootstrap the system with XDELTA:

- DMAXDT
- DBAXDT

To bootstrap the system with XDELTA, follow the procedures in the VAX-11 Software Installation Guide with two exceptions:

- Deposit the unit number of the device in R3.
- Specify one of the command files listed above instead of the command files listed in the installation guide.

The dialogue in Figure 15-1 is an example of bootstrapping the system with XDELTA.

DEBUGGING A DEVICE DRIVER

>>>DEPOSIT R3 0	Deposit the unit number in R3.
>>>@DMAXDIT	Boot the system from DMA0. The procedure boots the processor and prompts the user from SYSBOOT.
SYSBOOT> USE 16USER.PAR	Specify a parameter file for the system.
SYSBOOT> SET BUGREBOOT 0	Request an XDELTA breakpoint after a system bugcheck.
SYSBOOT> CONTINUE	Continue with the booting operation.

Figure 15-1 Bootstrapping the System with XDELTA

After being bootstrapped, the system displays its welcoming message and halts in XDELTA, as follows:

```
1 BRK AT nnnnnnnn
```

XDELTA is waiting for input. (XDELTA never issues explicit prompts.) Usually, you proceed from this point with the following command:

```
⌘P (RET)
```

All of the XDELTA commands are described in Section 15.10.

If the operating system halts with a fatal bugcheck, the system prints the bugcheck information on the console terminal. Then, because the system parameter BUGREBOOT was set to zero, XDELTA prompts. Bugcheck information consists of the following:

- Type of bugcheck
- Register values
- Dump of one or more stacks

PC and stack content indicate how an experimental driver crashed the system. You can then examine the system state further by issuing XDELTA commands.

15.2 LOADING THE DRIVER

Once the system is running, you can log in to the system as the system manager and load the experimental driver.

To load the driver, run the SYSGEN utility and issue the appropriate LOAD and CONNECT commands. Figure 15-2 provides a sample dialogue.

The first SHOW command in Figure 15-2 causes the SYSGEN utility 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.

DEBUGGING A DEVICE DRIVER

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN>  LOAD IMA0:CYOUR.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=%0274/CSR=%0776240

SYSGEN>  SHOW /DEVICE=YRDRIVER
___Driver_____Start_____End_____Dev_____DDB_____CRB_____IDB_____Unit_____UCB_____
YRDRIVER      80060E50 80061070
                                     YRA 8005FDC0 80060B70 8005FE00
                                               O 80060B80

SYSGEN>  EXIT
```

Figure 15-2 Loading a Driver

15.3 INSERTING BREAKPOINTS IN THE SOURCE CODE

The SYSGEN command CONNECT calls controller initialization and unit initialization routines. To begin debugging the driver, you should ensure that the kernel mode debugging utility XDELTA gains control of the driver before these routines execute. This is accomplished by placing calls to the special system routine INI\$BRK within the source code of either the controller or unit initialization routines. To call INI\$BRK, give the following instruction:

```
JSB      G^INI$BRK
```

The INI\$BRK routine contains two instructions:

```
BPT
RSB
```

When the processor executes the BPT instruction, XDELTA gains control and reports the address of the breakpoint:

```
1 BRK AT nnnnnnnn
```

You can use INI\$BRK as a debugging tool and place calls to it within any part of the driver source code.

To determine the last driver PC before the breakpoint, examine the kernel stack. The stack register is register RE (hexadecimal format):

```
RE/address /address
```

Display RE to find the address of the current top of stack. Another display command (/) reveals the contents of the stack top, that is, the return address to the driver that called INI\$BRK.

DEBUGGING A DEVICE DRIVER

15.4 CALCULATING THE BASE OF DRIVER CODE

Before you debug the driver, it is a good idea to calculate the base address of driver code, as follows:

- Run the SYSGEN utility and issue the SHOW/DEVICE command. The resulting display lists the location in nonpaged pool at which SYSGEN loaded the driver.
- Consult the load map for the driver (obtained at driver link time). The driver resides in two program sections (PSECTs):

```
$$$105_PROLOGUE    driver prologue table
$$$115_DRIVER      driver code
```

The locations given in the driver code listing are offsets from \$\$\$115_DRIVER. Thus, you can calculate the base address of the driver by adding the address at which the driver was loaded to the offset associated with the PSECT \$\$\$115_DRIVER shown in the map.

If you do not have the load map, consult the driver prologue table in the driver listing. Look for the address of DPT_STORE_END, which generates a 2-byte entry that terminates the DPT. To get the base address of driver code, add the address of DPT_STORE_END + 2 to the address at which the driver was loaded. You can set an XDELTA relocation 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. At the console terminal, issue the following commands:

```
$ CTRL/P
>>>HALT
>>>DEPOSIT/I 14 5
>>>CONTINUE
```

The above procedure issues a software interrupt to the processor at IPL 5. The IPL 5 interrupt service routine handles the interrupt by calling the routine INI\$BRK, which in turn executes the first XDELTA breakpoint. XDELTA then issues the message:

```
1 BRK AT nnnnnnnn
```

DEBUGGING A DEVICE DRIVER

15.6 LOOKING AT THE VECTOR JUMP TABLE

To gain experience in looking around the I/O data base, you may wish to look for the address of the location in the channel request block that contains a JSB instruction to the driver's interrupt service routine. You can do this at a controller initialization breakpoint because one of the inputs is the IDB address:

```
R5/IDB-address Q+C/ADP-address
Q+10/vector-table-address
Q+vector-address-in-hex/address-of-JSB-instruction-in-CRB
Q/ JSB-instruction
```

Finding the driver interrupt service routine address at the expected vector does not guarantee that an interrupt from the device will dispatch to the driver's interrupt service routine. If the device's physical vector is set to some other address, an interrupt from the device may dispatch to some other interrupt service routine, or dispatch to an unassigned vector.

See the SYSGEN device table shown in Chapter 14 for a list of vectors. Consult 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 15.3 to determine the base address of driver code, then set a relocation register by issuing the following command:

```
1 BRK AT nnnnnnnn
driver-base-address,0#X
```

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
```

To set breakpoints in driver code, give the command:

```
X0 + offset#B
```

To display a driver instruction, add the instruction's offset to the base register, for example:

```
X0+1C/instruction .#B
```

The last XDELTA command sets a breakpoint at the displayed location. See Section 15.10 for a detailed discussion of XDELTA commands.

15.8 DESTROYING REGISTER CONTENTS

Since much driver code calls VAX/VMS I/O routines, you must be careful to anticipate the register usage of these routines. Most VAX/VMS common I/O support routines use R0 through R3 freely. A frequent driver bug is to load a value into R3 and expect to find it intact after a call to allocate or load UNIBUS adapter resources.

DEBUGGING A DEVICE DRIVER

Other VAX/VMS I/O routines write into R4. In some cases, the use of R4 is obvious; for example, IOC\$REQSCHNL writes the device's CRB address into R4. In other cases, you might not anticipate the use of R4.

For example, EXE\$IOFORK saves the calling code's R4 in a fork block, and then writes the device's IPL into R4. Since the normal flow of events is that an interrupt service routine restores a driver with a JSB instruction and the driver then calls EXE\$IOFORK which returns to the interrupt service routine, the instructions following the JSB in the interrupt service routine can only assume R5 is still untouched. The coding sequence is as follows:

```
                                ; Restore R3-R4.
JSB    @UCB$L_FPC(R5)          ; Restore the driver process.
.
.
.
```

Between these instructions, the interrupt service routine can make no assumptions about the contents of R0 through R4

```
.
.
.
POPR   #M^<R0,R1,R2,R3,R4,R5> ; Restore interrupt registers.
REI                                         ; Return from the interrupt.
```

15.9 EXAMINING UCB, IRP, AND PSL

In addition to using XDELTA to debug drivers, you also can examine the contents of the unit control block and the associated I/O request packet.

It also is useful to examine the contents of the PSL at the time of a system failure. The PSL, for example, indicates the IPL at the time. When the system fails it prints the PSL and other register contents on the console terminal.

While the system is running, the following command can be used to examine the PSL in XDELTA:

```
RF+4/
```

That is, the PSL location is stored in the longword following the PC.

15.10 XDELTA COMMANDS

Table 15-1 summarizes XDELTA commands. The sections that follow detail the commands.

DEBUGGING A DEVICE DRIVER

Table 15-1
XDELTA Command Summary

Command	Function
/	Open location
<code>(RET)</code>	Close current location
<code>(LF)</code>	Close current location; open next
<code>(TAB)</code>	Open location specified by current value
<code>(ESC)</code>	Display previous location
=	Display value of expression; set Q
+	Add
-	Subtract
space	Add
*	Multiply
@	Shift
%	Divide
,	Field separator
Q	Last quantity displayed
Rn	Register n
Xn	Base register n
Pn	Processor register n
G	Add ^X80000000 to subsequent or preceding value
H	Add ^X7FFE0000 to subsequent or preceding value
.	Current location
S	Execute one instruction
;P	Proceed from breakpoint
;B	Set/clear/display breakpoint
;E	Execute command string
;G	Go to location and proceed
;X	Set base register
[B	Set byte mode
[W	Set word mode
[L	Set longword mode
"	Set ASCII mode
'string'	Deposit string at current dot, autoincrementing dot. A single quote terminates the string; any <code>(RET)</code> s and <code>(LF)</code> s typed will be stored.

15.10.1 Values and Expressions

All numeric values are interpreted in hexadecimal radix. Expressions are strings of alternating values and binary operators, where the first and last items in the string are always values, as in the following example:

G4A32 + 24 - .

Trailing operators are ignored.

DEBUGGING A DEVICE DRIVER

15.10.2 Special Symbols

XDELTA defines the following special symbols:

.	Current location; set by slash (/) and (TAB) operations
Q	Last quantity displayed
X0→XF	Base registers; used for remembering values
R0→RF	General register names
P0→Pnn	Internal processor registers
RF+4	PSL
G	^X8000000; prefix for system space addresses; for example, G2E is equivalent to ^X8000002E
H	^X7FFE0000; prefix for control region prefix; for example, H2E is equivalent to ^X7FFE002E

15.10.3 Operators

XDELTA recognizes the following operators:

+ or space	add
-	negate, subtract
*	multiply
%	divide
@	shift (arithmetic)

Evaluation of expressions is left to right with no precedence.

15.10.4 Open and Display Command

Syntax

address_expression/old_value [new_value_expression] (RET)

You can 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 followed by return (RET). If you type a carriage return not preceded by 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.

DEBUGGING A DEVICE DRIVER

A slash preceded by a null address expression uses the displayed value (Q) as the address value. This feature is convenient for following address linked chains.

```
address_expression/old_value /old_value /old_value
```

15.10.5 Close and Display Next Location Command

Syntax

(LF)

Press the line feed key (LF). XDELTA closes the current open location, then opens and displays the value in the next location according to the current display mode. Next location is calculated to be the current location counter increased by the current data width (byte, word, or longword).

15.10.6 Display Range Command

Syntax

```
start_addr_expression,end_addr_expression/contents_of_start
```

Type two address expressions separated by a comma and followed by a slash (/) character. XDELTA displays the range of addresses specified; the location counter is increased by the current display width; the contents of each location is displayed in the current data type.

15.10.7 Indirect Command

Syntax

(TAB)

Press the (TAB) key. XDELTA uses the result of the last display operation (Q) as an address and displays the contents of that address according to the current display width and data type.

15.10.8 Display Previous Location Command

Syntax

(ESC)

Press the (ESC) key. XDELTA decreases the location counter by the current display width, and displays the contents of the resulting address according to the current display width and data type.

15.10.9 Show Value Command

Syntax

expression=value_of_expression

Type an expression followed by an equal sign (=). The expression can be composed of a series of values and operators from the set of operators listed in the command summary. XDELTA shows the value of the expression according to the current display data type. The last quantity (Q) is set to the value of the computed expression.

15.10.10 Step Instruction Command

Syntax

S

Type an S. XDELTA causes one instruction to be executed and then displays the address of the next instruction and its contents at the current display width and data type.

15.10.11 Setting Breakpoints

Syntax

address-expression;B (RET)

Specify an address followed by a semicolon (;) the letter B, and return (RET). XDELTA sets a breakpoint at the specified location and assigns it the first available breakpoint number.

Alternate syntax:

address-expression,n;B (RET)

Specify an address, followed by a comma, a single digit between 2 and 8, a semicolon (;), the letter B, and return (RET). XDELTA sets a breakpoint at the specified location and assigns it the specified breakpoint number. Breakpoint 1 is reserved for INI\$BRK.

15.10.12 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 return (RET). XDELTA clears the specified breakpoint. Never clear breakpoint 1.

DEBUGGING A DEVICE DRIVER

15.10.13 Displaying Breakpoint List

Syntax

`;B`

Type a semicolon (;) followed by the letter B. XDELTA shows the current setting of all breakpoints. For each breakpoint, XDELTA displays the following information:

- Breakpoint number
- Address at which the breakpoint is set
- Display address (for complex breakpoints; see Section 15.10.19)
- Command string address (for complex breakpoints)

15.10.14 Setting Base Registers

Syntax

`address-expression,n;X`

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. Base registers E and F are preassigned as described in Section 15.10.20.

XDELTA confirms that the base register is set by displaying the expression deposited in the base register.

15.10.15 Proceeding from Breakpoints

Syntax

`;P (RET)`

Type a semicolon (;) followed by the letter P and return (RET). XDELTA continues executing at the current PC.

15.10.16 Loading PC and Continuing

Syntax

`address_expression;G (RET)`

Type an address, a semicolon, and G and then press return. XDELTA loads the address into PC and continues executing at the new PC.

15.10.17 Display Mode Control

Syntax

```
[B   Byte width
[W   Word width
[L   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 (RET)
```

You can display contents of memory locations in ASCII characters by typing an address expression followed by a double quotation mark (").

```
address_expression" old_value_in_ASCII
```

A line feed (LF) command displays the next location in ASCII.

The display mode remains set to ASCII until the next slash (/) command. At this point, the display mode reverts to hexadecimal. Width remains unchanged.

15.10.18 The Execute String Command

Syntax

```
address_expression;E (RET)
```

Type an address expression followed by a semicolon, the letter E, and (RET). 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 (RET), (LF), 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
```

DEBUGGING A DEVICE DRIVER

You can also preassemble command strings with your experimental driver. Locate the addresses of these strings as you would any other address within your driver.

15.10.19 Setting Complex Breakpoints

Syntax

```
address-expression,n,display_address-expression,command_string_address;B
```

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.

15.10.20 XDELTA Stored Commands

XDELTA contains two predefined command strings whose addresses are contained in base registers XE and XF. You can use these commands during general system debugging as well as driver debugging; they perform the following functions:

- XE Use the value of base register X0 as a page frame number and display the PFN data base for that page.
- XF Set base register X0 to the value (PFN) in R0 and perform the same function as XE

You must initialize the stored commands to set the relocation registers they use (X6-XD). Issue the following commands:

```
XE:XE     (RET)
XF:XF     (RET)
```

Now you can use the stored commands to obtain the following information about a page frame number:

- Specified physical page number (PFN)
- PFN state
- PFN type
- PFN reference count
- PFN backward link/working set list index
- PFN forward link/share count
- Page table entry (PTE) pointer to PFN
- PFN backing store address
- Virtual block number in process swap image

DEBUGGING A DEVICE DRIVER

15.10.21 Stored Base Registers

XDELTA defines two base registers useful in system debugging: X4 and X5. Base register X4 corresponds to the global symbol SCH\$GL_CURPCB. This symbol contains the address of the current process's software process control block (PCB). Base register X5 corresponds to the global symbol SCH\$GL_PCBVEC, which contains the starting address of the list of PCB slots.

15.11 DELTA

DELTA is a debugging tool that can be linked with a user program to examine that program's execution. To link and run DELTA, issue the following commands:

```
$ LINK program-name
$ DEFINE LIB$DEBUG SYS$LIBRARY:DELTA
$ RUN/DEB program-name
```

DELTA accepts all the XDELTA commands, plus two additional commands described in the following sections.

15.11.1 The EXIT Command

Syntax

```
EXIT
```

Typing EXIT causes DELTA to return control to the command interpreter.

15.11.2 Examining and Modifying Locations in Process Space

Syntax

```
process_id:address_expression/old_contents
```

DELTA displays the current contents at the specified address expression within the specified process. The modify flag controls the ability to modify locations opened by this command. To examine the flag, type:

```
!M (RET)
```

Modify access is inhibited by default (M=0).

To open, examine and change a location, type the commands:

```
!M (RET)
process_id:address_expression/old_contents new_contents (RET)
```

15.12 DEBUGGING TECHNIQUES

The following sections discuss errors commonly made during debugging sessions and describe additional debugging techniques.

DEBUGGING A DEVICE DRIVER

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 the VAX-11 processor these references cause a UNIBUS adapter error interrupt. Normally, the system logs the 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  (F
                                saved_R3  (F
                                saved_R4  (F
                                saved_R5  (F
                                saved_PC  (F
                                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 bugs 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.

DEBUGGING A DEVICE DRIVER

To gain control in XDELTA following a fatal bugcheck, stop in SYSBOOT while initializing the system and set the BUGREBOOT parameter to zero. The system will stop in XDELTA, thereby allowing you to examine the device unit control block and other driver data to determine the driver error.

Another, more thorough, way to determine the cause of a system failure is to leave the BUGREBOOT parameter set to 1, allow the system to reboot, and then invoke the System Dump Analyzer (SDA) to examine the condition of the I/O data structures at the time of the fatal bugcheck. The VAX/VMS System Dump Analyzer Reference Manual provides detailed information on fatal bugcheck stack format and how SDA can help debug a device driver.

APPENDIX A
THE I/O DATA BASE

The I/O data base is a collection of control blocks allocated in nonpaged system memory. This data base provides the following information:

- I/O request packets describing in-progress I/O requests
- Device characteristics of each device type
- Number and type of each device unit
- Current activity on each device unit
- External entry points to all device drivers
- Entry points for controller and device unit initialization routines
- Interrupt vector dispatch code
- Addresses of device registers
- UNIBUS adapter map register bit map and data path bit map

Much of this I/O data base is created and used only by VAX/VMS routines. Other parts are the primary source of data for the device drivers. The sections that follow identify all I/O data base control blocks and describe their fields. Field descriptions are in the order in which they appear in the control blocks. Driver code must consider fields flagged with asterisks (*) as read-only fields. Fields marked by "spare" or "unused" are reserved for future use by DIGITAL unless otherwise specified.

The data structures described in this appendix are defined in source modules SYSDEF.MDL and STARDEF.MDL.

A.1 I/O REQUEST PACKET (IRP)

When a user process queues a valid I/O request by issuing a Queue I/O Request or Queue I/O Request and Wait system service, the service (EXE\$QIO) creates an I/O request packet. This packet contains a description of the request and receives the status of the I/O processing as it proceeds.

The fields of an I/O request packet are illustrated in Figure A-1 and detailed in Table A-1.

THE I/O DATA BASE

IRP\$L_IOQFL			0
IRP\$L_IOQBL			4
IRP\$B_RMOD*	IRP\$B_TYPE*	IRP\$W_SIZE*	8
IRP\$L_PID*			C
IRP\$L_AST*			10
IRP\$L_ASTPRM			14
IRP\$L_WIND			18
IRP\$L_UCB*			1C
IRP\$B_PRI*	IRP\$B_EFN*	IRP\$W_FUNC	20
IRP\$L_IOSB			
IRP\$W_STS		IRP\$W_CHAN*	
IRP\$L_SVAPTE			
IRP\$W_BCNT		IRP\$W_BOFF	
IRP\$L_IOST1 or IRP\$L_MEDIA			
IRP\$L_IOST2 or IRP\$L_MEDIA+4 or IRP\$B_CARCON			
IRP\$W_OBCNT		IRP\$W_ABCNT	
IRP\$L_SEGVBN			
IRP\$L_DIAGBUF			
IRP\$L_SEQNUM			
IRP\$L_EXTEND			
IRP\$L_ARB			
SPARE			
SPARE			

Figure A-1 I/O Request Packet

THE I/O DATA BASE

Table A-1
Contents of an I/O Request Packet

Field Name	Contents
IRP\$ <u>L</u> _IOQFL	I/O queue forward link. EXE\$INSERTIRP reads and writes this field when the routine inserts I/O packets into an I/O request packet wait queue. IOC\$REQCOM reads and writes this field when the routine dequeues I/O packets from an I/O request packet wait queue in order to send the packet to a device driver.
IRP\$ <u>L</u> _IOQBL	I/O queue backward link. EXE\$INSERTIRP and IOC\$REQCOM read and write these fields.
IRP\$ <u>W</u> _SIZE*	Size of the I/O request packet. EXE\$QIO writes the symbolic constant, IRP\$C_LENGTH, into this field when the routine allocates and fills an I/O packet.
IRP\$ <u>B</u> _TYPE*	Type of control block. EXE\$QIO writes the symbolic constant DYN\$C_IRP into this field when the routine allocates and fills an I/O packet.
IRP\$ <u>B</u> _RMOD*	Access mode of the process at the time of the I/O request. EXE\$QIO obtains the processor access mode from the PSL and writes the value into this field.
IRP\$ <u>L</u> _PID*	Process identification of the process that issued the I/O request. EXE\$QIO obtains the process identification from the process control block and writes the value into this field.
IRP\$ <u>L</u> _AST*	Address of the AST routine specified by the user in the I/O request. If the process specifies an AST routine address in the QIO call, EXE\$QIO writes the address in this field.
IRP\$ <u>L</u> _ASTPRM	<p>During I/O postprocessing, the kernel mode AST routine queues a user mode AST to the requesting process if this field contains the address of an AST routine.</p> <p>Address of a parameter to be sent as an argument to the AST routine specified by the user in the I/O request. If the process specifies an AST routine and a parameter to that AST routine in the QIO call, EXE\$QIO writes the parameter in this field.</p> <p>During I/O postprocessing, the kernel mode AST routine queues a user mode AST if the IRP\$<u>L</u>_AST field contains an address, and passes the value in IRP\$<u>L</u>_ASTPRM to the AST routine as an argument.</p>

(continued on next page)

THE I/O DATA BASE

Table A-1 (Cont.)
Contents of an I/O Request Packet

Field Name	Contents
IRP\$L_WIND	<p>Address of a window block describing the file being accessed in an I/O request. EXE\$QIO writes this field if the I/O request refers to a file-structured device. The ACP reads this field.</p> <p>When a process gains access to a file on a file-structured device or creates a logical link between a file and a process I/O channel, the device ACP creates a window control block (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.</p>
IRP\$L_UCB*	<p>Address of the unit control block for the device assigned to the process I/O channel. EXE\$QIO copies this value from the channel control block.</p>
IRP\$W_FUNC	<p>I/O function code that identifies the function to be performed for the I/O request. The I/O request call specifies an I/O function code; EXE\$QIO and driver FDT routines map the code value to its most basic level (virtual → logical → physical) and copy the reduced value into this field.</p> <p>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.</p>
IRP\$B_EFN*	<p>Event flag number and group specified in the I/O request. If the I/O request call does not specify an event flag number, EXE\$QIO uses event flag 0 by default. EXE\$QIO writes this field. The I/O postprocessing routine calls SCH\$POSTEF to set this event flag when the I/O operation is complete.</p>
IRP\$B_PRI*	<p>Base priority of the process when the I/O request was issued. EXE\$QIO obtains a value for this field from the process control block. EXE\$INSERTIRP reads this field to insert an I/O request packet into a priority-ordered I/O request packet wait queue.</p>

(continued on next page)

THE I/O DATA BASE

Table A-1 (Cont.)
Contents of an I/O Request Packet

Field Name	Contents																										
IRP\$L_IOSB	<p>Virtual address of the process I/O status block that receives the final status of the I/O request at I/O completion. EXE\$QIO writes a value into this field if the I/O request call specifies an IOSB address. The I/O postprocessing kernel mode AST routine writes two longwords of I/O status into the IOSB block after the I/O operation is complete.</p> <p>When an FDT routine aborts an I/O request by calling EXE\$ABORTIO, EXE\$ABORTIO zeroes the IRP\$L_IOSB field so that I/O postprocessing does not write status into the block.</p>																										
IRP\$W_CHAN*	<p>Index number of the process I/O channel for the request. EXE\$QIO writes this field.</p>																										
IRP\$W_STS	<p>Status of the I/O request. EXE\$QIO initializes this field to 0. EXE\$QIO, FDT routines, and driver fork processes modify this field according to the current status of the I/O request. I/O postprocessing reads this field to determine what sort of postprocessing is necessary (for example, deallocate system buffers and adjust quota usage).</p> <p>Bits in the IRP\$W_STS field describe the type of I/O function, as follows:</p> <table data-bbox="641 1161 1349 1631"> <tr> <td>IRP\$V_BUFIO</td> <td>Buffered I/O function</td> </tr> <tr> <td>IRP\$V_FUNC</td> <td>Read function</td> </tr> <tr> <td>IRP\$V_PAGIO</td> <td>Paging I/O function</td> </tr> <tr> <td>IRP\$V_COMPLX</td> <td>Complex buffered I/O function</td> </tr> <tr> <td>IRP\$V_VIRTUAL</td> <td>Virtual I/O function</td> </tr> <tr> <td>IRP\$V_CHAINED</td> <td>Chained buffered I/O function</td> </tr> <tr> <td>IRP\$V_SWAPIO</td> <td>Swapping I/O function</td> </tr> <tr> <td>IRP\$V_DIAGBUF</td> <td>Diagnostic buffer is present</td> </tr> <tr> <td>IRP\$V_PHYSIO</td> <td>Physical I/O function</td> </tr> <tr> <td>IRP\$V_TERMIO</td> <td>Terminal I/O (for priority increment calculation)</td> </tr> <tr> <td>IRP\$V_MBXIO</td> <td>Mailbox I/O function</td> </tr> <tr> <td>IRP\$V_EXTEND</td> <td>An extended IRP is linked to this IRP</td> </tr> <tr> <td>IRP\$V_FILACP</td> <td>File ACP I/O</td> </tr> </table>	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_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																										

(continued on next page)

THE I/O DATA BASE

Table A-1 (Cont.)
Contents of an I/O Request Packet

Field Name	Contents
IRP\$ <u>L</u> _SVAPTE	<p>For a direct I/O operation, specifies the virtual address of the first page table entry (PTE) of the I/O transfer buffer. FDT routines that lock pages in memory for a direct I/O transfer write the PTE address in this field.</p> <p>For a buffered I/O operation, specifies the address of the buffer in system address space. FDT routines that allocate system buffers for a buffered I/O transfer write this field.</p> <p>IOCSINITIATE copies the field into the device unit control block field UCB\$<u>L</u>_SVAPTE before transferring control to a device driver start I/O routine.</p>
IRP\$ <u>W</u> _BOFF	<p>Byte offset in first page of a direct I/O transfer. FDT routines calculate this offset and write the field.</p> <p>For buffered I/O operations, FDT routines must write the number of bytes to be charged to the process in this field because these bytes are being used for a system buffer.</p> <p>IOCSINITIATE copies the field into the device unit control block field UCB\$<u>W</u>_BOFF before calling a device driver start I/O routine.</p> <p>I/O postprocessing uses IRP\$<u>W</u>_BOFF in conjunction with IRP\$<u>W</u>_BCNT and IRP\$<u>L</u>_SVAPTE to unlock pages locked for direct I/O. For buffered I/O, I/O postprocessing adds the value of IRP\$<u>W</u>_BOFF to the process byte count quota.</p>
IRP\$ <u>W</u> _BCNT	<p>Byte count of I/O transfer. FDT routines calculate the count value and write the field. IOCSINITIATE copies the field into the device unit control block field UCB\$<u>W</u>_BCNT before calling a device driver start I/O routine.</p> <p>For a buffered I/O read function, I/O postprocessing uses IRP\$<u>W</u>_BCNT to determine how many bytes of data to write to the user's buffer.</p>
IRP\$ <u>L</u> _IOST1 (also called IRP\$ <u>L</u> _MEDIA)	<p>First I/O status longword. IOCSREQCOM and EXE\$FINISHIO(C) write the contents of R0 into this field. The I/O postprocessing routine copies the contents of this field into the user I/O status block.</p> <p>EXE\$ZEROPARM copies a 0 and EXE\$ONEPARM copies P1 into this field. This field is a good place to put a Queue I/O Request argument (P1 through P6) or a computed value.</p>

(continued on next page)

THE I/O DATA BASE

Table A-1 (Cont.)
Contents of an I/O Request Packet

Field Name	Contents
IRP\$L_IOST2 (also called IRP\$L_MEDIA+4 or IRP\$B_CARCON)	Second I/O status longword. IOC\$REQCOM and EXE\$FINISHIO(C) write the contents of R1 into this field. The I/O postprocessing routine copies the contents of this field into the user I/O status block. IRP\$B_CARCON contains carriage control instructions to the driver. EXE\$READ and EXE\$WRITE copy the contents of P4 of the user's I/O request into this field.
IRP\$W_ABCNT	Accumulated bytes transferred in a virtual I/O transfer. Read and written by IOC\$IOPPOST after a partial virtual transfer.
IRP\$W_OBCNT	Original transfer byte count in a virtual I/O transfer. Read by IOC\$IOPPOST to determine whether a virtual transfer is complete, or whether another I/O request is necessary to transfer the remaining bytes.
IRP\$L_SEGVBN	Virtual block number of the current segment of a virtual I/O transfer. Written by IOC\$IOPPOST after a partial virtual transfer.
IRP\$L_DIAGBUF*	Address of a diagnostic buffer in system address space. If the I/O request call specifies this address, and if a diagnostic buffer length is specified in the driver dispatch table, and if the process has diagnostic privilege, EXE\$OIO copies the buffer address into this field. EXE\$OIO allocates a diagnostic buffer in system address space to be filled by IOC\$DIAGBUFILL during I/O processing. During I/O postprocessing, the kernel mode AST routine copies diagnostic data from the system buffer into the process diagnostic buffer.
IRP\$L_SEQNUM*	I/O transaction sequence number. If an error is logged for the request, this field contains the universal error log sequence number.
IRP\$L_EXTEND	Address of the I/O request packet extension linked to this packet. FDT routines write an extension address to this field when a device requires more context than the I/O request packet can accommodate. This field is read by IOC\$POST. IRP\$V_EXTEND in IRP\$W_STS is set if this extension address is used.

(continued on next page)

THE I/O DATA BASE

Table A-1 (Cont.)
Contents of an I/O Request Packet

Field Name	Contents						
IRP\$L_ARB	Address of the access rights block. This block is located in the process control block and contains the process privilege mask and UIC, which are set up as follows: <table border="0" style="margin-left: 40px;"> <tr> <td>ARB\$Q_PRIV</td> <td>Quadword containing process privilege mask</td> </tr> <tr> <td>SPARE\$L</td> <td>Unused longword</td> </tr> <tr> <td>ARB\$L_UIC</td> <td>Longword containing process UIC</td> </tr> </table>	ARB\$Q_PRIV	Quadword containing process privilege mask	SPARE\$L	Unused longword	ARB\$L_UIC	Longword containing process UIC
ARB\$Q_PRIV	Quadword containing process privilege mask						
SPARE\$L	Unused longword						
ARB\$L_UIC	Longword containing process UIC						

A.2 DEVICE DATA BLOCK (DDB)

The device data block is a variable-length block that identifies the generic device/controller name and driver name for a set of devices attached to a single controller. The driver loading procedure creates a device data block for each controller during autoconfiguration at system startup and dynamically creates additional device data blocks for new controllers as they are added to the system using SYSGEN CONNECT commands. The procedure initializes all fields in the device data block. VAX/VMS routines and device drivers refer to the device data block.

Fields of the device data block are illustrated in Figure A-2 and described in Table A-2.

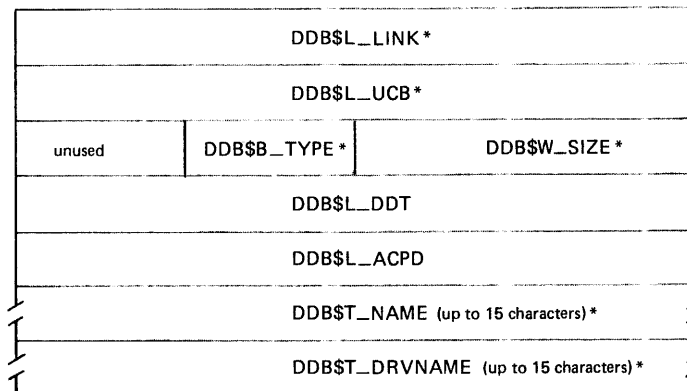


Figure A-2 Device Data Block

THE I/O DATA BASE

Table A-2
Contents of Device Data Block

Field Name	Contents
DDB\$L_LINK*	Address of the next DDB. A zero indicates that this is the last DDB in the DDB chain.
DDB\$L_UCB*	Address of the unit control block for the first unit attached to the controller.
DDB\$W_SIZE*	Size of the DDB.
DDB\$B_TYPE*	Type of control block. The driver loading procedure writes the constant DYN\$C DDB into this field when the procedure creates the DDB.
DDB\$L_DDT	Address of the driver dispatch table. VAX/VMS can transfer control to a device driver only through addresses listed in the DDT, the CRB, and the UCB fork block. The driver prologue table of every device driver must specify a value for this field.
DDB\$L_ACPD	Name of the default ACP for the controller. If the devices on the controller are file-structured devices, this field contains the first four letters of the name of an ACP that controls access to the devices. The driver prologue table specifies a value for this field if it is applicable.
DDB\$T_NAME*	Generic name of the devices attached to the controller. The first byte of this field is the number of characters in the generic name. The remainder of the field consists of a string of up to 15 characters in length that, suffixed by a device unit number, identifies devices on the controller.
DDB\$T_DRVNAME*	Name of the device driver for the controller. The first byte of this field is the number of characters in the driver name. The remainder of the field contains a string of up to 15 characters in length taken from the driver prologue table in the driver.

THE I/O DATA BASE

A.3 UNIT CONTROL BLOCK (UCB)

The unit control block is a variable-length block that describes a single device unit. Each device unit on the system has its own unit control block. The block describes or provides pointers to the device type, controller, driver, device status, and current I/O activity.

During autoconfiguration, the driver loading procedure creates one unit control block for each DIGITAL-supported device unit in the system configuration. A privileged system user can request the driver loading procedure to create unit control blocks for additional devices with the CONNECT command to SYSGEN as described in Chapter 14. The procedure creates unit control blocks of the length specified in the driver prologue table of the device's driver. The driver uses UCB storage located beyond the standard UCB fields for device-specific data and temporary driver storage.

The driver loading procedure initializes some static unit control block fields when it creates the block. VAX/VMS and device drivers can read and modify all nonstatic fields of the unit control block.

The fields of the unit control block that are present for all devices are illustrated in Figure A-3 and described in Table A-3, on pages A-10 and A-11.

THE I/O DATA BASE

UCB\$L_FQFL*		
UCB\$L_FOBL*		
UCB\$B_FIPL*	UCB\$B_TYPE*	UCB\$W_SIZE*
UCB\$L_FPC		
UCB\$L_FR3		
UCB\$L_FR4		
UCB\$W_VPROT*		UCB\$W_BUFQUO
UCB\$L_OWNUIC*		
UCB\$L_CRB*		
UCB\$L_DDB*		
UCB\$L_PID*		
UCB\$L_LINK*		
UCB\$L_VCB*		
UCB\$L_DEVCHAR		
UCB\$W_DEVBUFSIZ	UCB\$B_DEVTYPE	UCB\$B_DEVCLASS
UCB\$L_DEVDEPEND		
UCB\$L_IOQFL		
UCB\$L_IOQBL		
UCB\$W_CHARGE		UCB\$W_UNIT*
UCB\$L_IRP		
UCB\$B_AMOD*	UCB\$B_DIPL*	UCB\$W_REFC*
UCB\$L_AMB*		
UCB\$W_DEVSTS		UCB\$W_STS
UCB\$L_DUETIM*		
UCB\$L_OPCNT*		
UCB\$L_SVPN*		
UCB\$L_SVAPTE*		
UCB\$W_BCNT		UCB\$W_BOFF
UCB\$W_ERRCNT	UCB\$B_ERTMAX	UCB\$B_ERTCNT

0
4
8
C

Figure A-3 Unit Control Block

THE I/O DATA BASE

Table A-3
Contents of Unit Control Block

Field Name	Contents
UCB\$ <u>L</u> _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\$ <u>L</u> _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\$ <u>W</u> _SIZE*	Size of the UCB. The driver prologue table of every driver must specify a value for this field. The driver loading procedure uses the value to allocate space for a UCB and stores the value in each UCB created. Extra space beyond the standard bytes in a UCB (UCB\$K_LENGTH) is for device-specific data and temporary storage.
UCB\$ <u>B</u> _TYPE*	Type of the control block. The driver loading procedure writes the constant DYN\$C_UCB into this field when the procedure creates the UCB.
UCB\$ <u>B</u> _FIPL*	<p>Fork interrupt priority level (IPL) at which the driver of the device usually executes. The driver prologue table of every driver must specify a value for this field. The driver loading procedure writes the value in the UCB when the procedure creates the UCB.</p> <p>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.</p> <p>All devices that are attached to one UNIBUS adapter and actively compete for shared UNIBUS adapter resources and/or a controller data channel must specify the same value for the fork IPL field.</p>
UCB\$ <u>L</u> _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.

(continued on next page)

THE I/O DATA BASE

Table A-3 (Cont.)
Contents of Unit Control Block

Field Name	Contents
UCB\$L_FPC (Cont.)	<p>VAX/VMS routines that suspend driver processing include EXE\$IOFORK, IOC\$REQxCHANx IOCSREQMAPREG, IOC\$REQDATAP, and IOC\$WFIKPCH. Routines that reactivate suspended drivers include IOC\$RELCHAN, IOC\$RELMAPREG, IOC\$RELDATAP, EXE\$FORKDSPATH, and driver interrupt service routines.</p> <p>When a driver interrupt service routine determines that a device is expecting an interrupt, the routine restores control to the saved PC address in the device's UCB.</p>
UCB\$L_FR3	<p>Value of R3 at the time that a VAX/VMS routine suspends a driver fork process. The value of R3 is restored just before a suspended driver regains control.</p>
UCB\$L_FR4	<p>Value of R4 at the time that an operating system routine suspends a driver fork process. The value of R4 is restored just before a suspended driver regains control.</p>
UCB\$W_BUFQUO*	<p>Buffered I/O quota if this UCB represents a mailbox.</p>
UCB\$W_VPROT*	<p>Description of the volume protection if a volume is mounted on this device. This field is written by the MOUNT command when a volume is mounted. It is read by EXE\$QIO to check logical or physical access to a device and by the device's ACP. It is written by the SET PROTECTION/DEVICE command.</p>
UCB\$L_OWNUIC*	<p>User identification code of volume owner. This field is written by the MOUNT command when a volume is mounted. It is read by EXE\$QIO to check logical or physical access to a device and by the device's ACP. It is also written by the SET PROTECTION/DEVICE command.</p>
UCB\$L_CRB*	<p>Address of the primary channel request block associated with the device. The driver loading procedure writes this field after it creates the associated CRB. Driver fork processes read this field to gain access to device registers. VAX/VMS routines use UCB\$L_CRB to locate interrupt dispatching code and initialization routine addresses.</p>

(continued on next page)

THE I/O DATA BASE

Table A-3 (Cont.)
Contents of Unit Control Block

Field Name	Contents
UCB\$L_DDB*	Address of the device data block associated with the device. The driver loading procedure writes this field when the procedure creates the associated UCB. VAX/VMS routines generally read the DDB field in order to locate device driver entry points, the address of a driver function decision table, or the ACP associated with a given device.
UCB\$L_PID*	Process identification code of the process that has allocated the device. Written by the device's ACP.
UCB\$L_LINK*	Address of the next UCB in the chain of UCBs attached to a single controller and associated with a device data block. The driver loading procedure writes this field when the procedure adds the next UCB. Any VAX/VMS routines that examine the status of all devices on the system read this field. Such routines include EXE\$TIMEOUT, IOC\$SEARCHDEV, and power failure recovery routines.
UCB\$L_VCB*	Address of the volume control block (VCB) that describes the volume mounted on the device. This field is written by the device's ACP and read by EXE\$QIOACPPKT and ACPs.
UCB\$L_DEVCHAR	<p>Device characteristics bits. The driver prologue table of every driver should specify symbolic constant values (defined by the \$DEVDEF macro) for this field. The driver loading procedure writes the field when the procedure creates the UCB. The Queue I/O Request system service reads the field to determine whether a device is spooled, file-structured, shared, has a volume mounted, and so on.</p> <p>The system defines the following device characteristics:</p>
	<ul style="list-style-type: none"> DEV\$V_REC Record-oriented device DEV\$V_CCL Carriage control device DEV\$V_TRM Terminal device DEV\$V_DIR Directory-structured device DEV\$V_SDI Single directory-structured device DEV\$V_SQD Sequential block-oriented device (e.g., magtape) DEV\$V_SPL Device is being spooled DEV\$V_NET Network device DEV\$V_FOD Files-oriented device (e.g., disk and magtape)

(continued on next page)

THE I/O DATA BASE

Table A-3 (Cont.)
Contents of Unit Control Block

Field Name	Contents
UCB\$L_DEVCHAR (Cont.)	<p>DEV\$V_SHR Shareable device (used by more than one program simultaneously)</p> <p>DEV\$V_GEN Generic device</p> <p>DEV\$V_AVL Device is available for use</p> <p>DEV\$V_MNT Device is mounted</p> <p>DEV\$V_MBX Mailbox device</p> <p>DEV\$V_DMT Device is marked for dismount</p> <p>DEV\$V_ELG Error-logging is enabled on device</p> <p>DEV\$V_ALL Device is allocated</p> <p>DEV\$V_FOR Device is mounted foreign (i.e., non-file-structured)</p> <p>DEV\$V_SWL Device is software write-locked</p> <p>DEV\$V_IDV Device is capable of providing input</p> <p>DEV\$V_ODV Device is capable of providing output</p> <p>DEV\$V_RND Device allows random access</p> <p>DEV\$V_RTM Real time device</p> <p>DEV\$V_RCK Read-checking is enabled on device</p> <p>DEV\$V_WCK Write-checking is enabled on device</p>
UCB\$B_DEVCLASS	<p>Device class. The driver prologue table of every driver should specify a symbolic constant (defined by the \$DCDEF macro) for this field. The driver loading procedure writes this field when the UCB is created.</p> <p>Drivers with set mode and device characteristics functions rewrite the value in this field with data supplied in an I/O request.</p> <p>The VAX/VMS system defines the following device classes:</p> <p>DC\$_DISK Disk device</p> <p>DC\$_TAPE Tape device</p> <p>DC\$_SCOM Synchronous communications device</p> <p>DC\$_CARD Card reader device</p> <p>DC\$_TERM Terminal device</p> <p>DC\$_LP Line printer device</p> <p>DC\$_REALTIME Real time device</p> <p>DC\$_MAILBOX Mailbox device</p> <p>Note that the definition of a device as realtime is somewhat subjective; it implies no special treatment by VAX/VMS.</p>

(continued on next page)

THE I/O DATA BASE

Table A-3 (Cont.)
Contents of Unit Control Block

Field Name	Contents
UCB\$B_DEVTYPE	<p>Device type. The driver prologue table of every driver should specify a symbolic constant (defined by the \$DTDEF macro) for this field. The driver loading procedure writes the field when the procedure creates the UCB.</p> <p>Drivers with set mode and device characteristics functions rewrite the value in this field with data supplied in an I/O request.</p>
UCB\$W_DEVBUFSIZ	<p>Default buffer size. The driver prologue table can specify a value for this field if relevant. The driver loading procedure writes the field when the procedure creates the UCB.</p> <p>Drivers with set mode and device characteristics functions rewrite the value in this field with data supplied in an I/O request. This field is used by VAX-11 RMS for record I/O on non-file-oriented devices.</p>
UCB\$L_DEVDEPEND	<p>Device-dependent data. Contains device-descriptive data that only the device driver can interpret. The driver prologue table can specify a value for this field. The driver loading procedure writes this field when the procedure creates the UCB.</p> <p>Drivers with set mode and device characteristics functions rewrite the value in this field with data supplied in an I/O request.</p>
UCB\$L_IOQFL*	<p>I/O queue listhead forward link. The queue contains the addresses of I/O request packets waiting for processing on a device. EXE\$INSERTIRP inserts I/O request packets into the I/O request packet wait queue when a device is busy. IOC\$REQCOM dequeues I/O request packets when the device is idle.</p> <p>The queue is a priority queue that has the highest priority packets at the front of the queue. Priority is determined by the base priority of the requesting process. Packets with the same priority are processed first-in/first-out.</p>

(continued on next page)

THE I/O DATA BASE

Table A-3 (Cont.)
Contents of Unit Control Block

Field Name	Contents
UCB\$L_IOQBL*	I/O queue listhead backward link. EXE\$INSERTIRP and IOC\$REQCOM modify the I/O request packet wait queue.
UCB\$W_UNIT*	Number of the physical device unit. Stored as a binary value. The driver loading procedure writes a value into this field when the UCB is created. Drivers for multiunit controllers read this field during unit initialization to identify a unit to the controller.
UCB\$W_CHARGE*	Mailbox byte count quota charge, if the device is a mailbox.
UCB\$L_IRP	Address of the I/O request packet currently being processed on the device unit by a driver fork process. IOC\$INITIATE writes an I/O request packet address into this field before the routine creates a driver fork process to handle an I/O request. A driver fork process obtains the address of the I/O request packet being processed from this field. The value contained in this field is valid if the UCB\$V_BSY bit in UCB\$W_STS is set.
UCB\$W_REFC*	Reference count of processes that currently have process I/O channels assigned to the device. Incremented by the \$ASSIGN and \$ALLOC system services. Decrementd by the \$DASSGN and \$DALLOC system services.
UCB\$B_DIPL	Device interrupt priority level at which the device requests hardware interrupts. The driver prologue table of every driver must specify a value for this field. The driver loading procedure writes the field when the procedure creates the UCB. Some device drivers raise IPL to this value before reading or writing device registers.
UCB\$B_AMOD*	If the device unit is allocated, the access mode at which the allocation occurred. Written by the \$ALLOC and \$DALLOC system services.
UCB\$L_AMB*	Associated mailbox UCB pointer. This field is used for spooled devices and mailboxes.
UCB\$W_STS	Device unit status. Written by drivers, IOC\$REQCOM, IOC\$CANCELIO, IOC\$INITIATE, IOC\$WFIKPCH, IOC\$WFIRLCH, EXE\$INSIOQ, and EXE\$TIMEOUT. This field is read by drivers, the Queue I/O Request system service routines, IOC\$REQCOM, IOC\$INITIATE, and EXE\$TIMEOUT.

(continued on next page)

THE I/O DATA BASE

Table A-3 (Cont.)
 Contents of Unit Control Block

Field Name	Contents
UCB\$W_STS (Cont.)	<p>This status word includes the following bits:</p> <p>UCB\$V_TIM Timeout enabled</p> <p>UCB\$V_INT Interrupts expected</p> <p>UCB\$V_ERLOGIP Error log in progress</p> <p>UCB\$V_CANCEL Cancel I/O on unit</p> <p>UCB\$V_ONLINE Device is online</p> <p>UCB\$V_POWER Power has failed while unit was busy</p> <p>UCB\$V_TIMEOUT Unit is timed out</p> <p>UCB\$V_INTTYPE Receiver interrupt</p> <p>UCB\$V_BSY Unit is busy</p> <p>UCB\$V_MOUNTING Device is being mounted</p> <p>UCB\$V_DEADMO Deallocate device at dismount</p> <p>UCB\$V_VALID Software believes volume is valid</p> <p>UCB\$V_UNLOAD Unload volume at dismount</p> <p>UCB\$V_TEMPLATE Template unit control block 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.</p>
UCB\$W_DEVSTS	Device-dependent status. Read and written by device drivers.
UCB\$L_DUETIM*	<p>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\$WFIKPC and IOC\$WFIRLCH write this value when they suspend a driver to wait for an interrupt or timeout.</p> <p>EXE\$TIMEOUT examines this field in each UCB in the I/O data base once per second. If the timeout has occurred and timeouts are enabled for the device, EXE\$TIMEOUT calls a device timeout handler in the device driver.</p>
UCB\$L_OPCNT*	Count of operations completed on the device unit since VAX/VMS was booted. IOC\$REQCOM writes this field every time the routine inserts an I/O request packet in the I/O postprocessing queue.

(continued on next page)

THE I/O DATA BASE

Table A-3 (Cont.)
 Contents of Unit Control Block

Field Name	Contents
UCB\$L_SVPN*	<p>Virtual address of a page table entry permanently allocated to the device by the driver loading procedure. This field is used for ECC error correction by disk drivers.</p> <p>If a driver prologue table specifies DPT\$M_SVP in the flags argument to the DPTAB macro, the driver loading procedure allocates a page of nonpaged system memory to the device. The procedure writes the virtual address of the page table entry into UCB\$L_SVPN when the procedure creates the UCB.</p>
UCB\$L_SVAPTE	<p>For a direct I/O operation, the virtual address of the system page table entry (PTE) for the first page that is to be used in an I/O transfer. For a buffered I/O operation, the address of the system buffer used in the transfer. This field is used only in transfer operations.</p> <p>I/O postprocessing uses this field to deallocate the system buffer for a buffered I/O operation or to unlock pages locked for a direct I/O operation.</p> <p>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.</p>
UCB\$W_BOFF	<p>For direct I/O operations, byte offset in first page of the transfer buffer. For buffered I/O operations, the number of bytes charged to a process for a transfer. IOC\$INITIATE copies this field from the I/O request packet.</p> <p>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.</p>
UCB\$W_BCNT	<p>Count of bytes in I/O transfer. IOC\$INITIATE copies this field from the I/O request packet. Drivers read this field to determine how many bytes to transfer in an I/O operation.</p>

(continued on next page)

THE I/O DATA BASE

Table A-3 (Cont.)
Contents of Unit Control Block

Field Name	Contents
UCB\$B_ERTCNT	Error retry count of current I/O transfer. The driver sets this field to the maximum retry count each time it begins I/O processing. Before each retry, the driver decreases the value in this field. If error-logging is occurring, IOC\$REQCOM copies the value into the error message buffer.
UCB\$B_ERTMAX	Maximum error retry count allowed for a single I/O transfer. The driver prologue table of some drivers specifies a value for this field. The driver loading procedure writes the field when the procedure creates the UCB. If error-logging is occurring, IOC\$REQCOM copies the value into the error message buffer.
UCB\$W_ERRCNT	Number of errors that have occurred on the device since the system was bootstrapped. The driver loading procedure initializes the field to 0 when the procedure creates the UCB. ERL\$DEVICERR and ERL\$DEVICTMO increment the value in the field and copy the value into an error message buffer. The DCL command SHOW DEVICE displays in its error count column the value contained in this field.

Unit control blocks are variable length depending on the type of device and whether the driver performs error-logging for the device. The error log UCB extension, if present, appears directly after the UCB\$W_ERRCNT field of the standard UCB.

The fields in the UCB error log extension are illustrated in Figure A-4 and described in Table A-4.

UCB\$B_CEX	UCB\$B_FEX	UCB\$B_SPR	UCB\$B_SLAVE
UCB\$L_EMB*			
UCB\$W_FUNC		unused	
UCB\$L_DPC			

Figure A-4 UCB Error Log Extension

THE I/O DATA BASE

Table A-4
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. MBA drivers use this field to store a fixed offset to the MBA registers for the unit.
UCB\$B_FEX	Device-specific field. This field is reserved for driver use.
UCB\$B_CEX	Device-specific field. This field is reserved for driver use.
UCB\$L_EMB*	Address of the error message buffer. If error logging is enabled and a device/controller error or timeout occurs, the driver calls ERL\$DEVICERR or ERL\$DEVICTMO to allocate an error message buffer and copy the buffer address into this field. IOC\$REQCOM writes final device status, error counters, and I/O request status into the buffer specified by this field.
UCB\$W_FUNC	I/O function modifiers. This field is read and written by drivers that log errors.
UCB\$L_DPC	Device-specific field. This field is reserved for driver use.

Another extension of the unit control block is the disk extension block. This UCB extension is present for all disk devices. It follows the error log extension. A driver that supports a disk must allow space in the UCB for both the error log and disk extensions.

Disk drivers use three bits in UCB\$W_DEVSTS as follows:

UCB\$V_ECC	ECC correction made
UCB\$V_DIAGBUF	Diagnostic buffer specified
UCB\$V_NOCNVRT	No logical block number to media address conversion

The fields are illustrated in Figure A-5 and described in Table A-5.

UCB\$L_MAXBLOCK		
UCB\$W_OFFSET	UCB\$W_DIRSEQ	
UCB\$L_MEDIA		
UCB\$W_EC2	UCB\$W_EC1	
UCB\$W_BCR	UCB\$B_OFFRTC	UCB\$B_OFFNDX

Figure A-5 UCB Disk Extension

THE I/O DATA BASE

Table A-5
UCB Disk Extension

Field Name	Contents
UCB\$\$_MAXBLOCK	<p>Maximum number of logical blocks on a random access device. This field is written by a disk driver during unit initialization and power recovery.</p> <p>This field is a parameter of the disk device unit and must be reset to a standard value whenever the disk is started.</p>
UCB\$\$_DIRSEQ	Directory sequence number.
UCB\$\$_OFFSET	Current offset register contents.
UCB\$\$_MEDIA	Media address.
UCB\$\$_EC1	<p>ECC position register. This field records the starting bit number of an error burst. Disk driver register dump routines copy the contents of this field into an error-logging or diagnostic buffer.</p> <p>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.</p>
UCB\$\$_EC2	<p>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.</p> <p>The VAX/VMS ECC correction routine IOC\$APPLYECC reads the contents of this field to correct disk data.</p>
UCB\$\$_OFFNDX	Current offset table index. When a disk driver transfer ends in an error, the disk driver can retry the error a number of times with different offsets of the disk head from the centerline. This field is an index into a driver table of offset positions.
UCB\$\$_OFFRTC	Current offset retry count. This field records the number of times to try a particular offset setting in a disk transfer retry.
UCB\$\$_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.

A.4 CHANNEL REQUEST BLOCK (CRB)

The activity of each controller in a configuration is described in a channel request block. This control block contains pointers to the wait queue of drivers ready to gain access to a device through the controller. It also stores the entry points to the driver's interrupt service routines and device/controller initialization routines.

THE I/O DATA BASE

The fields of the channel request block are illustrated in Figure A-6 and described in Table A-6.

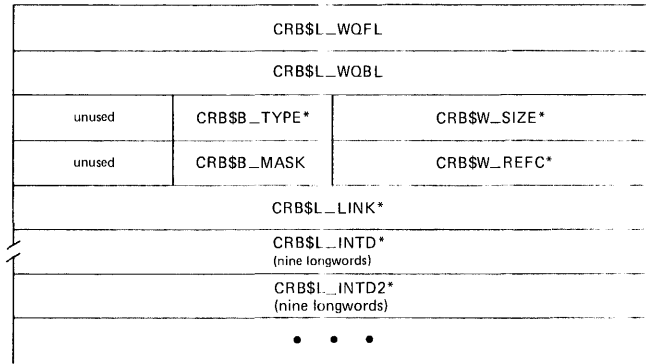


Figure A-6 Channel Request Block

Table A-6
Contents of Channel Request Block

Field Name	Contents
CRB\$L_WQFL*	<p>Controller data channel wait queue forward link. IOC\$REQxCHANx and IOC\$RELxCHAN insert and remove driver fork block addresses in this field.</p> <p>A channel wait queue contains addresses of driver fork blocks that record the context of suspended drivers waiting to gain control of a controller data channel. If a channel is busy when a driver requests access to the channel, IOC\$REQxCHANx suspends the driver by saving the driver's context in the device's UCB fork block and inserting the fork block address in the channel wait queue.</p> <p>When a driver releases a channel because an I/O operation no longer needs the channel, IOC\$RELxCHAN dequeues a driver fork block, allocates the channel to the driver, and reactivates the suspended driver fork process. If no drivers are awaiting the channel, IOC\$RELxCHAN clears the channel busy bit.</p>
CRB\$L_WQBL*	Controller channel wait queue backward link. IOC\$REQxCHANx and IOC\$RELxCHAN read and write this field.
CRB\$W_SIZE*	Size of the CRB. The driver loading procedure writes this field when the procedure creates the CRB.
CRB\$B_TYPE*	Type of control block. The driver loading procedure writes the symbolic constant DYN\$C_CRB into this field when the procedure creates the CRB.

(continued on next page)

THE I/O DATA BASE

Table A-6 (Cont.)
Contents of Channel Request Block

Field Name	Contents
CRB\$W_REFC*	Unit control block reference count. The driver loading procedure increases the value in this field each time the procedure creates a UCB for a device attached to the controller.
CRB\$B_MASK*	Mask that describes the status of the controller. At present, only one bit, CRB\$V_BSY, is defined in this field. IOC\$REQxCHANx reads the busy bit to determine whether the controller is free and sets this bit when it allocates the controller data channel to a driver. IOC\$RELxCHAN clears the busy bit if no driver is waiting to acquire the channel.
CRB\$L_LINK*	Address of secondary CRB (for MASSBUS devices only). This field is written by the driver loading procedure and read by IOC\$REQSCHANx and IOC\$RELSCHAN.
CRB\$L_INTD*	<p>Interrupt transfer vector. The driver prologue table in every driver for an interrupting device specifies the address of a driver interrupt service routine. The driver loading procedure writes two instructions in this field:</p> <pre> PUSHR #^M<R0,R1,R2,R3,R4,R5> JSB @#^driver_isr_address </pre> <p>When a UNIBUS device generates an interrupt on the VAX-11 processor, a VAX/VMS UNIBUS adapter interrupt service routine transfers control to the JSB instruction in a CRB. The UNIBUS adapter service routine determines the appropriate CRB address from the vector address of the device interrupt.</p> <p>The CRB\$L_INTD field is nine longwords long. Figure A-7 and Table A-7 describe the contents of the rest of block.</p>
CRB\$L_INTD2*	<p>Second interrupt transfer vector for devices with multiple interrupt vectors. If the driver prologue table in a device driver specifies the address of a second driver interrupt service routine, the driver loading procedure creates a CRB long enough to contain two INTDx fields of nine longwords each.</p> <p>The first two longwords of the CRB\$L_INTD2 field contain a PUSHHR and JSB instruction to the second driver interrupt service routine.</p>

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 CONNECT command.

THE I/O DATA BASE

The interrupt transfer vector blocks contained in the CRB store executable code, driver entry points, and UNIBUS adapter information. The fields of the CRB\$L_INTD block are illustrated in Figure A-7 and described in Table A-7.

VEC\$Q_DISPATCH*		
VEC\$L_IDB*		
VEC\$L_INITIAL*		
VEC\$B_DATAPATH	VEC\$B_NUMREG	VEC\$W_MAPREG
VEC\$L_ADP*		
VEC\$L_UNITINIT*		
spare longword		
spare longword		

Figure A-7 Contents of CRB\$L_INTD

Table A-7
Fields of CRB\$L_INTD

Field Name	Contents
VEC\$Q_DISPATCH*	Contains the two interrupt dispatching instructions described above in the CRB\$ <u>L</u> _INTD field. This field is written by the driver loading procedure.
VEC\$L_IDB*	Address of the interrupt data block for the controller. The driver loading procedure creates an IDB for each CRB and loads the address of the IDB in this field. Device drivers use the IDB address to obtain the virtual addresses of device registers. When a driver interrupt service routine gains control, the top of stack contains a pointer to the IDB.
VEC\$L_INITIAL*	Address of the controller initialization routine. If a device controller requires initialization at driver loading time and during recovery from a power failure, the driver specifies a value for this field in the driver prologue table. The driver loading procedure calls this routine each time the procedure loads the driver. The VAX/VMS powerfail recovery procedure also calls this routine to initialize a controller after a power failure.

(continued on next page)

THE I/O DATA BASE

Table A-7 (Cont.)
Fields of CRBSL_INTD

Field Name	Contents
VEC\$W_MAPREG	<p>Number of the first UNIBUS adapter map register allocated to the driver that owns the controller data channel. IOC\$REQMAPREG writes this field when the routine allocates a set of map registers to a driver fork process for a DMA transfer. IOC\$RELMAPREG reads the field to deallocate a set of map registers. If the high bit (VEC\$V_MAPLOCK) of this field is set, the map register set is permanently allocated.</p> <p>Device drivers read this field to calculate the starting address of a UNIBUS transfer.</p>
VEC\$B_NUMREG	<p>Number of UNIBUS adapter map registers allocated to a driver. IOC\$REQMAPREG writes this field when the routine allocates a set of map registers. IOC\$RELMAPREG reads this field to deallocate a set of map registers.</p>
VEC\$B_DATAPATH	<p>The data path specifier. The bits that make up this field are used as follows:</p> <p style="margin-left: 40px;">0 → 4 The number of the data path used in a DMA transfer. The routine IOC\$REQDATAP sets this field when a buffered data path is allocated and clears the field when the data path is released.</p> <p style="margin-left: 40px;">The routine IOC\$LOADUBAMAP copies the contents of this field into the UNIBUS adapter map registers. These bits also serve as implicit input to the IOC\$PURGDATAP routine.</p> <p style="margin-left: 40px;">VEC\$V_LWAE Longword access enable (LWAE) bit. Drivers set this bit when they wish to limit the data path to longword-aligned random access mode. The routine IOC\$LOADUBAMAP copies the value in this field to the UNIBUS adapter map registers.</p> <p style="margin-left: 40px;">6 Reserved to DIGITAL.</p> <p style="margin-left: 40px;">VEC\$V_PATHLOCK Buffered data path allocation indicator. Drivers set this bit to specify that the buffered data path is permanently allocated.</p>

(continued on next page)

THE I/O DATA BASE

Table A-7 (Cont.)
Fields of CRB\$\$_INTD

Field Name	Contents
VEC\$_ADP*	<p>Address of the UNIBUS adapter control block (ADP). The CONNECT command to SYSGEN 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\$_ADP field.</p> <p>IOC\$REQMAPREG and IOC\$RELMAPREG read and write fields in the ADP to allocate and deallocate UNIBUS adapter map registers.</p>
VEC\$_UNITINIT*	<p>Address of the device unit initialization routine. If a device unit requires initialization at driver loading time and during recovery from a power failure, the driver specifies a value for this field in the driver prologue table.</p> <p>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.</p> <p>MASSBUS drivers that support mixed device types must not use this field. Instead, they should specify unit initialization in the (DDT\$_UNITINIT) unit initialization field of the driver dispatch table. Other drivers may use either field.</p>

A.5 INTERRUPT DATA BLOCK (IDB)

The interrupt data block records controller characteristics. The driver loading procedure creates and initializes this block when the procedure creates a channel request block. The interrupt data block points to the physical controller by storing the virtual address of the control/status register. This register is the indirect pointer to all device unit registers.

The fields of the interrupt data block are illustrated in Figure A-8 and detailed in Table A-8.

THE I/O DATA BASE

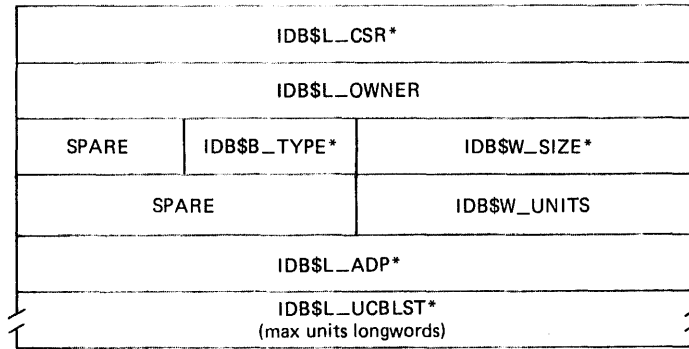


Figure A-8 Interrupt Data Block

Table A-8
Contents of Interrupt Data Block

Field Name	Contents
IDB\$L_CSR*	<p>Address of the control/status register (CSR). The CONNECT command to SYSGEN must specify the address of a device's control/status register. The driver loading procedure writes the system virtual equivalent of this address into the IDB\$L_CSR field.</p> <p>Device drivers set and clear bits in device registers by referencing all device registers at fixed offsets from the CSR address.</p>
IDB\$L_OWNER	<p>Address of the unit control block of the device that owns the controller data channel. IOC\$REQxCHANx writes a UCB address into this field when the routine allocates a controller data channel to a driver. IOC\$RELxCHAN confirms that the proper driver fork process is releasing a channel by comparing the driver's UCB with the UCB stored in the IDB\$L_OWNER field. If the UCB addresses are the same, IOC\$RELxCHAN allocates the channel to a waiting driver by writing a new UCB address into the field. If no driver fork processes are waiting for the channel, IOC\$RELxCHAN clears the field.</p> <p>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.</p>
IDB\$W_SIZE*	<p>Size of the IDB. The driver loading procedure writes the constant IDB\$K_LENGTH into this field when the procedure creates the IDB.</p>
IDB\$B_TYPE*	<p>Type of control block. The driver loading procedure writes the symbolic constant DYN\$C_IDB into this field when the procedure creates the IDB.</p>

(continued on next page)

THE I/O DATA BASE

Table A-8 (Cont.)
Contents of Interrupt Data Block

Field Name	Contents
IDB\$W_UNITS*	Maximum number of units connected to the controller. The maximum number of units is specified in the driver prologue table and may be overridden at driver loading time.
IDB\$L_ADP*	Address of the UNIBUS adapter control block (ADP). The CONNECT command to SYSGEN must specify the nexus number of the UNIBUS adapter used by a device. The driver loading procedure writes the address of the ADP for the specified UNIBUS adapter into the IDP\$L_ADP field.
IDB\$L_UCBLST*	List of UCB addresses. The size of this field is the maximum number of units supported by the controller, as defined in the driver prologue table. The maximum specified in the DPT can be overridden at driver load time. The driver loading procedure writes a UCB address into this field every time the routine creates a new UCB associated with the controller.

A.6 ADAPTER CONTROL BLOCK (ADP)

Each MASSBUS and UNIBUS adapter configured in the system is represented to VAX/VMS and driver routines by an adapter control block. The adapter control block stores adapter-specific static and dynamic data such as the adapter CSR address and map register wait queues.

The fields of the ADP are illustrated in Figure A-9 and described in Table A-9.

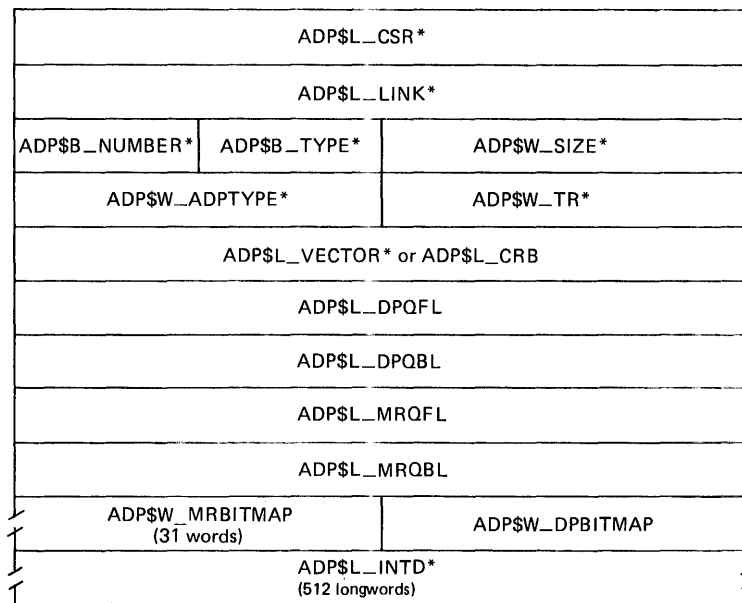


Figure A-9 Adapter Control Block

THE I/O DATA BASE

Table A-9
Contents of Adapter Control Block

Field Name	Contents
ADP\$L_CSR*	<p>Virtual address of the adapter configuration register. The CPU initialization sets this field.</p> <p>The configuration register marks the base of adapter register space, an area that contains data path registers, map registers, or any other registers appropriate to the implementation of the adapter.</p>
ADP\$L_LINK*	<p>Address of next ADP. The CPU initialization routine writes this field. A value of 0 indicates that this is the last ADP.</p>
ADP\$W_SIZE*	<p>Size of the ADP control block. The CPU initialization routine writes this field when the routine creates the ADP. For the UNIBUS adapter, this includes the UNIBUS interrupt service code and device vector table.</p>
ADP\$B_TYPE*	<p>Type of control block. The CPU initialization routine writes the symbolic constant DYN\$C ADP into this field when the routine creates the ADP.</p>
ADP\$B_NUMBER*	<p>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.</p>
ADP\$W_TR*	<p>Nexus number of the adapter. The CPU initialization routine writes this field when the routine creates the ADP. The driver loading procedure compares the nexus number specified in a CONNECT command with this field of each ADP in the system to determine to which adapter a device is attached.</p>
ADP\$W_ADPTYPE*	<p>Type of adapter. The CPU initialization routine writes the symbolic constant AT\$_UBA into this field when the routine creates an ADP for a UNIBUS adapter. AT\$_MBA is the type code for a MASSBUS adapter.</p>

(continued on next page)

THE I/O DATA BASE

Table A-9 (Cont.)
Contents of Adapter Control Block

Field Name	Contents
ADP\$L_VECTOR*	<p>Address of vector table. The table is 512 bytes of longword vectors. The CPU initialization routine allocates portions of nonpaged pool to create this table. Each longword in the vector table that corresponds to a vector in use contains the address of the controller's interrupt dispatcher (CRB\$L_INTD) plus 2. When the UNIBUS adapter interrupts on the behalf of its UNIBUS devices, the UNIBUS adapter service routine saves R0 through R5, then determines the vector of the interrupting device, indexes into the vector table, and executes the instruction at CRB\$L_INTD+2.</p> <p>Longwords in this table that correspond to unused vectors contain the address of an unexpected UNIBUS interrupt routine.</p>
ADP\$L_CRB	<p>Address of the MASSBUS adapter's channel request block. The CPU initialization routine sets this address when it creates the CRB and the adapter control block.</p>
ADP\$L_DPQFL*	<p>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.</p> <p>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.</p>
ADP\$L_DPQBL*	<p>Data path wait queue backward link. IOC\$REQDATAP and IOC\$RELDATAP read and write this field.</p>
ADP\$L_MRQFL*	<p>Map register wait queue forward link. IOC\$REQMAPREG and IOC\$RELMAPREG read and write these fields. When a driver fork process requests a set of map registers and the set is not currently available, IOC\$REQMAPREG saves driver fork context in the device's UCB fork block, inserts the fork block address in the map register wait queue, and suspends the driver fork process.</p>

(continued on next page)

THE I/O DATA BASE

Table A-9 (Cont.)
Contents of Adapter Control Block

Field Name	Contents
ADP\$ <u>L</u> _MRQFL* (Cont.)	When another driver calls IOC\$REL <u>MAP</u> REG to release a set of map registers, the routine dequeues a UCB fork block address from the map register wait queue, allocates the requested set of map registers to the driver, and reactivates that driver fork process.
ADP\$ <u>L</u> _MRQBL*	Map register wait queue backward link. IOC\$REQ <u>MAP</u> REG and IOC\$REL <u>MAP</u> REG read and write this field.
ADP\$ <u>W</u> _DPBITMAP*	Data path allocation bit map. IOC\$REQ <u>DATAP</u> and IOC\$REL <u>DATAP</u> read and write this field. The CPU initialization routine sets the bit map to show as available all the buffered data paths supported by the UNIBUS adapter. The VAX-11 UNIBUS adapter supports fifteen buffered data paths.
ADP\$ <u>W</u> _MRBITMAP*	<p>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.</p> <p>Map register allocation bit map. The field is 31 words long. IOC\$REQ<u>MAP</u>REG and IOC\$REL<u>MAP</u>REG read and write this field.</p>
ADP\$ <u>L</u> _INTD*	<p>The state of each of the 496 map registers (whether in use or available) is stored in the map register bit map. One map register corresponds to each bit in the field. If a bit is clear, the related map register is currently allocated to a driver fork process.</p> <p>Interrupt transfer vector. When a device attached to the UNIBUS adapter requests a hardware interrupt, the processor transfers control to the ADP\$<u>L</u>_INTD field of the UNIBUS adapter's control block. The field contains code that dispatches the interrupt to the proper driver interrupt service routine. The interrupt transfer vector is only used for UNIBUS adapters that directly generate interrupts.</p>

A.7 DRIVER DISPATCH TABLE (DDT)

Each device driver contains a driver dispatch table. The table lists entry points in the driver that various VAX/VMS routines call. An example is the entry point for the driver routine that starts an I/O operation on a device.

THE I/O DATA BASE

A device driver creates a driver dispatch table by invoking the VAX/VMS macro DDTAB. The fields in the driver dispatch table are illustrated in Figure A-10 and described in Table A-10.

DDT\$L_START
DDT\$L_UN SOLINT
DDT\$L_FDT
DDT\$L_CANCEL
DDT\$L_REGDUMP
DDT\$W_ERRORBUF
DDT\$W_DIAGBUF
DDT\$L_UNITINIT
DDT\$L_ALTSTART

Figure A-10 Driver Dispatch Table

Table A-10
Contents of Driver Dispatch Table

Field Name	Contents
DDT\$L_START	<p>Entry point to the driver start I/O routine. Every driver must specify this field with the value of the START argument in the DDTAB macro invocation.</p> <p>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.</p>
DDT\$L_UN SOLINT	<p>Entry point to the driver unsolicited interrupt service routine. The driver specifies this field with the value of the UNSOLIC argument in the DDTAB macro invocation.</p> <p>This field contains the address of a routine that analyzes unexpected interrupts from a device. The standard driver interrupt service routine, the address of which is stored in the CRB, determines whether an interrupt was solicited by a driver. If the interrupt is unsolicited, the service routine may call the unsolicited interrupt service routine.</p>
DDT\$L_FDT	<p>Address of the driver's function decision table. Every driver must specify this field with the value of the FUNCTB argument in the DDTAB macro invocation.</p>

(continued on next page)

THE I/O DATA BASE

Table A-10 (Cont.)
Contents of Driver Dispatch Table

Field Name	Contents
DDT\$ <u>L</u> _FDT (Cont.)	EXE\$QIO refers to the FDT to validate I/O function codes, decide which functions are buffered, and call FDT action routines associated with function codes.
DDT\$ <u>L</u> _CANCEL	<p>Entry point to the driver cancel I/O routine. The driver specifies this field with the value of the CANCEL argument in the DDTAB macro invocation.</p> <p>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.</p>
DDT\$ <u>L</u> _REGDUMP	<p>Entry point to the driver register dump routine. The driver specifies this field with the value of the REGDMP argument in the DDTAB macro invocation.</p> <p>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.</p>
DDT\$ <u>W</u> _DIAGBUF	<p>Size of the diagnostic buffer. The driver specifies this field with the value of the DIAGBF argument in the DDTAB macro invocation. The value is the size in bytes of a diagnostic buffer for the device.</p> <p>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.</p>
DDT\$ <u>W</u> _ERRORBUF	<p>Size of the error log buffer. The driver specifies this field as the value of the ERLGBF argument in the DDTAB macro invocation. The value is the size in bytes of an error-logging buffer for the device.</p> <p>If error logging is enabled and an error occurs during an I/O operation, the driver calls ERL\$DEVICERR or ERL\$DEVICTMO to allocate and write error-logging data into the error message buffer. IOC\$INITIATE and IOC\$REQCOM write values into the error message buffer if an error has occurred.</p>

(continued on next page)

THE I/O DATA BASE

Table A-10 (Cont.)
Contents of Driver Dispatch Table

Field Name	Contents
DDT\$\$_UNITINIT	Address of the device unit initialization routine, if one exists. Drivers for MASSBUS devices use this field rather than CRB\$_INTD+VEC\$_UNITINIT. Drivers for UNIBUS devices may use either field.
DDT\$_ALTSTART	Address of the alternate start I/O routine. The VAX/VMS routine EXE\$ALTQUEPKT initiates the alternate start I/O routine at this address.

A.8 DRIVER PROLOGUE TABLE (DPT)

When loading a device driver and its data base into virtual memory, the driver loading procedure finds the basic description of the driver and its device in a driver prologue table. This table provides the length, name, adapter type, and loading and reloading specifications for the driver.

A device driver creates a driver prologue table by invoking the VAX/VMS macros DPTAB and DPT_STORE. The fields of the DPT are illustrated in Figure A-11 and described in Table A-11.

DPT\$_FLINK*		
DPT\$_BLINK*		
DPT\$_REFC*	DPT\$_TYPE	DPT\$_SIZE
DPT\$_UCBSIZE		DPT\$_FLAGS DPT\$_ADPTYPE
DPT\$_REINITTAB		DPT\$_INITTAB
DPT\$_MAXUNITS		DPT\$_UNLOAD
SPARE		DPT\$_VERSION
DPT\$_NAME (up to 15 characters)		SPARE

Figure A-11 Driver Prologue Table

THE I/O DATA BASE

Table A-11
Contents of Driver Prologue Table

Field Name	Contents						
DPT\$L_FLINK*	Forward link to the next DPT. The driver loading procedure writes this field. The procedure links all driver prologue tables in the system in a doubly linked list.						
DPT\$L_BLINK*	Backward link to the previous DPT. The driver loading procedure writes this field.						
DPT\$W_SIZE	Size in bytes of the device driver. The DPTAB macro writes this field by subtracting the address of the beginning of the DPT from the address specified as the END argument in the invocation of the DPTAB macro. The driver loading procedure uses this value to determine the space needed in nonpaged system memory to load the driver.						
DPT\$B_TYPE	Type of control block. The DPTAB macro always writes the symbolic constant, DYN\$C_DPT, into this field.						
DPT\$B_REFC*	Number of device data blocks that refer to this driver. The driver loading procedure increments the value in this field each time the procedure creates another DDB that points to the driver's DDT.						
DPT\$B_ADPTYPE	Type of adapter used by devices driven by this driver. Every driver must specify the string "UBA" or "MBA" as value of the argument ADAPTER in the invocation of the DPTAB macro. The macro writes the value AT\$_UBA or AT\$_MBA in this field.						
DPT\$B_FLAGS	<p>Driver loader flags. The driver can specify any of a set of flags as the value of the argument FLAGS in the invocation of the DPTAB macro. The driver loading procedure modifies the loading and reloading algorithm followed based on the settings of these flags.</p> <p>Flags defined in the flag field include the following:</p> <table data-bbox="649 1512 1383 1726"> <tr> <td data-bbox="649 1512 893 1554">DPT\$M_SUBCNTRL</td> <td data-bbox="893 1512 1383 1554">Device is a subcontroller</td> </tr> <tr> <td data-bbox="649 1554 893 1659">DPT\$M_SVP</td> <td data-bbox="893 1554 1383 1659">Device requires permanent system page; allocated during driver loading</td> </tr> <tr> <td data-bbox="649 1659 893 1726">DPT\$M_NOUNLOAD</td> <td data-bbox="893 1659 1383 1726">Driver cannot be reloaded</td> </tr> </table>	DPT\$M_SUBCNTRL	Device is a subcontroller	DPT\$M_SVP	Device requires permanent system page; allocated during driver loading	DPT\$M_NOUNLOAD	Driver cannot be reloaded
DPT\$M_SUBCNTRL	Device is a subcontroller						
DPT\$M_SVP	Device requires permanent system page; allocated during driver loading						
DPT\$M_NOUNLOAD	Driver cannot be reloaded						

(continued on next page)

THE I/O DATA BASE

Table A-11 (Cont.)
Contents of Driver Prologue Table

Field Name	Contents														
DPT\$W_UCBSIZE	<p>Size in bytes of unit control blocks created for device units driven by this driver. Every driver must specify a value for this field as the value of the argument UCBSIZE in the invocation of the DPTAB macro.</p> <p>The driver loading procedure allocates blocks of nonpaged system memory of the specified size when creating UCBS for devices associated with the driver.</p>														
DPT\$W_INITTAB	<p>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.</p> <p>The driver invokes the VAX/VMS macro DPT_STORE to specify these fields and their values. Every driver must specify the following fields:</p> <table data-bbox="673 892 1380 1008"> <tr> <td>UCB\$B_FIPL</td> <td>Fork interrupt priority level</td> </tr> <tr> <td>UCB\$B_DIPL</td> <td>Device interrupt priority level</td> </tr> </table> <p>Other commonly initialized fields are:</p> <table data-bbox="673 1081 1315 1249"> <tr> <td>UCB\$L_DEVCHAR</td> <td>Device characteristics</td> </tr> <tr> <td>UCB\$B_DEVCLASS</td> <td>Class of device</td> </tr> <tr> <td>UCB\$B_DEVTYPE</td> <td>Type of device</td> </tr> <tr> <td>UCB\$W_DEVBUSIZ</td> <td>Default buffer size</td> </tr> <tr> <td>UCB\$L_DEVDEPEND</td> <td>Device-dependent parameters</td> </tr> </table>	UCB\$B_FIPL	Fork interrupt priority level	UCB\$B_DIPL	Device interrupt priority level	UCB\$L_DEVCHAR	Device characteristics	UCB\$B_DEVCLASS	Class of device	UCB\$B_DEVTYPE	Type of device	UCB\$W_DEVBUSIZ	Default buffer size	UCB\$L_DEVDEPEND	Device-dependent parameters
UCB\$B_FIPL	Fork interrupt priority level														
UCB\$B_DIPL	Device interrupt priority level														
UCB\$L_DEVCHAR	Device characteristics														
UCB\$B_DEVCLASS	Class of device														
UCB\$B_DEVTYPE	Type of device														
UCB\$W_DEVBUSIZ	Default buffer size														
UCB\$L_DEVDEPEND	Device-dependent parameters														
DPT\$W_REINITTAB	<p>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.</p> <p>The driver invokes the VAX/VMS macro DPT_STORE to specify these fields and their values. Every driver must specify the following field:</p> <table data-bbox="673 1543 1299 1585"> <tr> <td>DDB\$L_DDT</td> <td>Driver dispatch table</td> </tr> </table> <p>Other commonly initialized fields are:</p> <table data-bbox="673 1659 1364 1848"> <tr> <td>CRB\$L_INTD+4</td> <td>Interrupt service routine</td> </tr> <tr> <td>CRB\$L_INTD2+4</td> <td>Second interrupt service routine</td> </tr> <tr> <td>VEC\$L_INITIAL</td> <td>Controller initialization routine</td> </tr> <tr> <td>VEC\$L_UNITINIT</td> <td>Unit initialization routine</td> </tr> </table>	DDB\$L_DDT	Driver dispatch table	CRB\$L_INTD+4	Interrupt service routine	CRB\$L_INTD2+4	Second interrupt service routine	VEC\$L_INITIAL	Controller initialization routine	VEC\$L_UNITINIT	Unit initialization routine				
DDB\$L_DDT	Driver dispatch table														
CRB\$L_INTD+4	Interrupt service routine														
CRB\$L_INTD2+4	Second interrupt service routine														
VEC\$L_INITIAL	Controller initialization routine														
VEC\$L_UNITINIT	Unit initialization routine														

(continued on next page)

THE I/O DATA BASE

Table A-11 (Cont.)
Contents of Driver Prologue Table

Field Name	Contents
DPT\$W_UNLOAD	Relative address of a driver action routine to be called when a driver is reloaded. The driver specifies this field with the value of the UNLOAD argument in the invocation of the macro DPTAB. If the driver requires special clean-up processing such as buffer or map register deallocation before the driver can be reloaded, the driver must specify this field. The driver loading procedure calls the driver unloading routine before reinitializing all device units associated with the driver.
DPT\$W_VERSION	The DPTAB macro fills this field with the current version of VAX/VMS. This field is checked at driver load time for the correct value.
DPT\$T_NAME	Name of the device driver. Field is 12 bytes in length. One byte records the length of the name string; the name string can be up to 11 characters in length. Drivers specify this field as the value of the NAME argument in the invocation of the DPTAB macro. The driver loading procedure compares the name of a driver to be loaded with the values in this field in all DPTs already loaded into system memory. If the procedure finds a match, the procedure unloads the old driver and replaces it with the new driver. Otherwise, the procedure adds a new DPT to the DPT linked list and then loads the new driver.

A.9 CHANNEL CONTROL BLOCK (CCB)

When a process assigns an I/O channel to a device unit with the Assign I/O Channel system service, EXE\$ASSIGN locates a free block among the process's preallocated channel control blocks. EXE\$ASSIGN then writes a description of the device attached to the channel in the CCB.

The fields of a channel control block are illustrated in Figure A-12 and described in Table A-12.

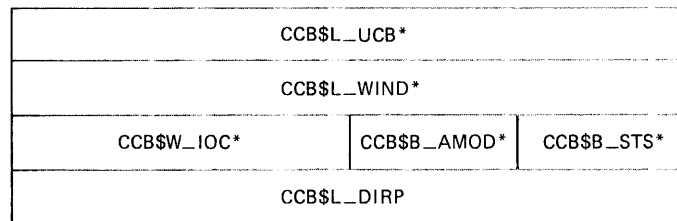


Figure A-12 Channel Control Block

THE I/O DATA BASE

Table A-12
Contents of Channel Control Block

Field Name	Contents
CCB\$L_UCB*	Address of the unit control block of the assigned device unit. EXE\$ASSIGN writes a value into this field. EXE\$QIO reads this field to determine that the I/O request specifies a process I/O channel assigned to a device and to obtain the device's UCB address.
CCB\$L_WIND*	Address of window control block for a file-structured device assignment. This field is written by an ACP and read by EXE\$QIO. A file-structured device's ACP creates a window control block when a process accesses a file on a device assigned to a process channel. The window control block maps the virtual block numbers of the file to a series of physical locations on the device.
CCB\$B_STS*	Channel status.
CCB\$B_AMOD*	Access mode plus 1 of the process at the time of the channel assignment. EXE\$ASSIGN writes the process access mode value into this field.
CCB\$W_IOC*	Number of outstanding I/O requests on the channel. EXE\$QIO increases this field when it begins to process an I/O request that specifies the channel. During I/O postprocessing, the kernel mode AST routine decrements this field. Some FDT routines and EXE\$DEASSIGN read this field.
CCB\$L_DIRP*	Address of deaccess I/O request packet. A number of outstanding I/O requests can be pending on the same process I/O channel at one time. If the process that owns the channel issues an I/O request to deaccess the device, EXE\$QIO holds the deaccess request until all other outstanding I/O requests are processed.

A.10 I/O REQUEST PACKET EXTENSION (IRPE)

I/O request packet extensions hold additional I/O request information for devices that require more context than the standard I/O request packet can accommodate. IRP extensions are also used when more than one buffer (region) must be locked into memory for a direct I/O operation, or when a transfer requires a buffer that is larger than 64K bytes. An IRPE provides space for two buffer regions, each with a 32-bit byte count.

FDT routines allocate IRPEs by calling EXE\$ALLOCIRP. Driver routines link the IRP extension to the I/O request packet by storing the extension's address in two fields within the packet: IRP\$V_EXTEND in IRP\$W_STS and IRP\$L_EXTEND. The FDT routine initializes the contents of the IRPE. Any fields within the extension not described in Table A-13 can store driver-dependent information.

THE I/O DATA BASE

If the IRP extension specifies additional buffer regions, the FDT routine must use those buffer locking routines that perform coroutine calls back to the driver if the locking procedure fails (EXE\$READLOCKR, EXE\$WRITELOCKR, and EXE\$MODIFYLOCKR). If an error occurs during the locking procedure, the driver must unlock all previously locked regions and deallocate the I/O request packet extension before returning to the buffer locking routine.

IOC\$IOPPOST automatically unlocks the pages in region 1 (if defined) and region 2 (if defined) for all the IRP extensions linked to the packet being completed. IOC\$IOPPOST also deallocates all the IRPEs.

The fields of the I/O request packet extension are illustrated in Figure A-13 and described in Table A-13.

spare longword		
spare longword		
spare byte	IRP\$B_TYPE	IRP\$W_SIZE
spare longword		
spare longword		
spare longword		
spare longword		
spare longword		
spare longword		
spare longword		
spare longword		
spare longword		
IRP\$W_STS		spare word
IRP\$L_SVAPTE1		
spare word		IRP\$W_BOFF1
IRP\$L_BCNT1		
IRP\$L_SVAPTE2		
spare word		IRP\$W_BOFF2
IRP\$L_BCNT2		
spare longword		
spare longword		
IRP\$L_EXTEND		

Figure A-13 I/O Request Packet Extension

THE I/O DATA BASE

Table A-13
Contents of the I/O Request Packet Extension

Field Name	Contents
IRPE\$W_SIZE	Size of the I/O request packet extension. EXE\$ALLOCIRP writes the constant IRP\$C_LENGTH to this field.
IRPE\$B_TYPE	Type of control block. EXE\$ALLOCIRP writes the constant DYN\$C_IRP to this field.
IRPE\$W_STS	IRP extension status field. Bits in the status field describe the following conditions: IRPE\$V_EXTEND Another IRPE is linked to this one
IRPE\$L_SVAPTE1	System virtual address of the page table entry mapping the start of region 1. FDT routines write this field. If the region is not defined, this field is zero.
IRPE\$W_BOFF1	Byte offset of region 1. FDT routines write this field.
IRPE\$L_BCNT1	Size in bytes of region 1. FDT routines write this field.
IRPE\$L_SVAPTE2	System virtual address of the page table entry mapping the start of region 2. Set by FDT routines. This field contains a value of zero if region 2 is not defined.
IRPE\$W_BOFF2	Byte offset of region 2. This field is set by FDT routines.
IRPE\$L_BCNT2	Size in bytes of region 2. FDT routines write this field.

APPENDIX B

VAX/VMS MACROS INVOKED BY DRIVERS



This appendix contains an alphabetical listing of macros that drivers invoke. Default values are provided where applicable.

CASE	Generates a CASE instruction and CASE table
SRC	Source of CASE index value
DISPLIST	List of destinations for each case (dest1, dest2, dest3)
TYPE=W	Data type (B, W, L)
LIMIT=#0	Lower limit of CASE value
NMODE=S^#	Address mode for number of table entries; the short literal default is good for up to 63 entries
DDTAB	Generates a driver dispatch table named devnam\$DDT
DEVNAM	Generic device name
START=0	Address of start I/O routine
UNSOLIC=0	Address of unsolicited interrupt service routine for MASSBUS drivers
FUNCTB	Address of function decision table
CANCEL=0	Address of cancel I/O routine
REGDMP=0	Address of error-logging register dump routine
DIAGBF=0	Length in bytes of diagnostic buffer
ERLGBF=0	Length in bytes of error logging buffer
UNITINIT=0	Device unit initialization routine
ALTSTART=0	Alternate start I/O routine
DPTAB	Generates a driver prologue table in PSECT \$\$\$105_PROLOGUE
END	Address of the end of the driver
ADAPTER	Type of adapter (UBA or MBA)
FLAGS=0	Driver loading flags (DPT\$M_SVP and DPT\$M_NOUNLOAD)
UCBSIZE	Size in bytes of each device UCB
[UNLOAD]	Optional address of a routine to call if the driver is to be unloaded
MAXUNITS=8	Maximum number of units that can be connected
NAME	Driver name

VAX/VMS MACROS INVOKED BY DRIVERS

DPT_STORE	Generates a table containing initialization values for fields in the I/O data base
STR_TYPE	Type of control block (DDB, UCB, CRB, IDB); or table marker (INIT, REINIT, END)
STR_OFF	Offset into control block
OPER	Type of initialization operation (B=byte, W=word, L=long, D=address relative to driver, V=bit field); if an @ sign (@) precedes the OPERATION, then the EXPRESSION argument is the address of the initialization data
EXP	Initialization value to be stored in control block
POS	Bit position for OPERATION=V
SIZE	Field size for OPERATION=V
DSBINT	Disables interrupts by raising IPL
[IPL]	IPL value to be loaded into the IPL processor register PR\$IPL (defaults to 31)
[DST]	Location for old IPL value (defaults to top of stack)
ENBINT	Enables interrupts by restoring a saved IPL
[SRC]	Location in which an IPL is saved (defaults to top of stack)
FORK	Calls EXE\$FORK to create a fork process
FUNCTAB	Generates a function decision table consisting of two 64-bit entries of function codes, and n 96-bit entries of function codes and action routine addresses
[ACTION]	Address of an FDT routine to call for the function codes listed
CODES	A list of I/O function codes
IFNORD	Branches if a range of addresses is not readable
SIZ	Number of bytes in range
ADR	Address of first byte in range
DEST	Location to branch to if the range of addresses is not readable
MODE=#0	Access mode at which to probe (defaults to USER)
IFNOWRT	Branches if a range of addresses is not writeable
SIZ	Number of bytes in range
ADR	Address of first byte in range
DEST	Location to branch to if the range of addresses is not writeable
MODE=#0	Access mode at which to probe (defaults to USER)

VAX/VMS MACROS INVOKED BY DRIVERS

IFRD	Branches if a range of addresses is readable
SIZ	Number of bytes in range
ADR	Address of first byte in range
DEST	Location to branch to if the range of addresses is readable
MODE=#0	Access mode at which to probe (defaults to USER)
IOFORK	Calls EXE\$IOFORK to create a device driver fork process
LOADUBA	Calls IOC\$LOADUBAMAP to load a preallocated set of UNIBUS adapter map registers
PURDPR	Calls IOC\$PURGDATAP to purge a data path
RELCHAN	Calls IOC\$RELCHAN to release all controller data channels that are allocated by the driver
RELDPR	Calls IOC\$RELDATAP to release a preallocated UNIBUS adapter data path
RELMPR	Calls IOC\$RELMAPREG to release a preallocated set of UNIBUS adapter map registers
RELSCHAN	Calls IOC\$RELSCHAN to release all secondary controller data channels that are allocated by the driver
	Calls  to complete an I/O request after driver processing is finished
REQDPR	Calls IOC\$REQDATAP to request a UNIBUS adapter data path
REQMPR	Calls IOC\$REQMAPREG to request a set of UNIBUS map registers
REQPCHAN	Calls IOC\$REQPCHANH or IOC\$REQPCHANL to request a primary controller data channel
[PRI]	Priority of request; if PRI=HIGH, calls IOC\$REQPCHANH; otherwise calls IOC\$REQPCHANL
REQSCHAN	Calls IOC\$REQSCHANH or IOC\$REQSCHANL to request a secondary controller data channel
[PRI]	Priority of request; if PRI=HIGH calls IOC\$REQSCHANH; otherwise calls IOC\$REQSCHANL

VAX/VMS MACROS INVOKED BY DRIVERS

SAVIPL Saves the current IPL value as recorded in the processor register PR\$_IPL

 DST=-(SP) Location in which to save the current IPL (defaults to a new top of stack)

SETIPL Sets IPL to a new value

 [IPL] New IPL value (defaults to 31)

SOFTINT Initiates a software interrupt

 IPL IPL value of the interrupt; loads IPL into the processor register PR\$_SIRR

~~WFIKPC~~ Calls an executive subroutine to wait for an interrupt or a device timeout and keep the controller data channel

 EXCPT Relative address of a device timeout handling routine; writes the address into the two bytes following the call to the executive routine.

 [TIME] Number of seconds to allow before a device timeout (defaults to 65536 seconds)

WFIRLCH Calls an executive subroutine to wait for an interrupt or a device timeout and release the controller data channel

 EXCPT Relative address of a device timeout handling routine; writes the address into the two bytes following the call to the executive routine.

 [TIME] Number of seconds to allow before a device timeout (defaults to 65536 seconds)

APPENDIX C
OPERATING SYSTEM ROUTINES

This appendix describes the VAX/VMS operating system routines that are used by device drivers. The information given in this section follows the conventions listed below:

- Fields used for both input and output are not specified.
- Registers are assumed preserved unless otherwise specified.
- IPL at execution refers to the interrupt priority level at which the routine executes, not the IPL at which it is called.

COM\$DELATTNAST in module COMDRVSUB

Driver fork processes call this routine to deliver all the AST control blocks linked to the specified AST list.

INPUT TO ROUTINE

Registers	Contents
R4	Address of specified listhead
R5	Address of the unit control block

Fields	Contents
---------------	-----------------

---	---
-----	-----

IPL at execution: caller's IPL

This routine removes all AST blocks from the specified list and schedules an IPL\$_QUEUEAST level fork process to queue each AST to its process.

OUTPUT FROM ROUTINE

Registers	Contents
------------------	-----------------

---	---
-----	-----

Fields	Contents
---------------	-----------------

Specified listhead	0
--------------------	---

IPL at exit: caller's IPL

OPERATING SYSTEM ROUTINES

COM\$DRVDEALMEM in module COMDRVSUB

Drivers use this routine to deallocate system dynamic memory. COM\$DRVDEALMEM can be called from any interrupt priority level.

INPUT TO ROUTINE

Registers	Contents
R0	Address of the block to be deallocated

Fields	Contents
IRP\$W_SIZE	Size of the block in bytes

IPL at execution: caller's IPL and IPL\$_QUEUEAST

If the block size is smaller than 24 bytes or the block is not properly aligned, a system bugcheck occurs. This routine also calls SCH\$RAVAIL to mark the resource free.

IPL at exit: caller's IPL

COM\$FLUSHATTNS in module COMDRVSUB

Driver FDT and fork routines call this routine to flush an attention AST list. Drivers use this routine during cancel I/O operations.

INPUT TO ROUTINE

Registers	Contents
R4	Address of the current PCB
R5	Address of the UCB
R6	Number of the assigned channel
R7	Address of the AST control block listhead

Fields	Contents
UCB\$_DIPL	Device IPL
PCB\$_PID	Process's ID
PCB\$_ASTCNT	ASTs remaining in quota

IPL at execution: device IPL (UCB\$_DIPL)

COM\$FLUSHATTNS locates all the control blocks whose channel number and process identification match those specified as input to the routine, removes them from the specified list and deallocates them. This routine exits by returning to its caller.

OPERATING SYSTEM ROUTINES

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL
R1	Destroyed
R2	Destroyed
R7	Destroyed
Fields	Contents
PCB\$W_ASTCNT	Number of ACBs flushed (added to previous contents)
Specified listhead	Updated

IPL at exit: caller's IPC

COM\$POST in module COMDRVSUB

Drivers call this routine after they have completed all device-dependent I/O postprocessing for an I/O request. This routine inserts the I/O request packet into the I/O postprocessing queue and returns to the driver fork process. COM\$POST operates independently of the device unit; it does not attempt to dequeue another packet nor does it change the busy status of the device.

Drivers can use this routine to complete I/O request packets initiated by the routine EXE\$ALTQUEPKT.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet
R5	Address of the unit control block
Fields	Contents
IRP\$L_MEDIA	Data to be copied into the I/O status block
IRP\$L_MEDIA+4	Data to be copied to the I/O status block

IPL at execution: caller's IPL (driver fork level or above)

This routine places the I/O request packet into the queue headed by IOC\$GL_PSBL.

OPERATING SYSTEM ROUTINES

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed
R1	Destroyed
Fields	Contents
UCB\$L_OPCNT	Incremented by 1
IPL at exit:	caller's IPL

COM\$SETATTNAST in module COMDRVSUB

Driver FDT routines call this routine to enable or disable attention ASTs, depending upon the contents of the queue I/O parameter P1. To enable an AST, P1 contains the address of an AST routine. The routine allocates a control block that can double as an AST control block when the AST is delivered.

This control block contains the following information:

- The address of the specified AST routine
- The specified AST parameter
- The specified access mode
- The channel number
- The process identification of the requesting process

COM\$SETATTNAST links the control block to the start of the specified linked list of AST control blocks located in the unit control block extension area.

If P1 is clear, the routine disables ASTs by searching through the linked list, extracting each entry, and deallocating it.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the IRP
R4	Address of the current PCB
R5	Address of the UCB
R6	Address of the assigned channel control block
R7	Address of the specified AST control block listhead
AP	Address of the QIO parameter list

OPERATING SYSTEM ROUTINES

Fields	Contents
IRP\$W_CHAN	I/O request channel number
UCB\$B_DIPL	Device IPL
PCB\$W_ASTCNT	Number of ASTs remaining in process quota
PCB\$L_PID	Process identification
0 (AP)	Process AST address
4 (AP)	AST parameter
8 (AP)	Access mode for AST

IPL at execution: caller's IPL and device IPL

If the process exceeds buffered I/O or AST quotas, or if there is no memory available to allocate an AST control block, this routine transfers control to EXE\$ABORTIO with error status.

If P1 is clear, the routine transfers control to COM\$FLUSHATTNS to remove the identified AST control block.

This routine exits to its caller.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL (success) SS\$_EXQUOTA SS\$_INSFMEM
R1	Destroyed
R2	Destroyed
R3	Address of the IRP
R5	Address of the UCB
R6	Destroyed
R7	Destroyed
R8	Destroyed

Fields	Contents
DCB\$W_ASTCNT	Decreased by 1
Specified listhead	Updated

IPL at exit: caller's IPL

OPERATING SYSTEM ROUTINES

ERL\$DEVICERR in module ERRORLOG

Logs a controller and/or device error. This routine allocates an error message buffer and writes data from the I/O request packet and unit control block. It also calls the driver register dump routine for device registers.

INPUT TO ROUTINE

Registers	Contents
R5	Address of unit control block

ERL\$DEVICERR sets the error type code to device error. This routine uses fields in the UCB, DDB, DDT, and I/O request packet. It also assumes that the driver contains a register dump routine. It uses the DDT to calculate the address of the register dump routine and then calls it.

If you do not specify a dump routine in the DDTAB macro invocation, DDTAB supplies the address of IOC\$RETURN. IOC\$RETURN simply returns; it is a NOP.

OUTPUT FROM ROUTINE

Registers	Contents
---	---

Fields	Contents
UCB\$L_EMB	Address of the error message buffer
UCB\$W_STS	Shows error log in progress

ERL\$DEVICTMO in module ERRORLOG

Logs a device timeout. This routine performs the same functions and uses the same input and output as ERL\$DEVICERR with one exception: the error type code is device timeout.

ERL\$RELEASEMB in module ERRORLOG

Wakes the error log process to write the contents of an error message buffer into the error logging file.

INPUT TO ROUTINE

Registers	Contents
R2	Address of error message buffer

Fields	Contents
ERL\$V_TIMER (in ERL\$GB_BUFFLAG)	Determines whether a timer is running on the buffer

IPL at execution: caller's IPL

OPERATING SYSTEM ROUTINES

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed
Fields	Contents
Busy message count (in ERL\$B_BUSY)	Decreased by 1
Complete message count (in error message buffer header)	Incremented by 1

If ERL\$B_MSGCNT is greater than the maximum message count, this routine wakes the error logger.

IPL at exit: caller's IPL

EXE\$ABORTIO in module SYSQIOREQ

FDT routines jump to this routine to finish an I/O operation without returning final I/O status in the IOSB. This routine zeroes the IOSB field of the I/O request packet, clears a bit to prevent a user mode AST, and inserts the I/O request packet in the I/O postprocessing queue.

INPUT TO ROUTINE

Registers	Contents
R0	First longword of status for I/O status block
R3	Address of I/O request packet
R4	Address of current PCB
R5	Address of UCB

Fields	Contents
ACB\$V_QUOTA (in IRP\$B_RMOD)	Set to 1 (when an AST is specified)

IPL at execution: IPL\$ASTDEL

OUTPUT FROM ROUTINE

Registers	Contents
None written	---
Fields	Contents
ACB\$V_QUOTA (in IRP\$B_RMOD)	Cleared to zero (if field previously set)
IRP\$L_IOSB	Zero

OPERATING SYSTEM ROUTINES

PCB\$W_ASTCNT Incremented if ACB\$V_QUOTA was set

EXE\$ABORTIO places the I/O request packet into the I/O postprocessing queue headed by IOC\$GL_PSBL.

IPL at exit: 0 (normal process IPL)

EXE\$ALLOCBUF in module MEMORYALC

FDT routines call this routine to allocate a buffer for a buffered I/O operation from the nonpaged system pool. This routine can place the process in a resource wait state if sufficient memory is not available, and the process has resource wait mode enabled. The caller must adjust process quotas.

INPUT TO ROUTINE

Register	Contents
R1	Size of requested buffer in bytes
R4	Address of current PCB
Fields	Contents
PCB\$V_SSRWAIT	One or zero. Determines whether process should wait, if no memory available for requested buffer. If this field is set, resource wait mode is disabled.

IPL at execution: caller's IPL, IPL 11, and IPL\$_SYNCH

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL (success) SS\$_INSFMEM
R1	Size of allocated buffer (requested size is rounded up to next 16-byte multiple)
R2	Address of allocated buffer
R3	Destroyed
Fields	Contents
IRP\$W SIZE (in allocated buffer)	Buffer size in bytes
IRP\$B TYPE (in allocated buffer)	DYN\$_BUFIO

IPL at exit: IPL\$_ASTDEL

OPERATING SYSTEM ROUTINES

EXE\$ALLOCIRP in module MEMORYALC

This routine allocates an I/O request packet from nonpaged dynamic memory. It performs the same functions and has the same input and output as EXE\$ALLOCBUF, with the following exceptions:

- The caller does not specify a buffer size
- The allocated buffer is IRP\$C_LENGTH bytes long
- The buffer size is set to IRP\$C_LENGTH
- The buffer type is set to DYN\$C_IRP

EXE\$ALONONPAGED in module MEMORYALC

Driver fork processes use this routine to allocate a block of memory from the nonpaged system pool.

The block header is not initialized.

INPUT TO ROUTINE

Registers	Contents
R1	Requested block size in bytes

Fields	Contents
none	---

IPL at execution: caller's IPL and IPL 11

OUTPUT FROM ROUTINE

Registers	Contents
R0	Status code (0 or 1)
R1	Size of allocated buffer (requested size rounded up to next 16-byte multiple)
R2	Address of allocated block
R3	Destroyed

Fields	Contents
---	---

IPL at exit: caller's IPL

EXE\$ALTQUEPKT in module SYSQIOREQ

Driver FDT routines and fork processes call this routine to send an I/O request packet to a driver's alternate start I/O routine so that it bypasses the I/O request queue for the device's unit control block. EXE\$ALTQUEPKT passes the address of the I/O request packet to the driver without regard for the status of the device unit.

OPERATING SYSTEM ROUTINES

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet
R5	Address of the unit control block
Fields	Contents
DDT\$L_ALTSTART	Address of the alternate start I/O routine
UCB\$B_FIPL	Driver fork IPL
UCB\$L_DDB	Address of unit's DDB
DDB\$L_DDT	Address of the driver dispatch table
IPL at execution: UCB\$L_FIPL	

EXE\$ALTQUEPKT calls the alternate start I/O routine and returns to its caller.

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed
R1	Destroyed
R2-R5	Destroyed
IPL at exit: caller's IPL	

EXE\$BUFFERQUOTA in module EXSUBROUT

FDT routines call this routine to determine whether a process's buffered byte count quota usage permits the process to be granted additional buffered I/O. This routine may place the process in a resource wait state if quota usage is too large, and the process has resource wait mode enabled.

INPUT TO ROUTINE

Registers	Contents
R1	Number of requested bytes
R4	Address of PCB

OPERATING SYSTEM ROUTINES

Fields	Contents
PCB\$V_SSRWAIT	When process exceeds quota, determines whether process should wait. If this field is set, resource wait mode is disabled.
IOC\$GW_MAXBUF	Maximum number of buffered I/O bytes that system allows to any process
JIB\$L_BYTLM	Process's byte count limit
JIB\$L_BYTCNT	Process's byte count usage quota
IPL at execution:	caller's IPL and IPL\$_SYNCH

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL (success) SS\$_EXQUOTA
R3	Destroyed

Fields	Contents
---	---

IPL at exit: IPL\$_ASTDEL

EXE\$BUFQUOPRC in module EXSUBROUT

EXE\$BUFQUOPRC performs the same function and has the same input and output as EXE\$BUFRQUOTA with the following exception: EXE\$BUFQUOPRC does not check the field IOC\$GW_MAXBUF.

EXE\$DEANONPAGED in module MEMORYALC

Deallocates a block of memory to the nonpaged system pool.

This routine performs the same functions and has the same input and output as the routine COM\$DRVDEALMEM, with the following exceptions:

- R3 is destroyed
- The caller's IPL must be at IPL\$_QUEUEAST or lower

OPERATING SYSTEM ROUTINES

EXE\$FINISHIO in module SYSQIOREQ

FDT routines transfer control to this routine to finish an I/O operation and return a quadword of final I/O status to the requesting process. This routine writes final I/O status into the I/O request packet and inserts the I/O request packet in the I/O postprocessing queue.

INPUT TO ROUTINE

Registers	Contents
R0	First longword of status for the I/O status block
R1	Second longword of status for the I/O status block
R3	Address of the I/O request packet
R4	Address of the current process control block
R5	Address of the UCB

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL

Fields	Contents
IRP\$L_MEDIA	First longword of I/O status (R0)
IRP\$L_MEDIA+4	Second longword of I/O status (R1)
UCB\$L_OPCNT	Incremented by 1

This routine places the I/O request packet into the I/O postprocessing queue headed by IOC\$GL_PSBL.

EXE\$FINISHIOC in module SYSQIOREQ

This routine performs the same functions and has the same input and output as EXE\$FINISHIO with the following exception: EXE\$FINISHIOC clears the contents of R1 before storing R0 and R1 in the I/O request packet.

EXE\$FORK in module FORKCNTL

This routine performs the same functions as EXE\$IOFORK except that this routine does not disable timeouts by clearing UCB\$V_TIM in the UCB\$W_STS field of the unit control block.

OPERATING SYSTEM ROUTINES

EXE\$FORKDSPTH in module FORKCNTRL

The interrupt service routine that dispatches fork processes in a fork queue. This routine gains control when the processor grants a software interrupt at IPLs 6 and 8 through 11. When EXE\$FORKDSPTH gains control the stack contains the following information:

- 0(SP) contains the PC at the time of the interrupt
- 4(SP) contains the PSL at the time of the interrupt

R0 through R5 at the time of the interrupt are also saved by EXE\$FORKDSPTH.

SWI\$GL_FQFL indexed by the current IPL contains the address of the head of the fork queue for this IPL. Each entry in the fork queue is the address of a fork block that contains R3, R4, a PC, and implicitly R5; R5 is the address of the fork block.

If the queue is empty when the interrupt occurs, EXE\$FORKDSPTH dismisses the interrupt without error.

EXE\$FORKDSPTH empties the fork queue corresponding to the IPL of the interrupt. For each queue entry, it restores R3 and R4 from the fork block, saves the dispatch address and IPL on the stack, and executes a JSB to the saved PC address. When the queue is empty, it dismisses the interrupt.

The IPL on return from each fork process must equal the IPL at which the process was called. If IPL does not match, EXE\$FORKDSPTH signals the fatal bugcheck BADFORKIPL.

EXE\$INSERTIRP in SYSQIOREQ.MAR

Inserts an I/O request packet according to the base priority of the I/O request packet's originating process into the I/O request packet wait queue of a unit control block.

INPUT TO ROUTINE

Register	Contents
R2	Address of the I/O queue list head for the device
R3	Address of the I/O request packet

Fields	Contents
--------	----------

--- ---

IPL at execution: caller's IPL (fork level or higher)

OUTPUT FROM ROUTINE

Register	Contents
R1	Destroyed

OPERATING SYSTEM ROUTINES

This routine places the I/O request packet in the queue and sets the Z condition code in the PSL as follows:

1 indicates that the entry is first in the queue.

0 indicates that at least one entry was already in the queue.

IPL at exit: caller's IPL

EXE\$INSIOQ in SYSQIOREQ.MAR

Examines the unit control block. If the device is idle, this routine calls IOC\$INITIATE; if the device is busy, it calls EXE\$INSERTIRP.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet
R5	Address of the UCB
Fields	Contents
UCB\$B_FIPL	Driver fork IPL
UCB\$V_BSY (in UCB\$W_STS)	Determines whether device is busy
UCB\$L_IOQFL	Address of device I/O queue listhead

IPL at execution: driver fork level

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed
R1	Destroyed
R2	Destroyed

Additional registers used by the driver start I/O routine will be destroyed if the start I/O routine is called.

Fields	Contents
UCB\$V_BSY (in UCB\$W_STS)	Set to 1

IPL at exit: original IPL

OPERATING SYSTEM ROUTINES

EXE\$IOFORK in module FORKCNTRL

Saves the contents of R3 and R4 in the fork block specified by R5. This routine pops the return PC off the top of stack and saves the PC value in the fork block. It inserts the fork block address into the fork queue corresponding to the IPL stored in the fork block.

INPUT TO ROUTINE

Register	Contents
R5	Address of the fork block (usually the UCB)
0(sp)	Return address of caller
4(sp)	Return address of caller's caller

Fields	Contents
FKB\$B_FIPL (in fork block)	Fork IPL

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R3	Destroyed
R4	FKB\$B_FIPL

Fields	Contents
UCB\$V_TIM (in UCB\$W_STS)	Zero
FKB\$L_FR3 (in UCB)	R3
FKB\$L_FR4 (in UCB)	R4
FKB\$L_FPC (in UCB)	0(SP)

The routine queues the UCB address to the list headed by SWI\$GL_FQFL. If the queue is empty, requests a software interrupt at fork IPL.

IPL at exit: caller's IPL

OPERATING SYSTEM ROUTINES

EXE\$MODIFY in module SYSQIOFDT

FDT routines transfer control to this device-independent routine that validates and readies a user buffer for a DMA read/write operation. Use EXE\$MODIFY instead of EXE\$READ when you wish your driver to read and write to a buffer. EXE\$MODIFY disables a paging mechanism used during write-only operations.

This routine performs the following functions:

- Translates read logical functions to read physical functions
- Transfers queue I/O parameters to the I/O request packet
- Verifies that the caller has access to the specified buffer
- Locks the buffer's pages into physical memory. If a page fault occurs during this step, the routine returns control to the Queue I/O Request system service, which repeats the request.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet
R4	Address of the current PCB
R5	Address of the UCB assigned to the device unit
R6	Address of the CCB for the channel assigned to the device unit
R7	Bit number of the I/O function code
R8	FDT entry address
AP	Address of the first function-dependent QIO parameter (P1)

Fields	Contents
0(AP)	Virtual address of buffer (P1)
4(AP)	Number of bytes in transfer (P2)
12(AP)	Carriage control byte (P4)
IRP\$W_FUNC	I/O function code

IPL at execution: caller's IPL (IPL\$ASTDEL)

If this routine completes successfully, it transfers control to EXE\$QIODRVPKT. If EXE\$MODIFY fails, it transfers control to EXE\$ABORTIO.

OPERATING SYSTEM ROUTINES

OUTPUT FROM ROUTINE

Registers	Contents
R0, R1, R2	Destroyed
Fields	Contents
IRP\$B_CARCON	P4
IRP\$V_FUNC (in IRP\$W_STS)	Set to 1 (indicates a read function)
IRP\$L_SVAPTE	Address of page table entry that maps the first page of the buffer
IRP\$W_BCNT	Size of the transfer in bytes
IPL at exit:	caller's IPL

EXE\$MODIFYLOCK in module SYSQIOFDT

FDT routines call EXE\$MODIFYLOCK to perform buffer processing on a DMA transfer. This routine:

- Determines whether the caller has write access to the buffer
- Locks the buffer's pages into memory. If a page fault occurs during this process, the routine returns control to the Queue I/O Request system service, which resubmits the request.

Use EXE\$MODIFYLOCK instead of EXE\$READLOCK when you expect your driver to read and write to a buffer. EXE\$MODIFYLOCK disables a paging mechanism used in write-only operations.

INPUT TO ROUTINE

Registers	Contents
R0	Starting address of buffer
R1	Size of transfer in bytes
R3	Address of the I/O request packet
R4	Address of current PCB
R6	Address of the CCB
Fields	Contents
---	---

IPL at execution: caller's IPL (IPL\$ASTDEL)

OPERATING SYSTEM ROUTINES

If EXE\$MODIFYLOCK fails, it transfers control to EXE\$ABORTIO. If the routine completes successfully, control passes to EXE\$QIODRVPKT.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL
R1	Address of the PTE that maps the first page of the buffer
R2	Destroyed
R3	Address of the IRP
Fields	Contents
IRP\$L_SVAPTE	Address of the PTE that maps the first page of the buffer
IRP\$W_BCNT	Size of the transfer in bytes
IRP\$V_FUNC (in IRP\$W_STS)	A value of 1 (indicating a read function)
IPL at exit:	caller's IPL

EXE\$MODIFYLOCKR in module SYSQIOFDT

This routine determines whether a process has write access to the buffer pages it requested, then, if access is permitted, it locks the pages into memory. If the access check or page locking procedure fails, the routine calls the driver to clean up QIO bookkeeping. Drivers typically use EXE\$MODIFYLOCKR when they must lock multiple areas into memory for one I/O request, and then need to unlock previously locked areas after a QIO is aborted.

INPUT TO ROUTINE

Registers	Contents
R0	Starting address of buffer
R1	Length of the buffer in bytes
R3	Address of the IRP
R4	Address of the current process's PCB
R6	Address of the channel control block
Fields	Contents
---	---

OPERATING SYSTEM ROUTINES

EXE\$MODIFYLOCKR may fail for a number of reasons:

- The buffer access check fails. In this case, the routine returns SS\$_ACCVIO to the driver in R0.
- The caller process has an insufficient working set limit to lock all the buffer pages into memory. The routine returns SS\$_INSFWSL in R0.
- A page fault occurs while the routine is locking pages into memory. The status returned in R0 in this case is zero.

If any of the above errors occur, the routine calls back the driver as a coroutine with error status in R0 and all other registers preserved. The driver performs necessary queue I/O cleanup, that is, it carries out any procedures that the system does not perform as part of the normal QIO abort processing.

The driver must preserve all registers, including R0 and R1.

When the driver returns by executing an RSB instruction, EXE\$MODIFYLOCKR aborts the I/O request if R0 contains an error status, then performs processing that results in the I/O request's being resubmitted to the driver. For example:

```
          JSB          G^EXE$MODIFYLOCKR
          BLBS         BUF_LOCK_OK

BUF_LOCK_FAIL:
          <clean up this QIO bookkeeping>
          RSB

BUF_LOCK_OK:
          <continue this QIO>
```

If the subroutine is successful, it returns control to its caller.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL (1)
R1	Address of the PTE that maps the first page of the buffer
R2	Function indicator (set to 1)
R3	Address of the IRP

Fields	Contents
IRP\$L_SVAPTE	Address of the PTE that maps the first page of the buffer
IRP\$W_BCNT	Size of the transfer in bytes
IRP\$M_FUNC (in IRP\$W_FUNC)	Set to 1

IPL at exit: caller's IPL

OPERATING SYSTEM ROUTINES

EXE\$ONEPARM in module SYSQIOFDT

Device-independent FDT routine that copies a single QIO parameter into the I/O request packet and calls EXE\$QIODRVPKT.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet for the current I/O request
R4	Address of the process control block of the current process
R5	Address of the unit control block of the device assigned to the user-specified process I/O channel
R6	Address of the channel control block that describes the user-specified process I/O channel
R7	Bit number of the user-specified I/O function code
R8	Address of FDT entry
AP	Address of the first function-dependent parameter specified in the user's request

Fields	Contents
--------	----------

---	---
-----	-----

OUTPUT FROM ROUTINE

Registers	Contents
-----------	----------

---	---
-----	-----

Fields	Contents
--------	----------

IRP\$ <u>L</u> MEDIA (of \overline{IRP})	P1
--	----

IPL at exit: caller's IPL

This routine exits to EXE\$QIODRVPKT.

Chapter 8 provides more information about this routine.

EXE\$QIODRVPKT in module SYSQIOREQ

FDT routines call this routine to send an IRP to a driver start I/O routine. This routine calls EXE\$INSIOQ and then transfers control to EXE\$QIORETURN.

OPERATING SYSTEM ROUTINES

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet
R4	Address of the process control block
R5	Address of the unit control block

Fields	Contents
UCB\$B_FIPL	Driver fork IPL
UCB\$V_BSY (in UCB\$W_STS)	Unit busy flag
UCB\$L_IOQFL	Address of unit I/O queue listhead

EXE\$QIORETURN in module SYSQIOREQ

Sets a success status code in R0, lowers IPL to 0, and returns to the system service dispatcher.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL

IPL at exit: 0

This routine returns by issuing a RET instruction.

EXE\$READ in module SYSQIOFDT

Device-independent FDT routine that validates and readies a user buffer for a DMA read operation. This routine performs the same functions and has the same input and output as EXE\$MODIFY, with a single exception noted in the description of EXE\$MODIFY.

EXE\$READCHK in module SYSQIOFDT

Checks pages for write accessibility by a process. This routine writes the total byte count of a transfer into the I/O request packet.

If pages do not allow write access, the routine transfers control to EXE\$ABORTIO, which terminates the request with access violation status. If EXE\$READCHK completes successfully, control returns to its caller.

OPERATING SYSTEM ROUTINES

INPUT TO ROUTINE

Registers	Contents
R0	Address of buffer
R1	Size of the transfer in bytes
R3	Address of the I/O request packet

Fields	Contents
---	---

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R0	Address of buffer (success)
R1	Size of transfer in bytes
R2	Value of 1 (to indicate a read)
R3	Address of IRP

Fields	Contents
IRP\$W_BCNT	Size of transfer in bytes
IRP\$V_FUNC (in IRP\$W_STS)	Value of 1 (indicates a read function)

IPL at exit: caller's IPL

EXE\$READCHKR in module SYSQIOFDT

This routine performs the same function as EXE\$READCHK, except that, upon error, it calls the driver FDT routine back as a coroutine to clean up QIO bookkeeping. See the description of error procedures in EXE\$MODIFYLOCKR for further information.

EXE\$READLOCK in module SYSQIOFDT

FDT routines call this routine to check buffer accessibility and lock the user buffer in memory for a DMA read transfer. This routine performs the same functions and has the same input and output as EXE\$MODIFYLOCK, except that it is used when the driver performs only a read I/O function.

EXE\$READLOCKR in module SYSQIOFDT

This subroutine determines whether a process has write access to requested buffer pages and, if access is permitted, it locks those pages into memory. EXE\$READLOCKR performs the same functions and has the same input and output as EXE\$MODIFYLOCKR.

OPERATING SYSTEM ROUTINES

EXE\$SENSEMODE in module SYSQIOFDT

Device-independent FDT routine that copies device-dependent characteristics from the device's UCB into R1. This routine writes a success code into R0 and transfers control to EXE\$FINISHIO.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet for the current I/O request
R4	Address of the PCB of the current process
R5	Address of the UCB of the device assigned to the user-specified process I/O channel
R6	Address of the CCB that describes the user-specified process I/O channel
R7	Bit number of the user-specified I/O function code
R8	Address of function decision table dispatch
AP	Address of the first function-dependent parameter specified in the user's request

Fields	Contents
UCB\$L_DEVDEPEND	Device-dependent status
IPL at execution:	caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL
R1	Device-dependent characteristics copied from UCB\$L_DEVDEPEND

Fields	Contents
---	---

IPL at exit: caller's IPL

This routine exits to EXE\$FINISHIO.

For additional information, refer to Chapter 8.

OPERATING SYSTEM ROUTINES

EXE\$SETCHAR in module SYSQIOFDT

Device-independent FDT routine that writes a quadword whose address is QIO parameter P1 into the device's unit control block. Writes a success code into R0 and transfers control to EXE\$FINISHIO.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the IRP for the current I/O request
R4	Address of the current PCB
R5	Address of the UCB of the assigned device unit
R6	Address of the CCB that describes the specified process I/O channel
R7	Bit number of the I/O function code
R8	Address of the FDT dispatcher
AP	Address of the first function-dependent QIO parameter

Fields	Contents
0 (AP)	Address of new device characteristics (P1)

IPL at execution: caller's IPL

If this routine fails because the user lacks read access to the characteristics quadword, control transfers to EXE\$ABORTIO with access violation status.

If EXE\$SETCHAR completes successfully, it transfers control to EXE\$FINISHIO.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL (success) SS\$_ACCVIO (failure)

Fields	Contents
UCB\$_DEVCLASS	Byte 0 of quadword
UCB\$_DEVTYPE	Byte 1 of quadword
UCB\$_DEVBUFSIZ	Bytes 2 and 3 of quadword
UCB\$_DEVDEPEND	Bytes 4 through 7 of quadword

IPL at exit: caller's IPL

Refer to Chapter 8 for additional information on this routine.

OPERATING SYSTEM ROUTINES

EXE\$SETMODE in module SYSQIOFDT

Device-independent FDT routine that writes a quadword whose address is a QIO parameter into the I/O request packet and calls EXE\$QIODRVPKT.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the I/O request packet for the current I/O request
R4	Address of the PCB of the current process
R5	Address of the UCB of the device assigned to the user-specified process I/O channel
R6	Address of the CCB that describes the user-specified process I/O channel
R7	Bit number of the I/O function code
R8	Address of the FDT entry
AP	Address of the first function-dependent QIO parameter

Fields	Contents
P0 (AP)	Address of a quadword of device characteristics

IPL at execution: caller's IPL

If the user lacks read access to the device characteristics quadword, the routine transfers control to EXE\$ABORTIO with access violation status. If EXE\$SETMODE completes successfully, it normally exits to EXE\$QIODRVPKT.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL (success) SS\$_ACCVIO

Fields	Contents
IRP\$L_MEDIA	First longword of device characteristics quadword
IRP\$L_MEDIA+4	Second longword of device characteristics quadword

IPL at exit: caller's IPL

For more information about this routine, refer to Chapter 8.

OPERATING SYSTEM ROUTINES

EXE\$SNDEVMSG in module MBDRIVER

Driver fork processes call this routine to send messages to system processes such as OPCOM. This routine constructs a message on the stack and calls EXE\$WRMAILBOX to send the message to a mailbox.

INPUT TO ROUTINE

Registers	Contents
R3	Address of the mailbox UCB
R4	Message type
R5	Address of the UCB

Fields	Contents
UCB\$W_INIT	Device unit number
UCB\$L_DDB	Address of device DDB
DDB\$T_NAME	Device controller name

Mailbox UCB fields

IPL at execution: caller's IPL and IPL\$MAILBOX

This routine can fail for one of the following reasons:

- The message is too large for the mailbox
- The message mailbox is full of messages
- The system is unable to allocate memory for the message

If any of the above conditions occur, EXE\$SNDEVMSG returns error status to the caller.

If EXE\$SNDEVMSG completes successfully, it exits with an RSB instruction.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$NORMAL (success) SS\$MBTOOSML SS\$MBFULL SS\$INSFMEM
R1	Destroyed
R2	Destroyed
R3	Destroyed
R4	Destroyed
R5	Destroyed

OPERATING SYSTEM ROUTINES

Fields	Contents
--------	----------

IPL at exit: caller's IPL

EXE\$WRITE in module SYSQIOFDT

Device-independent FDT routine that validates and readies a user buffer for a DMA write operation. This routine performs the same steps as EXE\$MODIFY, and has the same input and output.

EXE\$WRITECHK in module SYSQIOFDT

Checks pages for read accessibility by a process and writes the total byte count of a transfer into the I/O request packet. If pages do not allow read access, the routine transfers control to EXE\$ABORTIO, which terminates the request with access violation status. If EXE\$WRITECHK completes successfully, it exits to its caller.

INPUT TO ROUTINE

Registers	Contents
R0	Address of buffer
R1	Size of the transfer in bytes
R3	Address of the I/O request packet

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R0	Buffer address (success)
R1	Size of the transfer in bytes
R2	Cleared (indicates a write function)
R3	Address of the I/O request packet

Fields	Contents
IRP\$W_BCNT	Contains transfer size in bytes

IPL at exit: caller's IPL

EXE\$WRITECHKR in module SYSQIOFDT

This routine performs the same functions as EXE\$WRITECHK, except that, upon error, it calls the driver FDT routine back as a coroutine to clean up QIO bookkeeping.

See the description of error procedures in EXE\$MODIFYLOCKR for more information about coroutine cleanup.

OPERATING SYSTEM ROUTINES

EXE\$WRITELOCK in module SYSQIOFDT

FDT routines call this routine to determine whether the caller has read access to the buffer and to lock the buffer in memory for a DMA write transfer.

INPUT TO ROUTINE

Register	Contents
R0	Starting address of I/O buffer
R1	Length of transfer in bytes
R3	Address of the I/O request packet
R4	Address of the PCB
R6	Address of the CCB

Fields	Contents
--------	----------

--- ---

IPL at execution: caller's IPL (IPL\$ASTDEL)

This routine calls EXE\$WRITECHK and MMG\$IOLOCK. MMG\$IOLOCK locks pages in memory. If EXE\$WRITELOCK fails because a page fault occurs during the locking procedure, it transfers control to the Queue I/O Request system service, which repeats the I/O request. It exits to EXE\$ABORTIO if it cannot complete successfully. If the routine does complete without error, it exits to EXE\$QIODRVPKT.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL
R1	Address of the PTE that maps the first page of the buffer
R2	Destroyed
R3	Address of the IRP

Fields	Contents
--------	----------

IRP\$L_SVAPTE	Address of the PTE that maps the first page of the buffer
IRP\$W_BCNT	Size of the transfer in bytes
IRP\$V_FUNC (in IRP\$W_STS)	A value of 0 (indicating a write function)

IPL at exit: caller's IPL

OPERATING SYSTEM ROUTINES

EXE\$WRITELOCKR in module SYSQIOFDT

This routine determines whether the process has read access to the requested buffer pages, and, if access is permitted, it locks those pages into memory. EXE\$WRITELOCKR performs the same functions as EXE\$MODIFYLOCKR, with the following exceptions:

- R2, on output, contains a zero to indicate a write function.
- IRP\$M_FUNC (in IRP\$W_FUNC) is clear (zero) to indicate a write function)

EXE\$ZEROPARM in module SYSQIOFDT

Device-independent FDT routine that clears the parameter field of the IRP and calls EXE\$QIODRVPKT.

INPUT TO ROUTINE

Register	Contents
R3	Address of the I/O request packet for the current I/O request
R4	Address of the process control block of the current process
R5	Address of the unit control block of the device assigned to the user-specified process I/O channel
R6	Address of the channel control block that describes the user-specified process I/O channel
R7	Bit number of the user-specified I/O function code
R8	Address of FDT entry
AP	Address of the first function-dependent parameter specified in the user's request

Fields	Contents
--------	----------

---	---
-----	-----

IPL at execution: caller's IPL

This routine exits by transferring control to EXE\$QIODRVPKT.

OUTPUT FROM ROUTINE

Registers	Contents
-----------	----------

---	---
-----	-----

Fields	Contents
--------	----------

IRP\$L_MEDIA	Zero
--------------	------

OPERATING SYSTEM ROUTINES

IPL at exit: caller's IPL

For additional information, refer to Chapter 8.

IOC\$ALOUBAMAP(N) in module IOSUBNPAG

This routine searches the map register bit map in the adapter control block to allocate a set of contiguous map registers to a driver fork process.

INPUT TO ROUTINE

Registers	Contents
R3	Number of map registers to allocate (if entry is IOC\$ALOUBAMAPN)
R5	Address of the UCB
Fields	Contents
UCB\$W_BCNT	Transfer byte count (if entry is IOC\$ALOUBAMAP)
UCB\$W_BOFF	Byte offset in page (if entry is IOC\$ALOUBAMAP)
UCB\$L_CRB	Address of the CRB
CRB\$L_INTD+ VEC\$L_ADP	Address of the device's adapter control block
VEC\$V_MAPLOCK (in CRB\$L_INTD +VEC\$W_MAPREG)	Bit that indicates whether map registers are permanently allocated to this controller
ADP\$W_MRBITMAP	Determines which map registers are available

IPL at execution: caller's IPL

If map registers are already permanently allocated to the controller, this routine exits successfully without allocating any map registers. Otherwise, the routine searches the map register bit map for the required number of contiguous map registers, calls IOC\$ALTUBAMAP, and exits by issuing an RSB instruction.

OUTPUT FROM ROUTINE

Registers	Contents
R0	1 (success) 0 (insufficient contiguous map registers)
R1	Destroyed
R2	Destroyed

OPERATING SYSTEM ROUTINES

Fields	Contents
CRB\$L_INTD+ VEC\$B_NUMREG	Number of map registers allocated
CRB\$L_INTD+ VEC\$W_MAPREG	Starting map register number
ADP\$W_MRBITMAP	Bits for allocated map registers set to zero.
IPL at exit:	caller's IPL

IOC\$ALTUBAMAP in module IOSUBPAG

Clears or sets a field of bits in the UNIBUS adapter map register allocation bit map.

Register	Contents
R0	Alternation bit mask (zeros to clear bits, ones to set bits)
R1	Address of the channel request block
R2	Address of the adapter control block
R4	Number of the starting map register

Fields	Contents
CRB\$L_INTD+ VEC\$B_NUMREG	Number of map registers needed
IPL at execution:	caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R3	Destroyed
R4	Destroyed

Fields	Contents
ADP\$W_MRBITMAP	Bits describing available map registers
IPL at exit:	caller's IPL

OPERATING SYSTEM ROUTINES

IOC\$APPLYECC in module IOSUBRAMS

Disk drivers call this routine to apply an ECC correction to data transferred from a device into memory. This routine corrects the data by exclusive ORing a correction pattern from the unit control block. It also sets a UCB bit to indicate that an ECC correction has been made.

INPUT FROM ROUTINE

Register	Contents
R0	Number of bytes of data that have been transferred, not including the block to be corrected; this must be a multiple of 512 bytes
R5	Address of the unit control block

Fields	Contents
UCB\$W_BCNT	Length of transfer in bytes
UCB\$W_EC1	Starting bit number of the error burst
UCB\$W_EC2	Exclusive OR correction pattern
UCB\$L_SVPN	Address of the system page table entry of a page that is available for use by driver
UCB\$L_SVAPTE	System virtual address of the page table entry that maps the transfer

IPL at execution: caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed
R1	Destroyed
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

OPERATING SYSTEM ROUTINES

IOC\$CANCELIO in IOSUBNPAG.MAR

Device-independent cancel I/O routine that sets a cancel I/O bit in the unit control block if the I/O request packet being currently processed on the device originates from the current process on the specified channel and the unit is busy.

INPUT TO ROUTINE

Register	Contents
R2	Negative of the channel number
R3	Address of the I/O request packet
R4	Address of the current PCB
R5	Address of the unit control block
Fields	Contents
IRP\$L_PID	Process identification of the process that queued the I/O request
IRP\$W_CHAN	Negative of the channel number
PCB\$L_PID	Process identification of the process that requested cancellation
UCB\$V_BSY (in UCB\$W_STS)	Device busy flag
IPL at execution:	caller's IPL

OUTPUT FROM ROUTINE

Registers	Contents
---	---
Fields	Contents
UCB\$V_CANCEL (in UCB\$W_STS)	Set if I/O request should be cancelled
IPL at exit:	caller's IPL

IOC\$DIAGBUFILL in module IOSUBNPAG

Driver fork processes call this routine to fill a diagnostic buffer, if the QIO specifies such a buffer. This routine writes completion time and final error counters into buffer. It also calls the driver register dump routine to fill the remainder of buffer.

OPERATING SYSTEM ROUTINES

INPUT TO ROUTINE

Registers	Contents
R4	Address of the device's control/status register
R5	Address of the unit control block

Field	Contents
UCB\$L_IRP	Address of the current IRP
IRP\$V_DIAGBUF (in IRP\$W_STS)	Determines whether diagnostic buffer is present. If set, one exists.
IRP\$L_DIAGBUF	Address of the diagnostic buffer, if one is present
UCB\$B_ERTCNT	Final error retry count
UCB\$L_DDB	Address of the device data block
DDB\$L_DDT	Address of the driver dispatch table
DDT\$L_REGDUMP	Address of the driver register dump routine
EXE\$GQ_SYSTIME	Current system time (time at I/O request completion)
DDT\$L_REGDUMP	Address of the driver register dump routine

IPL at execution: caller's IPL

This routine saves the system time and final error count in the diagnostic buffer. It then calls the driver register dump routine, and exits with an RSB instruction.

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed
R1	Destroyed
R2	Address of the DDT
R3	Address of the I/O request packet
R4	Device CSR register
R5	Address of the unit control block

Fields	Contents
---	---

IPL at exit: caller's IPL

OPERATING SYSTEM ROUTINES

IOC\$INITIATE in module IOSUBNPAG

Starts a driver fork process to process an I/O request packet. This routine writes the I/O request packet address and I/O request packet transfer parameters into the unit control block. It also clears device status bits. If the QIO specifies a diagnostic buffer, this routine writes system time into the buffer. It also executes a JMP instruction to transfer control to the driver start I/O routine.

INPUT TO ROUTINE

Register	Contents
R3	Address of the I/O request packet
R5	Address of the unit control block
Fields	Contents
IRP\$L_SVAPTE	Address of system buffer (buffered I/O) or address of PTE that maps process buffer (direct I/O).
IRP\$W_BOFF	Byte offset of start of buffer
IRP\$W_SIZE	Size in bytes of transfer
IRP\$V_DIAGBUF (in IRP\$W_STS)	Determines whether a diagnostic buffer is present. This field is set if one exists.
IRP\$L_DIAGBUF	Address of the diagnostic buffer, if one is present
EXE\$GQ_SYSTIME	Current system time (when I/O processing began)
UCB\$L_DDB	Address of DDB
UCB\$L_DDT	Address of DDT
DDT\$L_START	Address of driver start I/O routine
IPL at execution:	caller's IPL

IOC\$INITIATE exits by jumping to the driver start entry specified in the driver dispatch table.

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed
R1	Destroyed

OPERATING SYSTEM ROUTINES

Fields	Contents
UCB\$L_IRP	Address of the start of the I/O request packet
UCB\$L_SVAPTE	IRP\$L_SVAPTE
UCB\$W_BOFF	IRP\$W_BOFF
UCB\$W_BCNT	IRP\$W_BCNT
UCB\$V_CANCEL (in UCB\$W_STS)	Zero
UCB\$V_TIMEOUT (in UCB\$W_STS)	Zero
diagnostic buffer	Current system time (first quadword)
IPL at exit:	caller's IPL

IOC\$IOPPOST in module IOCIOPPOST

Interrupt service routine that processes I/O request packets in an I/O postprocessing queue. This routine gains control when the processor grants a software interrupt at IPL\$IOPPOST. For each queue entry, it adjusts quota use and unlocks pages or deallocates write buffers. It queues a kernel mode AST to copy final I/O status to the IOSB, to copy buffered read data, and to deallocate read buffers. The AST kernel mode routine code is located in module IOCIOPPOST. The kernel mode AST routine queues a user mode AST if specified in the QIO. When the postprocessing queue is empty, IOC\$IOPPOST dismisses the interrupt.

INPUT TO ROUTINE

Registers	Contents
---	---
Fields	Contents
IOC\$GL_PSFL	Head of the I/O postprocessing queue. This routine uses this field to locate fields in the IRP.
IRP\$L_PID	Process identification of the process that initiated the I/O request. This routine uses this field to locate the PCB.

IPL at execution: IPL\$IOPPOST, IPL\$ASTDEL

IOC\$IOPPOST generates different results for direct and buffered I/O. For direct I/O, the routine unlocks the pages locked for the I/O request and sets the Queue I/O event flag. The pages unlocked include any pages defined in the IRP extension area descriptors (if an IRPE exists). For buffered I/O read functions, the routine copies the data from the system buffer to the process buffer, then releases the system buffer. It also sets a Queue I/O event flag, if one was requested.

OPERATING SYSTEM ROUTINES

For both direct and buffered I/O, IOC\$IOPOST performs the following functions:

- Copies the diagnostic buffer from system to process space and releases the system buffer
- Copies I/O completion status (if requested) from the I/O request packet to the process's I/O status block
- Queues an AST to the process, if one was requested
- Deallocates the IRP and any IRP extensions

Note that kernel mode ASTs handle much of the processing described above.

IOC\$LOADUBAMAP(A) in module LOADMREG

Driver fork processes for DMA transfers call this routine to load the UNIBUS map registers required by the current transfer with a page frame number, the data path number, possibly the byte offset bit, and possibly the longword access enable bit. This routine confirms that enough map registers have been allocated and sets the last map register invalid to stop a wild transfer.

INPUT TO ROUTINE

Registers	Contents
R5	Address of unit control block

The data path and map registers are already allocated.

Field	Contents
UCB\$W_BOFF	Offset to the first byte in the first page of the transfer
UCB\$W_BCNT	Number of bytes in the transfer
UCB\$L_CRB	Address of the controller's channel request block
CRB\$L_INTD+ VEC\$B_DATAPATH	Number of the data path to be allocated
VEC\$V_LWAE (in CRB\$L_INTD+ VEC\$B_DATAPATH)	Determines length of buffering. Set if longword buffering used (instead of quadword buffering)
CRB\$L_INTD+VEC\$L_NUMREG	Number of map registers allocated
CRB\$L_INTD+VEC\$L_ADP	Address of the adapter control block
UBA\$L_MAP	Address of the first UNIBUS map register
UCB\$L_SVAPTE	Address of the page table entry for the first page of the transfer

OPERATING SYSTEM ROUTINES

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed
R1	Destroyed
R2	Destroyed

Fields	Contents
Allocated map registers	Byte offset is set for entry IOC\$LOADUBAMAP (never set for IOC\$LOADUBAMAPA)

IPL at exit: caller's IPL

IOC\$PURGDATAP in module LIOSUB

Device drivers using buffered data paths call this subroutine after a data transfer. IOC\$PURGDATAP purges the UNIBUS adapter buffered data path as well as checking for and clearing purge errors.

INPUT TO ROUTINE

Registers	Contents
R5	Address of the UCB

Fields	Contents
---	---

IPL at execution: caller's IPL

This routine obtains the start of UNIBUS adapter register space using the following chain of pointers:

UCB\$L_CRB ==> CRB\$L_INTD+VEC\$L_ADP ==> ADP\$L_CSR

This routine extracts the caller's data path number (buffered or direct) from the channel request block. The routine then purges the data path and stores the contents of the data path register in R1. IOC\$PURGDATAP clears any purge errors in the data path register. It also sets the appropriate status in R0, computes the base of UNIBUS map registers, and writes the base into R2.

A purge of data path 0 is legal and a NOP; it always results in success status.

IOC\$PURGDATAP alters R0 through R3 but preserves all other registers.

OPERATING SYSTEM ROUTINES

OUTPUT FROM ROUTINE

Registers	Contents
R0	Low bit set (success) Low bit clear (failure)
R1	Contents of data path after purge (for register dump routine)
R2	Address of the start of UNIBUS map registers (for the register dump routine)
R3	Address of the CRB

Fields	Contents
---	---

IPL at exit: caller's IPL

IOC\$RELCHAN in module IOSUBNPAG

Driver fork processes call this routine to release controller data channels assigned to a device. If the channel wait queue contains waiting fork processes, this routine dequeues a process, assigns the channel to that process, restores R3 through R5, and reactivates the suspended process.

INPUT TO ROUTINE

Register	Contents
R5	Address of the unit control block

Fields	Contents
UCB\$L_CRB	Address of the channel request block
CRB\$L_LINK	Address of the secondary CRB
CRB\$V_BSY (in CRB\$B_MASK)	Set if the channel is busy
CRB\$L_INTD+VEC\$L_IDB	Address of the interrupt data block
IDB\$L_OWNER	Channel's owner UCB address
CRB\$L_WQFL	Head of the queue of waiting UCBS

IPL at execution: caller's IPL

OPERATING SYSTEM ROUTINES

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed
R1	Destroyed
R2	Destroyed
Fields	Contents
IDB\$ <u>L</u> _OWNER	Clear (if no driver is waiting for the channel)
CRB\$ <u>V</u> _BSY	Clear (if no driver is waiting for the channel)

IPL at exit: caller's IPL

IOC\$RELDATAP in module IOSUBNPAG

Driver fork processes call this routine to release a UNIBUS adapter buffered data path. This routine performs no operation if a data path is permanently allocated to the controller. If the data path wait queue contains waiting fork processes, it dequeues a process, allocates the data path to that process, restores R3 through R5, and reactivates the suspended process. This routine should not be called unless the driver owns a buffered data path.

INPUT TO ROUTINE

Registers	Contents
R5	Address of unit control block
Fields	Contents
UCB\$ <u>L</u> _CRB	Address of the channel request block
CRB\$ <u>L</u> _INTD+VEC\$ <u>L</u> _ADP	Address of the adapter control block
CRB\$ <u>L</u> _INTD+ VEC\$ <u>B</u> _DATAPATH	Data path specifier
VEC\$ <u>V</u> _PATHLOCK	Set to 1 to indicate that the data path is permanently allocated to the controller
ADP\$ <u>L</u> _DPQFL	Head of the adapter data path wait queue

IPL at execution: caller's IPL

If the bit map is corrupted, this routine signals a bugcheck with message code INCONSTATE. After IOC\$RELDATAP completes successfully, it exits with an RSB instruction.

OPERATING SYSTEM ROUTINES

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed
R1	Destroyed
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 in module IOSUBNPAG

Driver fork processes call this routine to release a set of UNIBUS adapter map registers. This routine performs no operation if map registers are permanently allocated to the controller. If the map register wait queue contains waiting fork processes, it dequeues a process and attempts to allocate the required set of map registers. If successful, it restores R3 through R5 and reactivates the suspended process. If not successful, it reinserts the fork process in the map register wait queue and dequeues the next process. This routine assumes that the caller is the current owner of the controller data channel.

INPUT TO ROUTINE

Registers	Contents
R5	Address of unit control block
Fields	Contents
UCB\$L_CRB	Address of the CRB
VEC\$V_MAPLOCK (in CRB\$L_INTD+ VEC\$W_MAPREG)	If set, indicates that map registers are permanently allocated to the controller
CRB\$L_INTD+VEC\$L_ADP	Address of the adapter control block
CRB\$L_INTD+ VEC\$W_MAPREG	Number of the starting map register
CRB\$L_INTD+ VEC\$B_NUMREG	Number of map registers to release
ADP\$L_MRQFL	Head of the queue of waiting drivers

IPL at execution: caller's IPL

IOC\$RELMAPREG calls IOC\$ALTUBAMAP and IOC\$ALOUBAMAP. It exits with an RSB instruction.

OPERATING SYSTEM ROUTINES

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed
R1	Destroyed
R2	Destroyed
Fields	Contents
ADP\$W_MRBITMAP	Map registers set to free
IPL at exit:	caller's IPL

IOC\$RELSCHAN in module IOSUBNPAG

This routine releases a secondary controller's data channel; that is, the MBA controller data channel. For more information, refer to Appendix F.

This routine has the same inputs and outputs as IOC\$RELCHAN.

IOC\$REQCOM in module IOSUBNPAG

Driver fork processes call this routine after a device I/O operation and all device-dependent processing of an I/O request are complete. This routine writes R0 and R1 into the I/O request packet status field. It then inserts the I/O request packet into the I/O postprocessing queue. If error logging is occurring, it writes final status into the error message buffer and calls ERL\$RELEASEMB. If the I/O request packet wait queue contains entries, it dequeues an I/O request packet and calls IOC\$INITIATE. Otherwise, it clears a unit control block busy status bit to indicate that the device is idle.

INPUT TO ROUTINE

Registers	Contents
R0	First longword of I/O status
R1	Second longword of I/O status
R5	Address of unit control block
Fields	Contents
UCB\$V_ERLOGIP (in UCB\$W_STS)	Set or clear. Determines whether error logging should be performed
UCB\$W_STS	Final device status
UCB\$B_ERTCNT	Final error counters
UCB\$L_EMB	Address of the error log message buffer
UCB\$L_IRP	Address of the IRP

IPL at execution: caller's IPL

OPERATING SYSTEM ROUTINES

This routine places the I/O request packet in the queue headed by IOC\$GL_PSBL. If UCB\$L_IOQEL has a packet queued to it, IOC\$REOCOM sends the packet to IOC\$INITIATE. This routine exits by branching to IOC\$RELCHAN.

OUTPUT FROM ROUTINE

Registers	Contents
R2	Destroyed
R3	Destroyed

If IOC\$INITIATE is called, other registers will be destroyed.

Fields	Contents
IRP\$L_MEDIA	I/O status (R0)
IRP\$L_MEDIA+4	I/O status (R1)
EMB\$Q_IOSB	I/O status (R0 and R1)
UCB\$L_OPCNT	Incremented by 1
EMB\$B_ERTCNT	UCB\$B_ERTCNT
EMB\$B_ERTCNT+1	UCB\$B_ERRCNT
EMB\$W_DV_STS	UCB\$W_STS
UCB\$V_BSY (in UCB\$W_STS)	Clear (if no more packets in queue)

IPL at exit: caller's IPL

IOC\$REQDATAP(NW) in module IOSUBNPAG

Driver fork processes call this routine to request a UNIBUS adapter buffered data path for a DMA transfer. This routine performs no operation if a data path is permanently allocated to the controller. This routine locates a free data path and writes the data path number in the CRB. If no data paths are free, it saves R3 and R4 in the UCB fork block, inserts the fork block address in a data path wait queue, and suspends the driver fork process.

INPUT TO ROUTINE

Register	Contents
R5	Address of unit control block
0(SP)	Caller's return address
4(SP)	Return address of the caller's caller

OPERATING SYSTEM ROUTINES

Fields	Contents
UCB\$L_CRB	Address of the channel request block
VEC\$V_PATHLOCK (in CRB\$L_INTD+ VEC\$B_DATAPATH)	If set, indicates that the data path already is allocated
CRB\$L_INTD+VEC\$L_ADP	Address of the adapter control block
ADP\$W_DPBITMAP	Indicates what data paths are available

IPL at execution: caller's IPL

If IOC\$REQDATAP cannot allocate a data path, and NW is not specified, the routine saves process context by placing the contents of R3, R4 and the PC in the UCB fork block and placing R5 in the data path wait queue (ADP\$L_DPQBL). If, however, NW is specified, the routine does not suspend the process to wait for the data path.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$ NORMAL (success) 0 (Failure)

Fields	Contents
CRB\$L_INTD+ VEC\$B_DATAPATH	Data path number
ADP\$W_DPBITMAP	Bit for allocated data path clear

IPL at exit: caller's IPL

IOC\$REQMAPREG in module IOSUBNPAG

Driver fork processes call this routine to request a set of UNIBUS adapter map registers for a DMA transfer. This routine performs no operation if map registers are permanently allocated to the controller. This routine locates the required number of map registers and writes the number of registers and the number of the first register into the CRB. If sufficient map registers are not available, it saves R3 and R4 in the UCB fork block, inserts the fork block address in a map register wait queue, and suspends the driver fork process.

INPUT TO ROUTINE

Registers	Contents
R5	Address of unit control block
0(SP)	Return address of caller
4(SP)	Return address of the caller's caller

OPERATING SYSTEM ROUTINES

Fields	Contents
UCB\$W_BCNT	Transfer byte count
UCB\$W_BOFF	Byte offset into page of start of buffer
UCB\$L_CRB	Address of CRB
CRB\$L_INTD+ VEC\$L_ADP	Address of the adapter control block
VEC\$V_MAPLOCK (in CRB\$L_INTD+ VEC\$W_MAPREG)	Determines status of map lock bit
ADP\$W_MRBITMAP	Adapter map register allocation bit map

IPL at execution: caller's IPL

If registers are not available, this routine suspends the process by saving the following context:

- R3 and R4 are saved in UCB\$L_FR3 and UCB\$L_FR4, respectively.
- PC is saved in UCB\$L_FPC.
- R5 is saved in ADP\$L_MRQBL, which is the adapter's map register wait queue.

OUTPUT FROM ROUTINE

Registers	Contents
R0	SS\$_NORMAL (success)
R1	Destroyed
R2	Destroyed

Fields	Contents
CRB\$L_INTD+ VEC\$W_MAPREG	Starting map register number of those allocated
CRB\$L_INTD+ VEC\$B_NUMREG	Number of map registers allocated
ADP\$W_MRBITMAP	Allocated map registers

IPL after execution: caller's IPL

IOC\$REQPCHANH in IOSUBNPAG.MAR

Driver fork processes call this routine to request a channel on the primary controller with high priority. If the controller data channel is idle, this routine writes the UCB address in the interrupt data block and returns the CSR address in R4. Otherwise, it saves R3 in the UCB fork block, inserts the fork block address at the front of the channel wait queue, and suspends the driver fork process.

OPERATING SYSTEM ROUTINES

INPUT TO ROUTINE

Registers	Contents
R5	Address of unit control block
0(SP)	Return address of the caller
4(SP)	Return address of the caller's caller
Fields	Contents
UCB\$L_CRB	Address of the channel request block
CRB\$L_LINK	Address of the secondary channel request block
CRB\$L_INTD+VEC\$L_IDB	Interrupt data block address
CRB\$V_BSY in CRB\$B_MASK	Set or clear. If set, indicates that the channel is busy
IDB\$L_CSR	Address of device CSR

IPL at execution: caller's IPL

If the channel is busy, this routine saves driver context by storing the contents of R3 and R4 in UCB\$L_FR3 and UCB\$L_FR4, respectively, storing 0(SP) in UCB\$L_FPC and placing the contents of R5 in the CRB wait queue (CRB\$W_WQFL).

IDL\$REQPCHANH exits by issuing an RSB instruction.

OUTPUT FROM ROUTINE

Registers	Contents
R0	Destroyed
R1	Destroyed
R2	Destroyed
R4	IDB\$L_CBR
Fields	Contents
IDB\$L_OWNER	R5

IPL at exit: caller's IPL

IOC\$REQPCHANL in module IOSUBNPAG

Driver fork processes call this routine to request a channel on the primary controller with low priority. This routine performs in the same manner as IDL\$REQPCHANH, except that, should driver have to wait for the channel, IOC\$REQPCHANL places the UCB at the end of the channel wait queue.

OPERATING SYSTEM ROUTINES

IOC\$REQSCHANH in module IOSUBNPAG

Driver fork processes call this routine to request a channel on the secondary controller with high priority.

The input to and output from this routine are the same as for IOC\$REQPCHANH, except that the secondary controller data channel is assigned.

IOC\$REQSCHANL in module IOSUBNPAG

Driver fork processes call this routine to request a channel on the secondary controller with low priority.

The input to and output from this routine are the same as for IOC\$REQPCHANH, except that the secondary controller data channel is assigned.

IOC\$RETURN in module IOSUBNPAG

This routine merely returns by issuing an RSB instruction. It has no input requirements and produces no output.

IOC\$WFIKPCH in module IOSUBNPAG

Driver fork processes call this routine to suspend driver processing to wait for an interrupt or device timeout and still retain the controller data channel. This routine saves R3, R4, and the driver's return PC from top of stack in the UCB fork block. It sets UCB bits to indicate that an interrupt or a timeout is expected and sets the timeout time in the unit control block. It clears the UCB bit that indicates that the unit is timed out and lowers IPL back to the IPL saved on top of stack. Then, it returns to the caller of the driver fork process.

The two bytes following the JSB to IOC\$WFIKPCH contain the relative offset to the timeout routine.

INPUT TO ROUTINE

Register	Contents
R5	Address of unit control block
0(SP)	Address following the JSB to IOC\$WFIKPCH
4(SP)	Timeout value in seconds
8(SP)	IPL to which to lower before returning to the caller's caller
12(SP)	Return address of the caller's caller
Field	Contents
EXE\$GL_ABSTIM	Absolute time. Used to compute time at which device times out

IPL at execution: Fork a device IPL (caller's IPL)

OPERATING SYSTEM ROUTINES

This routine removes 0(SP) through 11(SP) from the stack explicitly and 12(SP) through 15(SP) implicitly by exiting with an RSB instruction, which returns to the caller's caller.

OUTPUT FROM ROUTINE

Registers	Contents
---	---
Fields	Contents
UCB\$L_DUETIM	Sum of timeout value and EXE\$GL_ABSTIM
UCB\$V_INT	Set to indicate that interrupts are expected on the device
UCB\$V_TIM	Set to indicate that timeouts are expected on the device
UCB\$V_TIMEOUT	Cleared to indicate that unit is not timed out
UCB\$L_FR3	R3
UCB\$L_FR4	R4
UCB\$FPC	0(SP)+2

IPL at exit: IPL specified in 8(SP)

IOC\$WFIRLCH in IOSUBNPAG

Driver fork processes call this routine to suspend driver processing to wait for an interrupt or device timeout first releasing the controller data channel.

The input to and output from this routine are the same as IOC\$WFIKPCB except that IOC\$WFIRLCH exits to IOC\$RELCHAN, which releases the controller data channel.

APPENDIX D

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

This appendix contains the source listing of a driver for an analog-to-digital converter.

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

.TITLE ADDRIVER - VAX/VMS AD11-K DRIVER
.IDENT 'V02-000'

```
;  
;*****  
;*  
;* Copyright (c) 1978,1979,1980 *  
;* by DIGITAL Equipment Corporation, Maynard, Mass. *  
;*  
;* This software is furnished under a license and may be used and copied *  
;* only in accordance with the terms of such license and with the *  
;* inclusion of the above copyright notice. This software or any other *  
;* copies thereof may not be provided or otherwise made available to any *  
;* other person. No title to and ownership of the software is hereby *  
;* transferred. *  
;*  
;* The information in this software is subject to change without notice *  
;* and should not be construed as a commitment by DIGITAL Equipment *  
;* Corporation. *  
;*  
;* DIGITAL assumes no responsibility for the use or reliability of its *  
;* software on equipment which is not supplied by DIGITAL. *  
;*  
;*****  
;  
;+  
; FACILITY:  
; VAX/VMS AD11-K I/O DRIVER  
; ABSTRACT:  
; DEVICE TABLES AND DRIVER CODE FOR THE AD11-K ANALOGUE  
; TO DIGITAL CONVERTER WITH OPTIONAL AM11-K MULTIPLEXER.  
; AUTHOR:  
; S. PROGRAMMER, SEPTEMBER 1978.  
; MODIFIED BY:  
;--
```

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

.SBTTL FUNCTIONAL DESCRIPTION OF DRIVER

```

;+
; THE DRIVER SUPPORTS A/D SAMPLING ON GROUPS OF CHANNELS VIA QIO
; READ REQUESTS. NO EXTERNALLY TRIGGERED SAMPLING (I.E., CLOCK
; OVERFLOW OR SCHMITT TRIGGER) IS SUPPORTED. THE AM11-K MULTIPLEXER
; MAY BE PRESENT, BUT NO AUTOMATIC RANGING AMPLIFICATION IS
; DONE AT DRIVER LEVEL. THE BUILT-IN DAC MAY BE USED FOR TESTING VIA
; A LOOPBACK QIO FUNCTION DEFINED ESPECIALLY FOR THIS DEVICE.
;
; THE QIO FUNCTIONS AVAILABLE ARE:
;
; IO$_READVBLK          -READ VIRTUAL BLOCK
; IO$_READLBLK          -READ LOGICAL BLOCK
; IO$_READPBLK          -READ PHYSICAL BLOCK=IO$_LOOPBACK
; IO$_LOOPBACK          -WRITE DAC, READ RESULTS; REQUIRES
;                        PHYSICAL I/O PRIVILEGE
;
; THE STANDARD QIO PARAMETERS ARE:
;
; P1=BUFFER ADDRESS
; P2=BUFFER BYTE COUNT
; P3=SPECIFIER OF CHANNELS TO SAMPLE:
;     BIT 0-7/INITIAL CHANNEL # (0-63)
;     BIT 8-15/TOTAL # OF CHANNELS TO SAMPLE (1-64)
;     BIT 16-23/CHANNEL INCREMENT (0-63)
;     BIT 24-31/IGNORED
; P4=DAC VALUE, USED FOR LOOPBACK ONLY:
;     BIT 0-7/8 BIT DAC VALUE
;     BIT 8-31/IGNORED
; P5,P6 ARE NOT USED
;
; IN ADDITION TO THE STANDARD STATUS CODES THAT CAN BE RETURNED FOR
; A QIO, THE FOLLOWING DEVICE-SPECIFIC I/O STATUS VALUES ARE DEFINED:
;
; SS$_DATAOVERUN        -ERROR BIT SET IN CSR; SAMPLING ABORTED
;                        WITH LAST GOOD SAMPLE IN BUFFER
; SS$_BADPARAM          -INVALID CHANNEL SPECIFIER; NO SAMPLES TAKEN
; SS$_BUFFEROVF         -USER BUFFER OVERRUN; AS MANY CHANNELS AS WILL
;                        FIT ARE SAMPLED
;
; THE SAMPLES ARE RETURNED IN THE CALLER'S BUFFER PACKED ONE SAMPLE
; PER WORD, BITS 0-11. THE BYTE COUNT RETURNED IN THE SECOND WORD OF
; THE I/O STATUS BLOCK ALWAYS REFLECTS THE # OF BYTES ACTUALLY FILLED
; WITH SAMPLE DATA. THE NUMBER OF SAMPLES IS ONE HALF THE RETURNED
; BYTE COUNT.
;
; EXAMPLE: SWEEP THROUGH 32 INPUTS CONNECTED IN DIFFERENTIAL MODE
;          (AD11-K AND AM11-K):
;
; SWEEPBUF:      .BLKW  32
; NUMINPUT:      .LONG  32
; CHANSPEC:      .BYTE  0,32,2                ;START WITH CHANNEL 0;
;                                                    ; SAMPLE CHANNELS 0,2,4,...,62
;
;                $QIO_S  CHAN=X,FUNC=IO$_READVBLK,-
;                P1=SWEEPBUF,P2=NUMINPUT,P3=CHANSPEC
;-

```


SAMPLE DRIVER FOR AN A-TO-D CONVERTER

.SBTTL MACRO LIBRARY CALLS

;
; EXTERNAL SYMBOLS (LIB/LIB):
;

\$CRBDEF	;CHANNEL REQUEST BLOCK
\$DDBDEF	;DEVICE DATA BLOCK
\$IDBDEF	;INTERRUPT DATA BLOCK
\$IODEF	;I/O FUNCTION CODES
\$IPLDEF	;HARDWARE IP DEFINITIONS
\$IRPDEF	;I/O REQUEST PACKET
\$UCBDEF	;UNIT CONTROL BLOCK
\$VECDEF	;INTERRUPT VECTOR BLOCK
\$JIBDEF	;JOB INFORMATION BLOCK

;
; USER DEFINED EXTERNAL SYMBOLS ARE CONTAINED IN A USER LIBRARY .
; THE CONTENTS OF THIS LIBRARY CAN BE MERGED WITH THE SYSTEM LIBRARY
; TO ALLOW USER PROGRAMS TO USE EXTENDED FUNCTION CODES WITHOUT HAVING
; TO DEFINE THEM LOCALLY.
; THIS DRIVER MUST BE ASSEMBLED WITH A USER LIBRARY TO DEFINE \$XIODEF.
;

\$XIODEF	;EXTENDED QIO FUNCTIONS.THIS MACRO ;CONTAINS THE DEFINITIONS FOR ;IO\$_LOOPBACK
----------	---

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

.SBTTL LOCAL DEFINITIONS

```

;
; LOCAL DEFINITIONS:
;
; QIO ARGUMENT LIST OFFSETS:
;

P1=0           ;FIRST,
P2=4           ; SECOND,
P3=8           ; THIRD,
P4=12          ; FOURTH,
P5=16          ; FIFTH,
P6=20          ; AND SIXTH PARAMETERS

;
; DEVICE PARAMETERS:
;

DAC_TIMER=20   ;20 USEC TIMER FOR DAC SETTLE
MAX_INLCHN=63  ;MAXIMUM INITIAL CHANNEL #,
MAX_NUMCHN=64  ; NUMBER OF CHANNELS,
MAX_INCCHN=63  ; AND CHANNEL INCREMENT
ADC_TIMER=2    ;A/D CONVERSION TIMEOUT=2 SEC

;
; DEVICE REGISTER DEFINITIONS:
;

        $DEFINI AD

$DEF     AD_CSR   .BLKW   1           ;CONTROL/STATUS REGISTER

        _VIELD  AD_CSR,0,<-         ;DEFINE CSR FIELDS: AD_CSR_M_XXX
        <GO,,M>,-                     ; START A/D CONVERSION
        <,3>,-                          ; 3 UNUSED BITS
        <EXT,,M>,-                      ; EXTERNAL START ENABLE
        <COV,,M>,-                      ; CLOCK OVERFLOW ENABLE
        <IE,,M>,-                       ; INTERRUPT ENABLE
        <DON,,M>,-                      ; CONVERSION DONE FLAG
        <MUX,6,M>,-                    ; 6 BIT MUX CHANNEL #
        <,1>,-                          ; BIT 14 IS UNUSED
        <ERR,,M>,-                     ; ERROR FLAG
        >                               ;END OF CSR FIELDS
$DEF     AD_DBR   .BLKW   1           ;A/D DATA BUFFER REGISTER
.=.-2
$DEF     AD_DAC   .BLKW   1           ;DAC DATA BUFFER REF

        $DEFEND AD                   ;END OF A/D REGISTER DEFNS

;
; DEVICE DEPENDENT UCB EXTENSIONS:
;

        $DEFINI UCB

.=UCB$K_LENGTH           ;STEP TO END OF STANDARD UCB
                          ;NOTE: NEXT 4 BYTES ASSUMED
                          ; ADJACENT
$DEF     UCB$B_AD_CURCHN .BLKB   1   ;CURRENT MUX CHANNEL #
$DEF     UCB$B_AD_NUMCHN .BLKB   1   ;# CHANNELS LEFT TO SAMPLE
$DEF     UCB$B_AD_INCCHN .BLKB   1   ;CHANNEL INCREMENT
                          ;SPARE BYTE
$DEF     UCB$W_AD_CSR    .BLKW   1   ;SAVED CSR

```

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

```
        _VIELD  UCB$W_CSR,1,<-          ;BORROW UNUSED CSR BIT
        <BFO,,M>,-                      ; FOR USER BUFFER OVERRUN
        >
UCB$K_ADLENGTH=.                        ;LENGTH OF A/D UCB
        $DEFEND UCB                      ;END OF UCB EXTENSIONS
;
; A/D DRIVER USE OF TEMPORARY IRP STORAGE:
;
IRP$L_CHSPEC=IRP$L_MEDIA                 ;CHANNEL SPECIFIER(P3)
IRP$L_DACVAL=IRP$L_MEDIA+4              ;OPTIONAL DAC VALUE(P4)
```

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

.SBTTL DRIVER PROLOGUE AND DISPATCH TABLES

```
;
; DRIVER PROLOGUE TABLE:
;
```

```
DPTAB - ;DEFINE DRIVER PROLOGUE TABLE:
      END=AD_END,- ; END OF DRIVER,
      ADAPTER=UBA,- ; UNIBUS ADAPTER,
      UCBSIZE=UCBSK_ADLENGTH,-; SIZE OF A/D UCB,
      NAME=ADDRIVER ; DRIVER NAME

DPT_STORE INIT ;VALUES TO BE SET ON LOAD
DPT_STORE UCB,UCBSB_FIPL,B,8 ;DEVICE FORK IPL
DPT_STORE UCB,UCBSB_DIPL,B,22 ;AD11 HARDWARE IPL
DPT_STORE UCB,UCBSL_DEVCHAR,L,- ;AD11 DEVICE CHARACTERISTICS:
      <DEVSM_AVL- ; AVAILABLE,
      !DEVSM_IDV- ; INPUT DEVICE,
      !DEVSM_RTM> ; REALTIME DEVICE

DPT_STORE REINIT ;VALUES TO SET ON RELOAD
DPT_STORE CRB,CRBSL_INTD+4,D,- ;INTERRUPT SERVICE ADDR
      AD_INTERRUPT

DPT_STORE CRB,- ;ADDR OF CONTROLLER
      CRBSL_INTD+VECSL_INITIAL,- ; INITIALIZATION
      D,AD_CTLINIT

DPT_STORE CRB,- ;ADDR OF UNIT
      CRBSL_INTD+VECSL_UNITINIT,- ; INITIALIZATION
      D,AD_UNITINIT

DPT_STORE DDB,DDBSL_DDT,D,- ;ADDR OF DRIVER
      ADSDDT ; DISPATCH TABLE

DPT_STORE END ;END DRIVER PROLOGUE
```

```
;
; DRIVER DISPATCH TABLE:
;
```

```
DDTAB - ;DDT CREATION MACRO
      DEVNAM=AD,- ;NAME OF DEVICE
      START=AD_STARTIO,- ;ADDR OF START I/O ROUTINE
      FUNCTB=AD_FUNCTABLE ;ADDR OF FDT
```

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

.SBTTL AD11-K FUNCTION DECISION TABLE

```
;  
; AD11 FUNCTION DECISION TABLE:  
;  
AD_FUNC_TABLE:                                ;FUNCTION DECISION TABLE START  
    FUNCTAB , -                                ;LEGAL FUNCTIONS:  
        <LOOPBACK, -                            ; LOOPBACK READ FROM DAC  
        READPBLK, -                            ; READ PHYSICAL BLOCK  
        READLBLK, -                            ; READ LOGICAL BLOCK  
        READVBLK>                             ; READ VIRTUAL BLOCK  
    FUNCTAB , -                                ;BUFFERED I/O FUNCTIONS:  
        <LOOPBACK, -                            ; LOOPBACK READ FROM DAC  
        READPBLK, -                            ; READ PHYSICAL BLOCK  
        READLBLK, -                            ; READ LOGICAL BLOCK  
        READVBLK>                             ; READ VIRTUAL BLOCK  
    FUNCTAB -                                  ;PREPROCESSING ROUTINES:  
        AD_READ, -                             ;CALL SINGLE PREPROCESSOR FOR:  
        <LOOPBACK, -                            ; LOOPBACK READ FROM DAC  
        READPBLK, -                            ; READ PHYSICAL BLOCK  
        READLBLK, -                            ; READ LOGICAL BLOCK  
        READVBLK>                             ; AND READ VIRTUAL BLOCK
```

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

```

.SBTTL  AD_READ:          READ FUNCTION PROCESSING
;+
; AD_READ - READ FUNCTION PREPROCESSING
;
; THIS ROUTINE IS CALLED FROM THE FUNCTION DECISION TABLE DISPATCHER
; TO PROCESS A READ PHYSICAL, READ LOGICAL, READ VIRTUAL, OR LOOPBACK
; I/O FUNCTION.
;
; AD_READ FIRST VERIFIES THE CALLER'S PARAMETERS, TERMINATING THE
; REQUEST WITH IMMEDIATE SUCCESS OR ERROR IF NECESSARY.  P3 AND
; P4 ARE STORED IN THE IRP.  A SYSTEM BUFFER IS ALLOCATED AND
; ITS ADDRESS IS SAVED IN THE IRP.  THE CALLER'S QUOTA IS UPDATED,
; AND THE READ REQUEST IS QUEUED TO THE DRIVER FOR STARTUP.
;
; INPUTS:
;
;      R0,R1,R2 = SCRATCH
;      R3 = IRP ADDRESS
;      R4 = ADDR OF PCB FOR CURRENT PROCESS
;      R5 = DEVICE UCB ADDRESS
;      R6 = ADDRESS OF CCB
;      R7 = I/O FUNCTION CODE
;      R8 = FDT DISPATCH ADDR
;      R9-R11 = SCRATCH
;      AP = ADDR OF FUNCTION PARAMETER LIST
;
; OUTPUTS:
;
;      R0,R1,R2 = DESTROYED
;      R3-R11,AP = PRESERVED
;      IRP$CHSPEC(R3) = CHANNEL SPECIFIER (P3)
;      IRP$DACVAL(R3) = OPTIONAL DAC VALUE (P4)
;      IRP$SVAPTE(R3) = ADDR OF ALLOCATED SYSTEM BUFFER
;      IRP$WBOFF(R3) = REQUESTED BYTE COUNT
;
;      SYSTEM BUFFER:
;          LONGWD 0/ADDR OF START OF DATA=BUFF ADDR+12
;          LONGWD 1/ADDR OF USER BUFFER
;          LONGWD 2/DATA STRUCTURE BOOKKEEPING
;-

.ENABL  LSB

AD_READ:
MOVZWL  P2(AP),R1          ;READ FUNCTION PREPROCESSING
BEQL    10$               ;GET USER BYTE COUNT
                          ;BRANCH IF READ OF 0 BYTES
                          ; (=INSTANT SUCCESS)
MOVZWL  #SS$BADPARAM,R0  ;ASSUME CHANNEL SPEC ERROR
MOVAL   P3(AP),R2        ;GET ADDR OF CHANNEL SPEC
CMPB    (R2)+,#MAX_INLCHN ;INITIAL CHAN # TOO LARGE?
BGTRU   20$              ;BRANCH IF SO
TSTB    (R2)              ;# CHANNELS = 0?
BEQL    10$              ;BRANCH IF SO (SUCCESS)
CMPB    (R2)+,#MAX_NUMCHN ;# CHANNELS TO SAMPLE TOO LARGE?
BGTRU   20$              ;BRANCH IF SO
CMPB    (R2),#MAX_INCCHN ;CHANNEL INCREMENT TOO LARGE?
BGTRU   20$              ;BRANCH IF SO
MOVQ    P3(AP),IRP$CHSPEC(R3) ;STORE P3 AND P4 (OPTIONAL DAC)
                          ; IN IRP UNTIL REQUEST EXECUTION
MOVL    P1(AP),R0        ;GET ADDR OF USER BUFFER
JSB     G^EXE$READCHK    ;VERIFY THAT CALLER HAS
                          ; WRITE ACCESS TO BUFFER
PUSHR   #^M<R0,R3>      ;SAVE USER BUFF ADDR, IRP ADDR
ADDL    #12,R1           ;ADD 12 BYTES TO REQUESTED BUFF

```

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

```

; SIZE FOR BUFF HEADER
JSB      G^EXE$BUFFRQUOTA      ;VERIFY BUFFER SPACE LEFT
; IN CALLER'S QUOTA
BLBC     R0,30$                ;BRANCH IF INSUFFICIENT QUOTA
JSB      G^EXE$ALLOCBUF        ;ALLOCATE A SYSTEM BUFFER
BLBC     R0,30$                ;BRANCH IF NONE AVAILABLE
POPR     #^M<R0,R3>            ;RESTORE USER BUFFER, IRP ADDR
MOVL     R2,IRP$SL_SVAPTE(R3)  ;SAVE ADDR OF SYSTEM BUFFER
MOVW     R1,IRP$W_BOFF(R3)     ; AND REQUESTED BYTE COUNT
MOVZWL   R1,R1                 ;CONVERT TO LONGWORD
MOVL     PCB$SL_JIB(R4),R4     ;GET JOB INFORMATION BLOCK ADDRESS
SUBL     R1,JIB$SL_BYTCNT(R4)  ;DEDUCT REQUESTED BYTE COUNT
; FROM PROCESS' QUOTA
MOVAB    12(R2),(R2)+          ;SAVE ADDR OF START OF USER DATA
; IN 1ST LONGWD OF SYSTEM BUFFER
MOVL     R0,(R2)              ;SAVE USER BUFFER ADDR IN
; 2ND LONGWD
JMP      G^EXE$QIODRVPKT      ;QUEUE I/O PKT TO DRIVER

;
; COME HERE IF USER REQUESTED READ OF 0 BYTES OR 0 CHANNELS.
; THIS IS ALWAYS SUCCESSFUL AND DOES NO DEVICE I/O:
;
10$:     MOVZWL #SS$ NORMAL,R0   ;SET NORMAL COMPLETION STATUS
20$:     JMP      G^EXE$FINISHIOC ;COMPLETE I/O REQUEST

;
; COME HERE TO ABORT I/O REQUEST WITH EXCEPTION STATUS IN R0:
;
30$:     POPR     #^M<R2,R3>    ;CLEAR BUFFER ADDR; RESTORE IRP
; ADDR
JMP      G^EXE$ABORTIO        ;COMPLETE I/O REQUEST

.DSABL   LSB

```

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

```

.SBTTL  AD_STARTIO:      PERFORM A/D CONVERSIONS
;+
; AD_STARTIO - START I/O OPERATION ON AD11-K A/D CONVERTER.
;
; THIS ROUTINE IS ENTERED WHEN THE ASSOCIATED UNIT IS IDLE AND A
; PACKET IS AVAILABLE FOR PROCESSING.
;
; TO PREPARE FOR SAMPLING, AD_STARTIO PERFORMS THESE STEPS:
;
; 1. SET UP UCB WITH CHANNEL SPECIFIER AND ADDRESS IN SYSTEM
;    BUFFER TO HOLD FIRST SAMPLE.
; 2. IF LOOPBACK WAS SPECIFIED, THE DAC IS SET WITH THE CALLER-
;    SPECIFIED VALUE.
;
; THE DRIVER THEN LOOPS FROM AD NXTSAMPLE TO AD ENDSAMPLE
; COLLECTING SAMPLES UNTIL ALL SAMPLES HAVE BEEN COLLECTED,
; OR AN ERROR OCCURS. AN INTERRUPT IS RECEIVED FOR EACH SAMPLE,
; BUT, TO SAVE TIME, THE DRIVER NEVER FORKS UNTIL TIME TO
; COMPLETE THE I/O REQUEST.
;
; INPUTS:
;
;     R3 = ADDR OF IRP
;     R5 = ADDR OF DEVICE UNIT UCB
;
; OUTPUTS:
;
;     R0,R1,R2 = DESTROYED
;     OTHER REGISTERS ARE PRESERVED
;-

.ENABL  LSB

AD_STARTIO:
    MOVL    IRP$L_CHSPEC(R3),-      ;START NEXT QIO
    UCB$B_AD_CURCHN(R5)           ;COPY CHANNEL SPEC FROM
    MOVL    @IRP$L_SVAPTE(R3),-   ; IRP TO UCB
    UCB$L_SVAPTE(R5)             ;SET ADDR OF START DATA
    MOVL    UCB$L_CRB(R5),R4        ; IN UCB
    MOVL    @CRB$L_INTD+VEC$L_IDB(R4),R4 ; GET CRB ADDRESS,
    BICB3   #^<IO$M FCODE>,-        ; THEN CSR ADDRESS
    IRP$W_FUNC(R3),R0              ; GET THE I/O
    CMPB    R0,#IO$ LOOPBACK        ; FUNCTION CODE
    BNEQ    AD_NXTSAMPLE            ; LOOPBACK?
    MOVZBW  IRP$L_DACVAL(R3),-      ; LOOPBACK?
    AD_DAC(R4)                     ; BRANCH IF NOT
    MFPR    S^#PR$ ICR,R1           ; SET DAC VALUE IN
    ADDL    #DAC_TIMER,R1          ; DAC BUFFER REGISTER
    BLSS    10$                    ; GET CURRENT INTERVAL COUNTER (USEC)
    MOVAV  -10000(R1),R1            ; +DAC SETTLE TIME IN USEC
    10$:   MFPR    S^#PR$ ICR,R0     ; BRANCH IF COUNTER DOESN'T
    CMPL    R0,R1                   ; OVERFLOW
    BLSS    10$                     ; ELSE CALCULATE COUNTER
    10$:   MFPR    S^#PR$ ICR,R0     ; FOR NEXT INTERVAL
    CMPL    R0,R1                   ; READ INTERVAL COUNTER NOW
    BLSS    10$                     ; REACHED SETTLE TIME YET?
    10$:   BLSS    10$              ; BRANCH IF NOT

AD_NXTSAMPLE:
    MOVZBW  #AD_CSR_M_IE!AD_CSR_M_GO,R0 ; START NEXT SAMPLE
    INSV    UCB$B_AD_CURCHN(R5),-   ; SET INTERRUPT ENABLE AND
    #8,#6,R0 ; SET MUX CHAN #
    DSBINT  #UCB$V_POWER,-          ; START A/D CONVERSION
    BBSC    UCB$W_STS(R5),AD_POWERFAIL ; DISABLE INTERRUPTS (IPL=IPL$POWER)
    ; AND CLEAR POWER FAIL SIGNAL

```


SAMPLE DRIVER FOR AN A-TO-D CONVERTER

```

MOVW    R0,AD_CSR(R4)           ;SET CSR
WFIKPCB AD_TIMEOUT,#ADC_TIMER  ;WAIT FOR INTERRUPT, OR TIMEOUT
MOVW    AD_CSR(R4),UCB$W_AD_CSR(R5) ;SAVE CSR IN UCB
BLSS    AD_CSRERROR             ;BRANCH IF ERROR
MOVW    AD_DBR(R4),@UCB$L_SVAPTE(R5) ;COPY A/D VALUE INTO
                                           ; SYSTEM BUFFER
ADDL    #2,UCB$L_SVAPTE(R5)      ;STEP BUFFER POINTER
SUBL    #2,UCB$W_BCNT(R5)       ;DECREASE # BYTES LEFT IN REQUEST
DECB    UCB$B_AD_NUMCHN(R5)     ;DECR # CHANNELS LEFT TO SAMPLE
BEQL    AD_DONE                 ;BRANCH IF NONE
CMPW    UCB$W_BCNT(R5),#2       ;AT LEAST 2 BYTES LEFT IN BUFFER?
BLSSU   AD_BUFFEROVF           ;BRANCH IF NOT
BICW    #UCB$W_CSR_M_BF0,-      ;ELSE CLEAR BUFFER OVERRUN
        UCB$W_AD_CSR(R5)       ; BIT IN CSR COPY
ADDB    UCB$B_AD_INCCHN(R5),-   ;NEXT CHANNEL # =
        UCB$B_AD_CURCHN(R5)   ; CURRENT CHANNEL+INCREMENT
BICB    #^C<MAX_NUMCHN-1>,-    ; MODULO MAXIMUM
        UCB$B_AD_CURCHN(R5)   ; CHANNEL #

AD_ENDSAMPLE:                    ;THIS SAMPLE COMPLETE
BRB     AD_NXTSAMPLE            ;GO START NEXT SAMPLE

.DSABL  LSB

```

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

.SBTTL - I/O REQUEST COMPLETION

```

;
; COME HERE TO COMPLETE I/O REQUEST WITH NORMAL OR ERROR STATUS.
;
; USER BUFFER OVERRUN, I.E., NO MORE SAMPLES CAN BE COLLECTED:
;
        .ENABL  LSB

AD_BUFFEROVF:
        BISW      #UCB$W_CSR_M_BFO,-      ;SET BUFFER OVERRUN BIT
                UCB$W_AD_CSR(R5)        ; IN CSR COPY
;
; CSR ERROR BIT WAS SET:
;

AD_CSRERROR:
        TSTW      AD_DBR(R4)              ;CLEAR ERROR
        BRB       AD_DONE                 ;JOIN COMMON I/O COMPLETION

;
; DEVICE TIMED OUT DUE TO EITHER A REAL TIMEOUT OR TO A
; POWER FAILURE.  BOTH CAUSES ARE HANDLED THE SAME.
;

AD_TIMEOUT:
        CLRW      AD_CSR(R4)              ;CLEAR INTERRUPT ENABLE,
        TSTW      AD_DBR(R4)              ; PENDING CONVERSION, INT, OR ERROR
        SETIPL    UCB$B_FIPL(R5)         ;LOWER PRIORITY TO DEVICE LEVEL
        BRB       10$                    ;JOIN COMMON CODE TO
                ; TERMINATE REQUEST

;
; POWER FAILURE DETECTED WHILE ATTEMPTING TO INITIATE A READ OR
; LOOPBACK REQUEST.  TERMINATE REQUEST THE SAME AS IF IT OCCURRED
; DURING THE QIO.
;

AD_POWERFAIL:
        ENBINT                                         ;LOWER IPL BACK TO FORK IPL
10$:    MOVZWL    #SS$_TIMEOUT,R0              ;SET STATUS TO TIMED OUT
        BRB       20$                    ;JOIN COMMON CODE TO TERMINATE
                ; REQUEST

;
; NORMAL STATUS, CANCEL I/O, AND GENERAL I/O REQUEST COMPLETION:
;

AD_DONE:
        CLRW      AD_CSR(R4)              ;CLEAR INTERRUPT ENABLE
        IOFORK                                         ;REQUEST RESUMPTION AS FORK PROCESS
        MOVZWL    #SS$_DATAOVERUN,R0          ;ASSUME CSR ERROR
        BBS       #AD_CSR_V_ERR,-           ;BRANCH IF SO
                UCB$W_AD_CSR(R5),20$        ;
        MOVZWL    #SS$_BUFFEROVF,R0          ;ASSUME BUFFER OVERRUN
        BBS       #UCB$W_CSR_V_BFO,-       ;BRANCH IF SO
                UCB$W_AD_CSR(R5),20$        ;
        MOVZWL    #SS$_NORMAL,R0           ;ELSE, STATUS IS NORMAL
    
```

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

```
20$: SUBW3  UCB$W_BCNT(R5),-      ;GET # BYTES REQUESTED
      IRP$W_BCNT(R3),R1          ; -# BYTES NOT XFERRED
      INSV  R1,#16,#16,R0        ; =# BYTES XFERRED
      CLRL  R1                   ;CLEAR SECOND I/O STATUS LONGWD
      REQCOM                          ;REQUEST I/O COMPLETION

      .DSABL  LSB
```

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

```

        .SBTTL  AD_INTERRUPT:    AD11-K A/D CONVERTER INTERRUPT SERVICE
;+
; AD_INTERRUPT - A/D CONVERTER INTERRUPT SERVICE
;
; THIS ROUTINE IS ENTERED VIA A JSB INSTRUCTION WHEN AN
; INTERRUPT OCCURS ON AN AD11 A/D CONVERTER.  INTERRUPT SERVICE
; GETS THE ADDRESS OF THE UCB OF THE INTERRUPTING DEVICE, RESTORES
; THE REMAINING CONTEXT OF THE DRIVER FORK PROCESS WHICH INITIATED
; THE DEVICE ACTIVITY, AND CALLS THE DRIVER FORK PROCESS.
;
; INPUTS:
;
;     ALL GENERAL REGISTERS = RANDOM
;     SP/ INTERRUPT STACK
;     0(SP) = ADDR OF IDB ADDR
;     4(SP) = SAVED R0
;     8(SP) = SAVED R1
;     12(SP) = SAVED R2
;     16(SP) = SAVED R3
;     20(SP) = SAVED R4
;     24(SP) = SAVED R5
;     28(SP) = SAVED PC
;     32(SP) = SAVED PSL
;     IPL/ HARDWARE DEVICE LEVEL
;
; OUTPUTS AT CALL TO DRIVER FORK:
;
;     R3 = RESTORED FROM DRIVER FORK PROCESS (IRP ADDR)
;     R4 = RESTORED FROM DRIVER FORK PROCESS (CSR ADDR)
;     R5 = UCB ADDR
;     STACK IS SAME AS ABOVE, BUT IDB POINTER POPPED
;     IPL/ HARDWARE DEVICE LEVEL
;-

        .ENABL  LSB

AD_INTERRUPT:                                ;A/D CONVERTER INTERRUPT SERVICE
        MOVL   @(SP)+,R3                     ;GET IDB ADDR
        MOVQ   IDB$$_CSR(R3),R4              ;GET DEVICE CSR AND UCB ADDR
        BCC    #UCB$$_INT,-                  ;BRANCH IF INT UNEXPECTED,
        UCB$$_STS(R5),AD_UN SOL             ; AND CLEAR EXPECTED BIT
        MOVL   UCB$$_FR3(R5),R3              ;RESTORE REMAINING DRIVER
        JSB    @UCB$$_FPC(R5)                ; CONTEXT: R3; (R4 ALREADY SET)
                                                ;CALL DRIVER FORK PROCESS
;
10$:    MOVQ   (SP)+,R0                       ;RESTORE REGISTERS
        MOVQ   (SP)+,R2
        MOVQ   (SP)+,R4
        REI
AD_UN SOL:                                    ;HANDLE UNSOLICITED INTERRUPT
        CLRW   AD_CSR(R4)                    ;DISMISS SPURIOUS INTERRUPT
        TSTW  AD_DBR(R4)                    ;READ DATA BUFFER TO CLEAR ERROR
        BRB   10$                            ;JOIN INTERRUPT RESTORE

        .DSABL  LSB
        .SBTTL  AD_CTLINIT:    AD11-K CONTROLLER INITIALIZATION
;+
; AD_CTLINIT - AD11-K CONTROLLER INITIALIZATION
;
; THIS ROUTINE IS CALLED AT SYSTEM STARTUP AND AFTER A POWER
; FAILURE.
;
; THE CSR IS CLEARED TO DISABLE INTERRUPTS.  THIS WILL FORCE THE
; LAST SAMPLE (IF ONE IS IN PROGRESS) TO TIME OUT IN CASE INITIALIZATION

```

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

```
; IS THE RESULT OF A POWER FAILURE.  THE TIMEOUT WILL OCCUR IN 0-1
; SECONDS.
;
; THE DATA BUFFER REGISTER IS READ TO CLEAR A PENDING CONVERSION,
; INTERRUPT, OR ERROR FOR DEVICE INITIALIZATION.
;
; INPUTS:
;
;     R4 = AD11 CSR ADDRESS
;     R5 = IDB ADDRESS OF DEVICE UNIT
;     R6 = ADDR OF DDB
;     R8 = ADDR OF CRB
;
; OUTPUTS:
;
;     ALL REGISTERS PRESERVED
;-

AD_CTLINIT:
    CLRW    AD_CSR(R4)           ; CLEAR CSR (IE IN PARTICULAR)
    TSTW    AD_DBR(R4)         ; CLEAR ANY PENDING CONVERSION,
                                ; INTERRUPT, OR ERROR
    RSB
;
```

SAMPLE DRIVER FOR AN A-TO-D CONVERTER

```

        .SBTTL  AD_UNITINIT:      AD11-K UNIT INITIALIZATION
;+
; AD_UNITINIT - AD11-K UNIT INITIALIZATION
;
; THIS ROUTINE IS CALLED AT SYSTEM STARTUP AND AFTER A POWER
; FAILURE.  THE UCB AND IDB ARE INITIALIZED.
;
; INPUTS:
;
;       R5 = ADDRESS OF DEVICE UCB
;
; OUTPUTS:
;
;       R0 = IDB ADDRESS
;       OTHER REGISTERS ARE PRESERVED
;       UCB$W_STS(R5), ONLINE BIT IS SET
;       IDB$L_OWNER(R0) = ADDRESS OF OWNING UCB
;-

AD_UNITINIT:
        BISW      #UCB$M_ONLINE,-          ;SET UNIT ONLINE
        UCB$W_STS(R5)                          ;
        MOVL     UCB$L_CRB(R5),R0           ;GET CRB ADDRESS
        MOVL     CRB$L_INTD+VEC$L_IDB(R0),R0 ;GET IDB ADDR
        MOVL     R5,IDB$L_OWNER(R0)        ;SET UCB ADDR OF OWNING UNIT
        RSB
AD_END:                                     ;
                                           ;END OF DRIVER LABEL

        .END

```


APPENDIX E

SAMPLE DRIVER FOR DR11s

This appendix contains the source listing of a driver for two connected DR11s.

SAMPLE DRIVER FOR DR11s

.TITLE XADRIVER - VAX/VMS DR11 DRIVER
.IDENT 'V02-005'

```
*****  
*  
* Copyright (c) 1979, 1980 *  
* by DIGITAL Equipment Corporation, Maynard, Mass. *  
*  
* This software is furnished under a license and may be used and copied *  
* only in accordance with the terms of such license and with the *  
* inclusion of the above copyright notice. This software or any other *  
* copies thereof may not be provided or otherwise made available to any *  
* other person. No title to and ownership of the software is hereby *  
* transferred. *  
*  
* The information in this software is subject to change without notice *  
* and should not be construed as a commitment by DIGITAL Equipment *  
* Corporation. *  
*  
* DIGITAL assumes no responsibility for the use or reliability of its *  
* software on equipment which is not supplied by DIGITAL. *  
*  
*****  
;   
; ++   
;   
; FACILITY:   
;   
; VAX/VMS Executive, I/O Drivers   
;   
; ABSTRACT:   
;   
; This module contains the DR11 driver:   
;   
; Tables for loading and dispatching   
; Controller initialization routine   
; FDT routine   
; The start I/O routine   
; The interrupt service routine   
; Device-specific cancel I/O   
; Error-logging register dump routine   
;   
; ENVIRONMENT:   
;   
; Kernel Mode, Nonpaged   
;   
; AUTHOR:   
;   
; S. PROGRAMMER JANUARY 1979   
;   
;   
; MODIFIED BY:   
;   
;   
; --
```

SAMPLE DRIVER FOR DR11s

.SBTTL External and local symbol definitions

; External symbols

\$ACBDEF	; AST control block
\$CRBDEF	; Channel request block
\$DDBDEF	; Device data block
\$DPTDEF	; Driver prologue table
\$EMBDEF	; EMB offsets
\$IDBDEF	; Interrupt data block
\$IODEF	; I/O function codes
\$IPLDEF	; Hardware IPL definitions
\$IRPDEF	; I/O request packet
\$PRDEF	; Internal processor registers
\$PRIDEF	; Scheduler priority increments
\$UCBDEF	; Unit control block
\$VECDEF	; Interrupt vector block
\$XADEF	; Define device specific characteristics

; Local symbols

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

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

; Other constants

XA_DEF_TIMEOUT	= 10	; 10 second default device timeout
XA_DEF_BUFSIZ	= 65535	; Default buffer size
XA_RESET_DELAY	= 2	; Delay N microseconds after RESET

; DR11 definitions that follow the standard UCB fields
; *** N O T E *** ORDER OF THESE UCB FIELDS IS ASSUMED

\$DEFINI UCB		
.=UCB\$\$_DPC+4		
\$DEF	UCB\$\$_XA_ATT	; Attention AST listhead
	.BLKL 1	
\$DEF	UCB\$\$_XA_CSRTMP	; Temporary storage of CSR image
	.BLKW 1	
\$DEF	UCB\$\$_XA_BARTMP	; Temporary storage of BAR image
	.BLKW 1	
\$DEF	UCB\$\$_XA_CSR	; Saved CSR on interrupt
	.BLKW 1	
\$DEF	UCB\$\$_XA_EIR	; Saved EIR on interrupt
	.BLKW 1	
\$DEF	UCB\$\$_XA_IDR	; Saved IDR on interrupt
	.BLKW 1	
\$DEF	UCB\$\$_XA_BAR	; Saved BAR register on interrupt
	.BLKW 1	
\$DEF	UCB\$\$_XA_WCR	; Saved WCR register on interrupt
	.BLKW 1	
\$DEF	UCB\$\$_XA_ERROR	; Saved device status flag
	.BLKW 1	
\$DEF	UCB\$\$_XA_DPR	; Data Path Register contents
	.BLKL 1	
\$DEF	UCB\$\$_XA_FMPR	; Final Map Register contents
	.BLKL 1	
\$DEF	UCB\$\$_XA_PMPR	; Previous Map Register contents

SAMPLE DRIVER FOR DR11s

```

        .BLKL 1
$DEF   UCB$W_XA_DPRN          ; Saved Datapath Register Number
        .BLKW 1              ; And Datapath Parity error flag

; Bit positions for device-dependent status field in UCB

        $VIELD UCB,0,<-          ; UCB device-specific bit definitions
          <ATTNAST,,M>,-        ; ATTN AST requested
          <UNEXPT,,M>,-        ; Unexpected interrupt received
        >
UCB$K_SIZE=.
        $DEFEND UCB

; Device register offsets from CSR address

        $DEFINI XA              ; Start of DR11 definitions
$DEF   XA_WCR                    ; Word count
        .BLKW 1
$DEF   XA_BAR                    ; Buffer address
        .BLKW 1
$DEF   XA_CSR                    ; Control/status

; Bit positions for device control/status register

        $EQLST XA$K,,0,1,<-      ; Define CSR FNCT bit values
          <FNCT1,2>-
          <FNCT2,4>-
          <FNCT3,3>-
          <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

```

SAMPLE DRIVER FOR DR11s

```
$DEF    XA_ODR          ; Output Data Buffer register
        .BLKW    1
$DEFEND XA          ; End of DR11 definitions
```

SAMPLE DRIVER FOR DRILLS

.SBTTL Device Driver Tables

; Driver prologue table

```

DPTAB      -                               ; DPT-creation macro
           END=XA END,-                     ; End of driver label
           ADAPTER=UBA,-                     ; Adapter type
           FLAGS=DPT$M_SVP,-                ; Allocate system page table
           UCBSIZE=UCB$K_SIZE,-             ; UCB size
           NAME=XADRIVER                     ; Driver name
DPT_STORE  INIT                             ; Start of load
           ; initialization table
DPT_STORE  UCB,UCB$B_FIPL,B,8                ; Device fork IPL
DPT_STORE  UCB,UCB$B_DIPL,B,22              ; Device interrupt IPL
DPT_STORE  UCB,UCB$L_DEVCHAR,L,<-           ; Device characteristics
           DEV$M_RTM!-                       ; Real time device
           DEV$M_ELG!-                       ; Error logging enabled
           DEV$M_IDV!-                       ; input device
           DEV$M_ODV>                        ; output device
DPT_STORE  UCB,UCB$B_DEVCLASS,B,DC$_REALTIME ; Device class
DPT_STORE  UCB,UCB$B_DEVTYPE,B,DT$_DR11W   ; Device Type
DPT_STORE  UCB,UCB$W_DEVBUFSIZ,W,-         ; Default buffer size
           XA DEF BUFSIZ
DPT_STORE  REINIT                             ; Start of reload
           ; initialization table
DPT_STORE  DDB,DDB$D_DDT,D,XA$DDT          ; Address of DDT
DPT_STORE  CRB,CRB$D_INTD+4,D,-            ; Address of interrupt
           XA INTERRUPT                       ; service routine
DPT_STORE  CRB,CRB$D_INTD+VEC$D_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=<<I3*4>+<<3+5+1>*4>>,-   ; Diagnostic buffer size
           ERLGBF=<<I3*4>+<1*4>+<EMB$D_DV_REGSAV>> ; Error log buffer size

```

; Function decision table

```

XA_FUNCTABLE:
FUNCTAB    ,-                               ; FDT for driver
           ; 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>

```

SAMPLE DRIVER FOR DR11s

```

.SBTTL XA_CONTROL_INIT, Controller initialization

;++;
; XA_CONTROL_INIT, Called when driver is loaded, system is booted, or
; power failure recovery.
;
; Functional Description:
;
;     1) Allocates the direct data path permanently
;     2) Assigns the controller data channel permanently
;     3) Clears the Control and Status Register
;     4) If power recovery, requests device timeout
;
; Inputs:
;
;     R4 = address of CSR
;     R5 = address of IDB
;     R6 = address of DDB
;     R8 = address of CRB
;
; Outputs:
;
;     VEC$V_PATHLOCK bit set in CRB$L_INTD+VEC$B_DATAPATH
;     UCB address placed into IDB$L_OWNER
;
;--

XA_CONTROL_INIT:

    MOVL   IDB$L_UCBLST(R5),R0      ; Address of UCB
    MOVL   R0,IDB$L_OWNER(R5)      ; Make permanent controller owner
    BISW   #UCB$M_ONLINE,UCB$W_STS(R0) ; Set device status "on-line"

; If powerfail has 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.

    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 DR11
    RSB    ; Done

```

SAMPLE DRIVER FOR DR11s

.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/Os with odd transfer count
; 2) Rejects QUEUE I/Os 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 IOSM_TIMED)
; P4 = Value for CSR FNCT bits (conditional on IOSM_SETFNCT)
; P5 = Value for ODR (conditional on IOSM_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 CSR and ODR image
;
; --

XA_READ_WRITE:
BLBC P2(AP),10$ ; Branch if transfer count even
2$: MOVZWL #SS$ BADPARAM,R0 ; Set error status code
5$: JMP G^EXE$ABORTIO ; Abort request
10$: MOVZWL IRP$W_FUNC(R3),R1 ; Fetch I/O function code
MOVWL P3(AP),IRP$L_MEDIA(R3) ; Set request specific timeout count
BBS #IO$V_TIMED,R1,15$ ; Branch if time-out specified
MOVWL #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

```

SAMPLE DRIVER FOR DR11s

```
; If this is a block mode transfer, check buffer for modify access
; whether or not the function is read or write. The DR11 does
; not decide whether to read or write, the user's device does.
; For word mode requests, return to read check or write check.
;
; If this is a BLOCK MODE request and the UBA Direct Data Path is
; in use, check the data buffer address for word alignment. If buffer
; is not word aligned, reject the request.
```

```
        BBS      #IO$V_WORD,IRP$W_FUNC(R3),30$
                                ; Branch if word mode transfer
        BBS      #XA$V_DATAPATH,UCB$SL_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
```


SAMPLE DRIVER FOR DRILLS

```

.SBTTL XA_SETMODE, Set Mode, Set characteristics FDT

; ++
; XA_SETMODE, FDT routine to process SET MODE and SET CHARACTERISTICS
;
; Functional description:
;
;     If IOSM_ATTNAST modifier is set, queue attention AST for device
;     If IOSM_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 IOSM_ATTNAST is specified, queue AST on UCB attention AST list.
;     If IOSM_DATAPATH is specified, queue packet to driver.
;     Else, use exec routine to update device characteristics
;
; --

XA_SETMODE:
    MOVZWL  IRP$W_FUNC(R3),R0      ; Get entire function code
    BBC     #IOSV_ATTNAST,R0,20$   ; Branch if not an ATTN AST

; Attention AST request

    PUSHR   #^M<R4,R7>
    MOVAB   UCBSL_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   #UCBSM_ATTNAST,UCBSW_DEVSTS(R5) ; Flag ATTN AST expected.
    BBC    #UCBSV_UNEXPT,UCBSW_DEVSTS(R5),10$ ; Deliver AST if unsolicited interrupt

    BSBW   DEL_ATTNA$T
10$:     JMP   G^EXE$FINISHIO       ; That's all for now

```

SAMPLE DRIVER FOR DR11s

```
; 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 priv for operation  
50$:   CLRL   R1                        ;  
       JMP     G^EXE$ABORTIO           ; Abort I/O on error
```

SAMPLE DRIVER FOR DR11s

```

.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 CSR are set. If
; the transfer count is zero, the STATUS bits in the DR11 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$$_CSR EQ 0
    MOVL   UCB$$_CRB(R5),R4      ; Address of CRB
    MOVL   @CRB$$_INTD+VECS$_IDB(R4),R4 ; Address of CSR

; Fetch the I/O function code

    MOVZWL IRP$$_FUNC(R3),R1      ; Get entire function code
    MOVW   R1,UCB$$_FUNC(R5)     ; Save FUNC in UCB for error logging
    EXTZV  #IOSV_FCODE,#IOSS_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   #IOS$_SETCHAR,R2      ; Set characteristics?
    BNEQ   3$

; ++
; SET CHARACTERISTICS - Process Set Characteristics QIO function
;
; INPUTS:
;
; XA_DATAPATH bit in Device Characteristics specifies which data path
; to use. If bit is a one, use buffered data path. If zero, use
; direct datapath.
;
;

```

SAMPLE DRIVER FOR DR11s

```

; OUTPUTS:
;
; CRB is flagged as to which datapath to use.
; DEVDEPEND bits in device characteristics are updated
;   XA_DATAPATH = 1 -> buffered data path in use
;   XA_DATAPATH = 0 -> direct data path in use
;--

        MOVL    UCB$$_CRB(R5),R0          ; Get CRB address
        MOVQ    IRP$$_MEDIA(R3),UCB$$_DEVCLASS(R5) ; Set device characteristics
        BISB    #VEC$$_PATHLOCK,CRB$$_INTD+VEC$$_DATAPATH(R0)
                                                ; Assume direct datapath
        BBC     #XA$$_DATAPATH,UCB$$_DEVDEPEND(R5),2$ ; Were we right?
        BICB    #VEC$$_PATHLOCK,CRB$$_INTD+VEC$$_DATAPATH(R0) ; Set buffered datapath
2$:
        CLRL    R1                          ; Return Success
        MOVZWL  #SS$$_NORMAL,R0
        REQCOM

; If subfunction modifier for device reset is set, do one here
3$:     BBC     S^#IO$$_V_RESET,R1,4$      ; Branch if not device reset
        BSBW    XA_DEV_RESET              ; Reset DR11

; This must be a data transfer function - i.e. READ OR WRITE
; Check to see if this is a zero length transfer.
; If so, only set CSR FNCT bits and return STATUS from CSR
4$:     TSTW    UCB$$_BCNT(R5)             ; Is transfer count zero?
        BNEQ    10$                       ; No, continue with data transfer
        BBC     S^#IO$$_V_SETFNCT,R1,6$   ; Set CSR FNCT specified?
        DSBINT
        MOVW    IRP$$_SEGVBN+2(R3),XA_ODR(R4)
                                                ; Store word in ODR
        MOVZWL  XA_CSR(R4),R0
        BICW    #<XA_CSR$$_FNCT!XA_CSR$$_ERROR>,R0
        BISW    IRP$$_SEGVBN(R3),R0
        MOVW    R0,XA_CSR(R4)
        BBC     #XA$$_LINK,UCB$$_DEVDEPEND(R5),5$ ; Link mode?
        BICW3   #XA$$_FNCT2,R0,XA_CSR(R4) ; Make FNCT bit 2 a pulse
5$:
        ENBINT
6$:
        BSBW    XA_REGISTER                ; Fetch DR11 registers
        BLBS    R0,7$                     ; If error, then log it
        JSB     G^ERL$$_DEVICERR          ; Log a device error
7$:     JSB     G^IOC$$_DIAGBUFILL         ; Fill diagnostic buffer if specified
        MOVL    UCB$$_XA_CSR(R5),R1       ; Return CSR and EIR in R1
        MOVZWL  UCB$$_XA_ERROR(R5),R0     ; Return status in R0
        BISB    #XA_CSR$$_IE,XA_CSR(R4)  ; Enable device interrupts
        REQCOM                                ; Request done

; Build CSR image in R0 for later use in starting transfers
10$:
        DIVW3   #2,UCB$$_BCNT(R5),UCB$$_XA_DPR(R5)
                                                ; Make byte count into word count
        MOVZWL  XA_CSR(R4),R0
        BICW    #^<XA_CSR$$_FNCT>,R0
        BISW    #XA_CSR$$_IE,R0          ; Set Interrupt Enable
        BBC     S^#IO$$_V_SETFNCT,R1,20$ ; Set FNCT bits in CSR?
        BICW    #<XA_CSR$$_FNCT>,R0     ; Yes, Clear previous FNCT bits
        BISB    IRP$$_SEGVBN(R3),R0     ; OR in new value

```

SAMPLE DRIVER FOR DRILLS

```
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
```

SAMPLE DRIVER FOR DR11s

```

; ++
; 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 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 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
        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 registers and start transfer
; Note that read-modify-write cycles are NOT performed to the DR11 CSR.
; The CSR is always written directly into. This prevents inadvertently setting
; the EIR select flag (writing bit 15) if error happens to become true.

        DSBINT                                ; Disable interrupts (powerfail)
        MNEGW  UCB$L_XA_DPR(R5),XA_WCR(R4)
                                                ; Load negative of transfer count
        MOVW  R1,XA_BAR(R4)                 ; Load low 16 bits of bus address
        MOVW  R0,XA_CSR(R4)                 ; Load CSR image less "GO" and "CYCLE"
        BBC   #XA$V_LINK,UCB$L_DEVDEPEND(R5),26$ ; Link mode?
        MOVW  R2,XA_CSR(R4)                 ; Yes, load CSR image less "FNCT" bit 2
                                                ; Only if link mode in dev characteristics

```

SAMPLE DRIVER FOR DR11s

```

26$:
; Wait for transfer complete interrupt, powerfail, or device timeout
      WFIKPCX XA_TIME_OUT,IRPSL_MEDIA(R3) ; Wait for interrupt
; Device has interrupted, FORK
      IOFORK                               ; FORK to lower IPL
; Handle request completion, release UBA resources, check for errors
      MOVZWL #SS$ NORMAL,-(SP)             ; Assume success, store code on stack
      CLRW   UCBSW_XA_DPRN(R5)             ; Clear DPR number and DPR error flag
      PURDPR                               ; Purge UBA buffered data path
      BLBS   R0,27$                         ; Branch if no datapath error
      MOVZWL #SS$ PARITY,(SP)              ; Flag parity error on device
      INCB   UCBSW_XA_DPRN+1(R5)           ; Flag PDR error for log
27$:   MOVL  R1,UCBSL_XA_DPR(R5)           ; Save data path register in UCB
      EXTZV  #VECSV_DATAPATH,-            ; Get datapath register no.
      #VECSV_DATAPATH,-                    ; For error log
      CRBSL_INTD+VECSB_DATAPATH(R3),R0
      MOVB   R0,UCBSW_XA_DPRN(R5)         ; Save for later in UCB
      EXTZV  #9,#7,UCBSW_XA_BAR(R5),R0    ; Low bits, final map register no.
      EXTZV  #4,#2,UCBSW_XA_CSR(R5),R1    ; High bits of map register no.
      INSV   R1,#7,#2,R0                  ; Entire map register number
      CMPW   R0,#496                       ; Is map register number in range?
      BGTR   28$                           ; No, forget it - compound error
      MOVL   (R2)[R0],UCBSL_XA_FMPR(R5)   ; Save mapregister contents
      CLRL   UCBSL_XA_PMPR(R5)            ; Assume no previous map register
      DECL   R0                             ; Was there a previous map register?
      CMPV   #VECSV_MAPREG,#VECSV_MAPREG,-
      CRBSL_INTD+VECSW_MAPREG(R3),R0
      BGTR   28$                           ; No if greater
      MOVL   (R2)[R0],UCBSL_XA_FMPR(R5)   ; Save previous map register contents
28$:   RELMPR                               ; Release UBA resources
      RELDPR
; Check for errors and return status
      TSTW   UCBSW_XA_WCR(R5)             ; All words transferred?
      BEQL   30$                           ; Yes
      MOVZWL #SS$ OPINCOMPL,(SP)          ; No, flag operation not complete
30$:   BBC   #XA_CSR$V_ERROR,UCBSW_XA_CSR(R5),35$ ; Branch on CSR error bit
      MOVZWL UCBSW_XA_ERROR(R5),(SP)      ; Flag for controller/drive error status
      BSBW   XA_DEV_RESET                  ; Reset DR11
35$:   BLBS   (SP),40$                     ; Any errors after all this?
      JSB   G^ERL$DEVICERR                ; Yes, log them
40$:   BSBW   DEL_ATTNAST                  ; Deliver outstanding ATTN ASTs
      JSB   G^IOC$DIAGBUFILL              ; Fill diagnostic buffer
      MOVL   (SP)+,R0                       ; Get final device status
      MULW3  #2,UCBSW_XA_WCR(R5),R1        ; Calculate final transfer count
      ADDW   UCBSW_BCNT(R5),R1
      INSV   R1,#16,#16,R0                 ; Insert into high byte of IOSB
      MOVL   UCBSW_XA_CSR(R5),R1          ; Return CSR and EIR in IOSB
      BISB   #XA_CSR$M_IE,XA_CSR(R4)      ; Enable interrupts
      REQCOM                               ; Finish request in exec
      .DSABL LSB

```

SAMPLE DRIVER FOR DR11s

```

; ++
; 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
;   and a read (from DR11 to memory).
;   For a write, data is fetched from memory, loaded into the ODR of the
;   DR11 and the system waits for an interrupt. For a read, the system
;   waits for a DR11 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 ODR one word at a time, wait for interrupt for each
;   word.
; --

10$:
        BSBW    MOVFRUSER                  ; Get two bytes from user buffer
        DSBINT                                ; Lock out interrupts
                                                ; Flag interrupt expected
        MOVW    R1,XA_ODR(R4)              ; Move data to DR11
        MOVW    UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ; Set DR11 CSR
        BBC     #XA$V_LINK,UCB$L_DEVDEPEND(R5),15$ ; Link mode?
        BICW3   #XA$K_FNCT2,UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ; Clear interrupt FNCT bit 2
                                                ; Only if Link mode specified

15$:

; Wait for interrupt, powerfail, or device timeout

        WFIKPCH XA_TIME_OUTW,IRP$L_MEDIA(R3)

; Check for errors, decrement transfer count, and loop until complete

        IOFORK                                ; Fork to lower IPL
        BITW    #XA_EIR$M_NEX!-
                XA_EIR$M_MULTI!-
                XA_EIR$M_ACLO!-
                XA_EIR$M_PAR!-
                XA_EIR$M_DLT,UCB$W_XA_EIR(R5) ; Any errors?
        BEQL   20$                            ; No, continue
        BRW    40$                            ; Yes, abort transfer.
20$:
        DECW    UCB$L_XA_DPR(R5)              ; All words transferred?
        BNEQ   10$                            ; No, loop until finished.

; Transfer is done, clear interrupt expected flag and FORK
; All words read or written in WORD MODE. Finish I/O.

```


SAMPLE DRIVER FOR DRILLS

RETURN_STATUS:

```

        JSB      G^IOC$DIAGBUFILL      ; Fill diagnostic buffer if present
        BSBW    DEL ATTNAST            ; Deliver outstanding ATTN ASTs
        MOVZWL  #SS$ NORMAL,R0        ; Complete success status
22$:     MULW3   #2,UCB$X_XA_DPR(R5),R1 ; Calculate actual bytes transferred
        SUBW3   R1,UCB$W_BCNT(R5),R1  ; From requested number of bytes
        INSV    R1,#16,#16,R0         ; And place in high word of R0
        MOVL   UCB$W_XA_CSR(R5),R1    ; Return CSR and EIR status
        BISB   #XA_C$R$M_IE,XA_CSR(R4) ; Enable device interrupts
        REQCOM  ; Finish request in executive

; ++
; WORD MODE READ -- Read (input) in word mode
;
; FUNCTIONAL DESCRIPTION:
;
; Transfer the requested number of words from the DR11 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$:     SETIPL  UCB$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.

        BBSC   #UCB$V_UNEXPT,UCB$W_DEVSTS(R5),37$ ; Branch if unexpected interrupt
        DSBINT

35$:

; Wait for interrupt, powerfail, or device timeout

        WFIKPCH XA_TIME_OUTW,IRP$M_MEDIA(R3)

; Check for errors, decrement transfer count and loop until done

        IOFORK  ; Fork to lower IPL

37$:     BITW   #XA_EIR$M_NEX!-
          XA_EIR$M_MULT!-
          XA_EIR$M_ACLO!-
          XA_EIR$M_PARI-
          XA_EIR$M_DLT,UCB$W_XA_EIR(R5) ; Any errors?
        BNEQ   40$ ; Yes, abort transfer.
        BSBW   MOVTOUSER ; Store two bytes into user buffer

; Send interrupt back to sender. Acknowledge we got last word.

        DSBINT
        MOVW   UCB$W_XA_CSRTMP(R5),XA_CSR(R4)
        BBC   #XA$V_LINK,UCB$M_DEVDEFEND(R5),38$ ; Link mode?
        BICW3 #XA$K_FNCT2,UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ; Yes, clear FNCT 2

38$:     DECW   UCB$M_XA_DPR(R5) ; Decrement transfer count
        BNEQ   35$ ; Loop until all words transferred
        ENBINT
        BRB   RETURN_STATUS ; Finish request in common code

```

SAMPLE DRIVER FOR DR11s

```

; Error detected in word mode transfer

40$:
    BSBW    DEL_ATTNAST                ; Deliver ATTN AST's
    BSBW    XA_DEV_RESET              ; Error, reset DR11
    JSB     G^IOC$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 loc
    MOVZBL  #2,R2                    ; Fetch two bytes
    JSB     G^IOC$MOVFRUSER          ; Call exec routine to do the deed
    MOVL    (SP)+,R1                 ; Retrieve the bytes
    BRB     20$                      ; Update UCB buffer pointers

;
; MOVTOUSER - Routine to store two bytes into 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$MOVTOUSER          ; Call exec
20$:
    ADDW    #2,UCB$W_BOFF(R5)        ; Update buffer pointers in UCB
    BICW    #^C<^X01FF>,UCB$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

```

SAMPLE DRIVER FOR DR11s

```

        .SBTTL  DR11 DEVICE TIMEOUT
; ++
; DR11 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 CSR
; Return error status
;
; Power failure will appear as a device timeout
; --
        .ENABL  LSB
XA_TIME_OUT:                                ; Timeout for DMA transfer

        SETIPL  UCB$B_FIPL(R5)              ; Lower to FORK IPL
        PURDPR                                ; Purge buffered data path in UBA
        RELMPR                                ; Release UBA map registers
        RELDPR                                ; Release UBA data path

XA_TIME_OUTW:                                ; Timeout for WORD mode transfer

        MOVL   UCB$L_CRB(R5),R4              ; Fetch address of CSR
        MOVL   @CRB$L_INTD+VEC$L_IDB(R4),R4
        BSBW   XA REGISTER                   ; Read DR11 registers
        JSB    G^IOC$DIAGBUFILL              ; Fill diagnostic buffer
        JSB    G^ERL$DEVICTMO                ; Log device timeout
        BSBW   DEL_ATTNAST                   ; And deliver the ASTs
        BSBW   XA DEV RESET                  ; Reset controller
        MOVZWL #S$S_TIMEOUT,R0              ; Error status
        CLRL   R1
        CLRW   UCB$W_DEVSTS(R5)              ; Clear ATTN AST flags
        BICW   #<UCB$M_TIM!UCB$M_INT!UCB$M_TIMEOUT!UCB$M_CANCEL!UCB$M_POWER>,-
        UCB$W_STS(R5)                        ; Clear unit status flags
        REQCOM                                ; Complete I/O in exec
        .DSABL  LSB
        .PAGE

```

SAMPLE DRIVER FOR DR11s

```

        .SBTTL  XA_INTERRUPT, Interrupt service routine for DR11
; ++
; XA_INTERRUPT, Handles interrupts generated by DR11
;
; Functional description:
;
;     This routine is entered whenever an interrupt is generated
;     by the DR11.  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 CSRs are stored in the UCB.
;
; --
XA_INTERRUPT:
        MOVL    @(SP)+,R4          ; Interrupt service for DR11
        MOVQ    (R4),R4           ; Address of IDB and pop SP
                                   ; CSR and UCB address from IDB

; Read the DR11 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$

```

SAMPLE DRIVER FOR DR11s

; Deliver ATTN ASTs if no interrupt expected and set unexpected
; interrupt flag.

25\$:

```
BISW   #UCB$M_UNEXPT,UCB$W_DEVSTS(R5) ; Set unexpected interrupt flag
BSBW   DEL_ATTNAST                    ; Deliver ATTN ASTs
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
```

SAMPLE DRIVER FOR DR11s

```

.SBTTL XA_REGISTER - Handle DR11 CSR transfers
; ++
; XA_REGISTER - Routine to handle DR11 register transfers
;
; INPUTS:
;
;     R4 - DR11 CSR address
;     R5 - UCB address of unit
;
; OUTPUTS:
;
;     CSR, EIR, WCR, BAR, IDR, and status are read and stored into UCB.
;     The DR11 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 SSS_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
; SSS_CTRLERR.
;
;     R0,R1 - destroyed, all other registers preserved.
; --

XA_REGISTER:

    MOVZWL #SS$ NORMAL,R0          ; Assume success
    MOVZWL XA_CSR(R4),R1          ; Read CSR
    MOVW   R1,UCB$W_XA_CSR(R5)    ; Save CSR in UCB
    BBC   #XA_CSR$V_ERROR,R1,55$ ; Branch if no error
    MOVZWL #SS$ DRVERR,R0        ; Assume "drive" error
55$:    BICW   #^C<XA_CSR$M_FNCT>,R1 ; Clear all uninteresting bits for later
    BISB   #<XA_CSR$M_ERROR/256>,XA_CSR+1(R4) ; Set EIR flag
    MOVW   XA_EIR(R4),UCB$W_XA_EIR(R5) ; Save EIR in UCB
    MOVW   R1,XA_CSR(R4)         ; Clear EIR flag and errors
    MOVW   XA_CSR(R4),R1         ; Read CSR back
    BBC   #XA_CSR$V_ATTN,R1,60$  ; If attention still set, hard error
    MOVZWL #SS$ CTRLERR,R0      ; Flag hard controller error
60$:    MOVW   XA_IDR(R4),UCB$W_XA_IDR(R5) ; Save IDR in UCB
    MOVW   XA_BAR(R4),UCB$W_XA_BAR(R5)
    MOVW   XA_WCR(R4),UCB$W_XA_WCR(R5)
    MOVW   R0,UCB$W_XA_ERROR(R5) ; Save status in UCB
    RSB

```

SAMPLE DRIVER FOR DR11s

```

.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 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$:
        SETIPL  UCB$B_DIPL(R5)                    ; Lock out device interrupts
        JSB     G^IO$CANCELIO                      ; Check if transfer going
        BBC     #UCB$V_CANCEL,UCB$W_STS(R5),30$    ; Branch if not for this guy

; If BLOCK mode DMA request in progress, release UBA resources
; If transfer is in progress, do a device reset to DR11

        BBC     #UCB$V_INT,UCB$W_STS(R5),25$      ; Branch if transfer not in progress
        BBS     #IO$V_WORD,IRP$W_FUNC(R3),25$    ; Branch if not BLOCK mode transfer
        PUSHR   #^M<R2,R3,R4>                    ; Save some registers
        MOVL   UCB$L_CRB(R5),R4                   ; Get CRB address
        MOVL   @CRB$L_INTD+8(R4),R4               ; Get pointer to CSR in IDB
        BSBW   XA_DEV_RESET                        ; Reset DR11
        PURDPR                                ; Purge UBA buffered data path
        RELMPR                                ; Release UBA map registers
        RELDPR                                ; Release UBA data path register
        POPR   #^M<R2,R3,R4>

```

SAMPLE DRIVER FOR DRlls

```
25$: MOVZWL #SS$_CANCEL,R0 ; Status is request canceled
      CLRL R1
      CLRW UCB$_DEVSTS(R5) ; Clear unexpected interrupt flag
      BICW #<UCB$_TIM!UCB$_BSY!UCB$_CANCEL!UCB$_INT!UCB$_TIMOUT>,-
           UCB$_STS(R5) ; Clear unit status flags
      REQCOM ; Jump to exec to finish I/O

30$: SETIPL UCB$_FIPL(R5) ; Lower to FORK IPL
      RSB ; Return
      .PAGE
```


SAMPLE DRIVER FOR DR11s

```

.SBTTL DEL_ATTNASt, Deliver ATTN ASTs
; ++
; DEL_ATTNASt, Deliver all outstanding ATTN ASTs
;
; Functional description:
;
; This routine is used by the DR11 driver to deliver all of the
; outstanding attention ASTs. It is copied from COM$DELATTNASt in
; the exec. In addition, it places the saved value of the DR11 CSR
; and Input Data Buffer Register in the AST paramater.
;
; Inputs:
;
; R5 = UCB of DR11 unit
;
; Outputs:
;
; R0,R1,R2 Destroyed
; R3,R4,R5 Preserved
; --
DEL_ATTNASt:
    BBCC      #UCB$V_ATTNASt,UCB$W_DEVSTs(R5),30$
              ; Any ATTN ASTs expected?
10$:    PUSHR  #^M<R3,R4,R5>          ; Save R3,R4,R5
        MOVL  8(SP),R1              ; Get address of UCB
        MOVAB UCB$L_XA_ATTNASt(R1),R2 ; Address of ATTN AST listhead
        MOVL  (R2),R5              ; Address of next entry on list
        BEQL  20$                  ; No next entry, end of loop
        BICW  #UCB$M_UNEXPT,UCB$W_DEVSTs(R1) ; Clear unexpected interrupt flag
        MOVL  (R5),(R2)            ; Close list
        MOVW  UCB$W_XA_IDR(R1),ACB$L_KAST+6(R5)
              ; Store IDR in AST paramater
        MOVW  UCB$W_XA_CSR(R1),ACB$L_KAST+4(R5)
              ; Store CSR in AST paramater
        PUSHAB B^10$              ; Set return address for FORK
        FORK                                ; FORK for this AST

; AST fork procedure

        MOVQ  ACP$L_KAST(R5),ACP$L_AST(R5)
              ; Rearrange entries
        MOVB  ACP$L_KAST+8(R5),ACB$B_RMOD(R5)
        MOVL  ACP$L_KAST+12(R5),ACP$L_PID(R5)
        CLRL  ACP$L_KAST(R5)
        MOVZBL #PRI$Iocom,R2      ; Set up priority increment
        JMP   G^SCH$QAST          ; Queue the AST

20$:    POPR   #M<R3,R4,R5>        ; Restore registers
30$:    RSB                                ; Return
        .PAGE

```

SAMPLE DRIVER FOR DR11s

```

.SBTTL XA_REGDUMP - DR11 register dump routine
; ++
; XA_REGDUMP - DR11 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 CSR by
;               by the driver.
;     BARTMP - The last value written into the DR11 BAR by
;               the driver during a block mode transfer.
;     CSR - The CSR image at the last interrupt
;     EIR - The EIR image at the last interrupt
;     IDR - The IDR image at the last interrupt
;     BAR - The BAR image at the last interrupt
;     WCR - Word count register
;     ERROR - The system status at request completion
;     PDRN - UBA Datapath Register number
;     DPR - The contents of the UBA Data Path register
;     FMPR - The contents of the last UBA Map register
;     PMRP - The contents of the previous UBA Map register
;     DPRF - Flag for purge datapath error
;             0 = no purge datapath error
;             1 = parity error when datapath was purged
;
; Note that the values stored are from the last completed transfer
; operation. If a zero transfer count is specified, then the
; values are from the last operation with a nonzero transfer count.
; --

XA_REGDUMP:

    MOVZBL #11,(R0)+           ; Eleven registers are stored
    MOVAB  UCB$W_XA_CSRTMP(R5),R1 ; Get address of saved register images
    MOVZBL #8,R2              ; Return 8 registers here
10$:   MOVZWL (R1)+,(R0)+
    SOBGTR R2,10$             ; Move them all
    MOVZBL UCB$W_XA_DPRN(R5),(R0)+ ; Save datapath register number
    MOVZBL #3,R2              ; And 3 more here
20$:   MOVL (R1)+,(R0)+       ; Move UBA register contents
    SOBGTR R2,20$
    MOVZBL UCB$W_XA_DPRN+1(R5),(R0)+ ; Save Datapath Parity Error Flag
    RSB
    .PAGE

```

SAMPLE DRIVER FOR DR11s

```

.SBTTL XA_DEV_RESET - Device reset DR11
; ++
; XA_DEV_RESET - DR11 Device reset routine
;
; This routine raises IPL to device IPL, performs a device reset to
; the required controller, and re-enables device interrupts.
;
; Inputs:
;
; R4 - Address of Control and Status Register
; R5 - Address of UCB
;
; Outputs:
;
; Controller is reset, controller interrupts are enabled
;
; --
XA_DEV_RESET:
        PUSHR    #^M<R0,R1,R2>           ; Save some registers
        DSBINT   #PR$ICR,R0              ; Raise IPL to lock all interrupts
        MOVB    #<XA_CSR$M_MAINT/256>,XA_CSR+1(R4)
        CLRB    XA_CSR+1(R4)

; *** Must delay here depending on reset interval

        MOVZBL  #XA_RESET_DELAY,R2      ; No. of microseconds to wait
5$:     MFPR    #PR$ICR,R0              ; Read interval clock
10$:    MFPR    #PR$ICR,R1              ; Read it again
        CML    R0,R1                    ; Compare both clock readings
        BEQL   10$                       ; Repeat until they differ
        SOBGTR R2,5$                     ; Do this the specified no. of times

        MOVB    #XA_CSR$M_IE,XA_CSR(R4) ; Re-enable device interrupts
        ENBINT  #PR$ICR,R0              ; Restore IPL
        POPR    #^M<R0,R1,R2>           ; Restore registers

        RSB

XA_END:                               ; End of driver label
        .END

```

APPENDIX F
MASSBUS ADAPTER

The MASSBUS links devices to physical memory. The MASSBUS adapter performs the following functions that allow communication between devices and memory:

- Mapping of virtual addresses to physical page frame numbers
- Buffering of data for transfers from main memory to the MASSBUS and vice versa
- Dispatching interrupts from MASSBUS devices to the SBI

A MASSBUS adapter supports any combination of mass storage devices. Each magnetic tape controller supports up to eight tape drives. Each disk controller supports a single disk drive. The DR70 is a general purpose interface that acts as a controller for one or more non-standard devices. Only one controller can transfer data at a time. Figure F-1 illustrates a possible MASSBUS configuration.

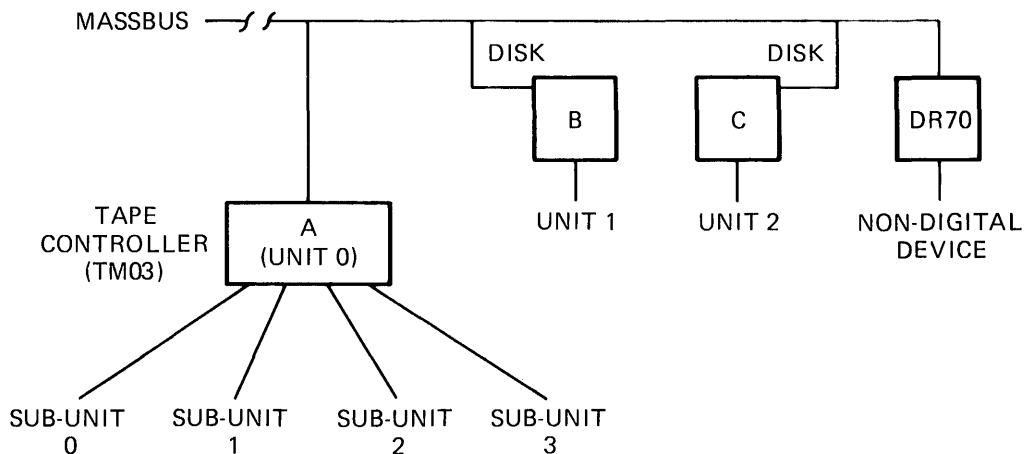


Figure F-1 MASSBUS Configuration

This appendix describes the data structures and macros used by DIGITAL for its standard magnetic tape and disk products. Customers using the DR70 should use equivalent techniques.

The MASSBUS adapter has two sets of registers:

- Internal registers for the MASSBUS adapter; that is, MBA registers
- External registers for each device on the MASSBUS; that is, device registers

MASSBUS ADAPTER

External registers are device-dependent.

The MASSBUS contains 256 map registers. The driver for a MASSBUS device must obtain ownership of the MBA controller before loading map registers.

Each map register is a longword. Bits 21 through 30 of each register are reserved; they cannot be written. Use of MBA map registers is analogous to use of UBA map registers with the following exceptions:

- MBA map registers do not contain a byte offset field; the MBA virtual address register (VAR) contains the byte offset.
- MBA map registers do not contain a data path field; the MBA has a single data path.

Figure F-2 illustrates the mapping of a virtual address to a page frame number.

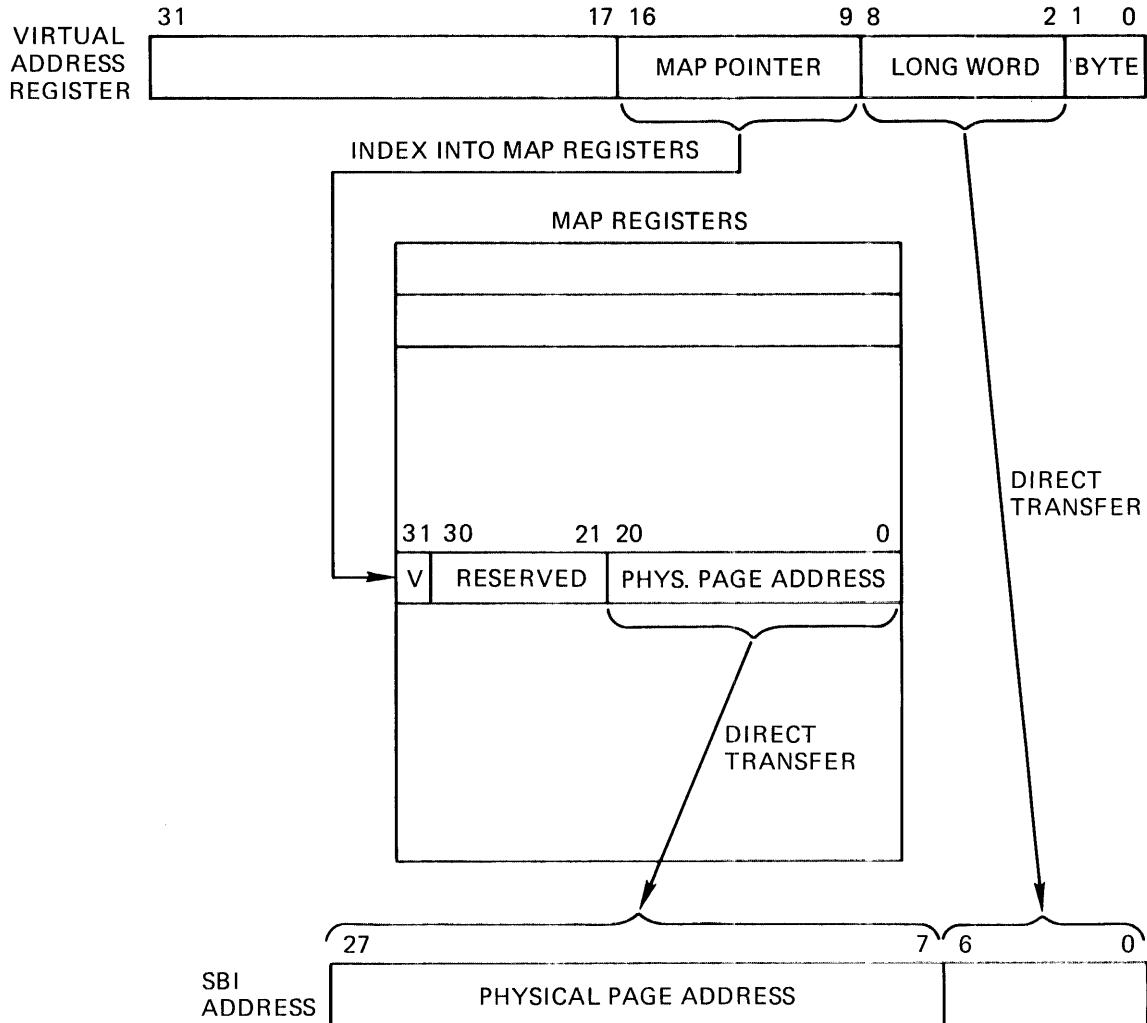


Figure F-2 Mapping of a Virtual Address to a Page Frame Number

MASSBUS ADAPTER

Each MASSBUS adapter has a 2048-longword physical address space. For the MASSBUS adapter, bits 10 and 11 of a longword address indicate the part of the MBA nexus address space to which the address refers. Addresses can refer to the MBA internal registers, external registers, or map registers with the following encodings of bits 10 and 11.

<u>Bits 11 and 10</u>	<u>Meaning</u>
0 0	MBA internal registers
0 1	MBA external registers Bits 0 through 6 select the register. Bits 7 through 9 select the unit or subcontroller.
1 0	Map register Bits 0 through 9 specify the map register index.

Bits 13 through 16 of the address specify the nexus position (tr number) of the MASSBUS adapter. The address of the nexus position is the address of the start of MBA registers. The address of the start of MBA space depends on the tr number at which the MASSBUS adapter is installed. For examples in this appendix, 20014000 is used as the starting address of the MBA registers for the MBA at tr number 0; refer to Figure F-3. The programmer of a driver under VAX/VMS uses only virtual addresses; physical addresses are visible only during the debugging of the driver.

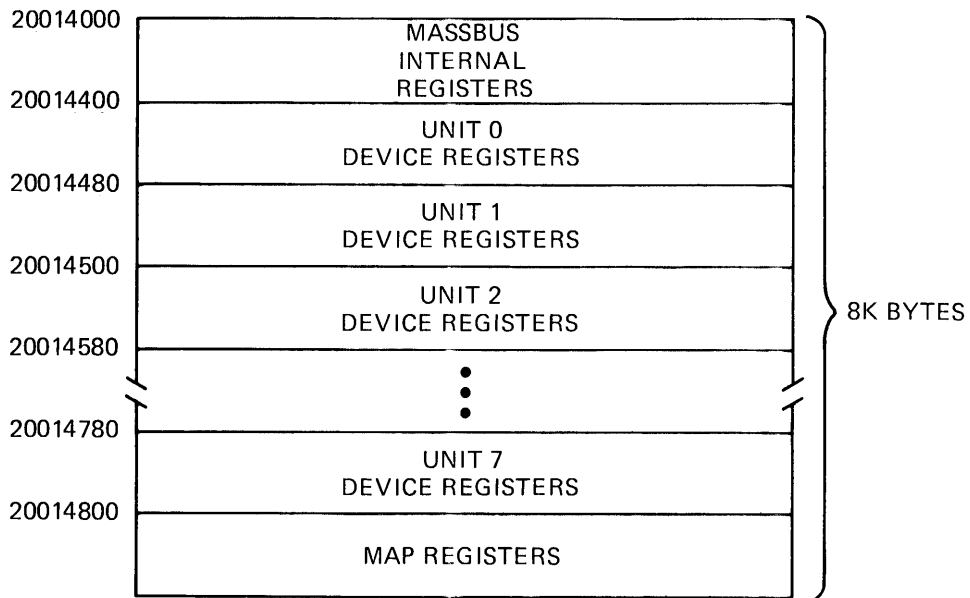


Figure F-3 Location of MASSBUS Registers

MASSBUS ADAPTER

Thus, to address a map address register in the MASSBUS adapter at tr 10, the driver constructs the following address:

20014800 + map register index

20014000 indicates tr 10.

800 indicates a map register (bits 10 and 11).

To address a device register, the driver constructs the following address:

20014400 .OR. device .OR. register select

400 indicates a device register (bits 10 and 11).

F.1 I/O DATA BASE FOR MASSBUS DEVICES

In the simple case (that is, a single-unit controller like a disk attached to the MASSBUS), the driver loading procedure constructs a channel request block for the MASSBUS adapter. The MASSBUS adapter is the device controller for all devices attached to the MASSBUS. Figure F-4 illustrates the I/O data base for a single-unit controller (disk) attached to the MASSBUS.

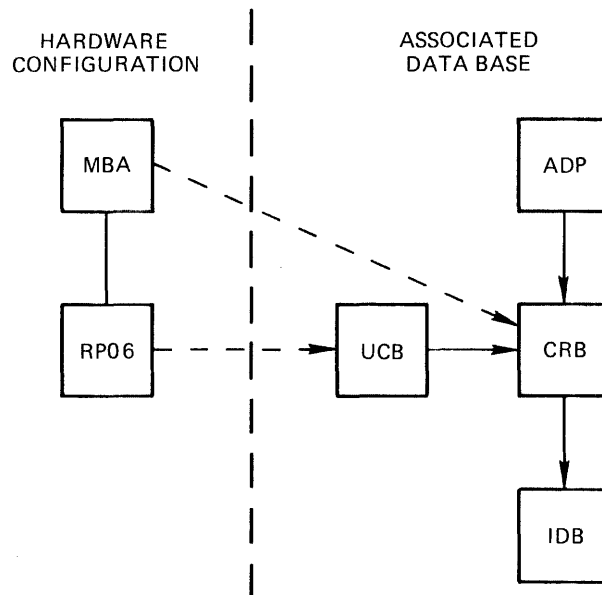


Figure F-4 I/O Data Base for MASSBUS Disk Unit

In the case of a multiunit controller, however, the I/O data base created varies slightly from the I/O data base for a UNIBUS configuration. Multiunit controllers (for example, magnetic tape drives on a TM03 formatter), have two levels of channel request blocks and interrupt data blocks. When multiple units are attached to a controller that is attached to the MASSBUS, the driver loading procedure creates one CRB and IDB for the MASSBUS adapter and one additional CRB and one additional IDB for each multiunit controller attached to the MASSBUS. Figure F-5 illustrates the I/O data base created for a disk unit and two tape units attached to the MASSBUS.

MASSBUS ADAPTER

Before a driver can activate a transfer on a unit attached to a multiunit controller, the driver must request both the primary controller (the TM03 controller) and the secondary controller (the MASSBUS adapter).

Nontransfer functions do not require the MASSBUS adapter. For example, tape positioning functions require only the magnetic tape controller and the unit; the MASSBUS controller is free for other operations (for example, data transfers on other units).

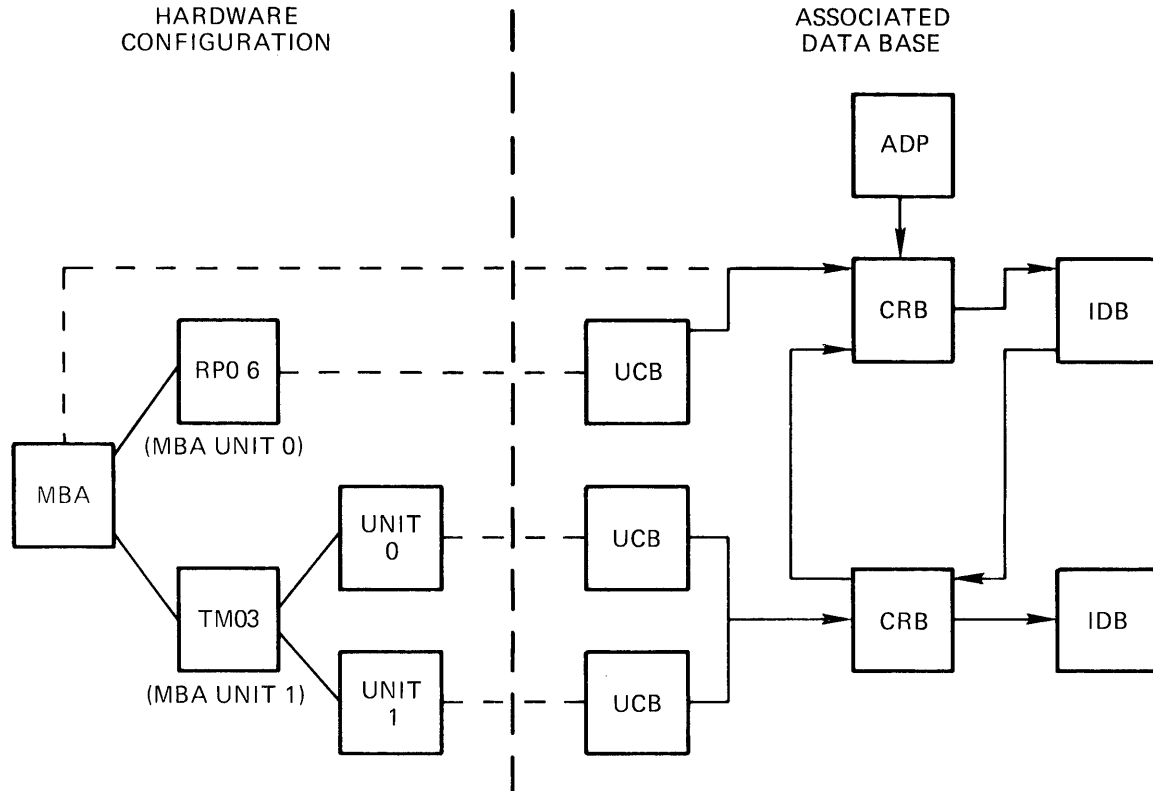


Figure F-5 I/O Data Base for MASSBUS Disk and Tape Units

VAX/VMS defines two levels of interrupt data blocks to dispatch interrupts from MASSBUS devices to the corresponding device driver. When an interrupt occurs, the VAX/VMS MBA interrupt dispatcher locates the channel request block for the MASSBUS adapter. The channel request block contains a pointer to the interrupt data block for the MASSBUS adapter.

The interrupt data block contains one entry for each controller attached to the MASSBUS. In the case of a single-unit controller, the interrupt data block contains a pointer to the unit control block for the device. Given the UCB address, the MBA interrupt dispatcher can restore the driver.

In the case of a solicited interrupt for a multiunit controller, however, the interrupt data block contains a pointer to the channel request block for the multiunit controller. The pointer addresses an instruction that transfers control to the controller's own interrupt service routine to determine which unit requested the interrupt. The second CRB, in turn, contains the pointer to the second interrupt data

MASSBUS ADAPTER

block. The interrupt data block contains a pointer to the unit control block for each unit attached to the multiunit controller. Figure F-6 illustrates the data base for the hardware configuration illustrated in Figure F-5.

If the field `IDB$L_OWNER` contains a zero (not filled), the VAX/VMS interrupt dispatcher also uses the MBA's Attention Summary register to determine the unit requesting the interrupt, as described in Section F.3.

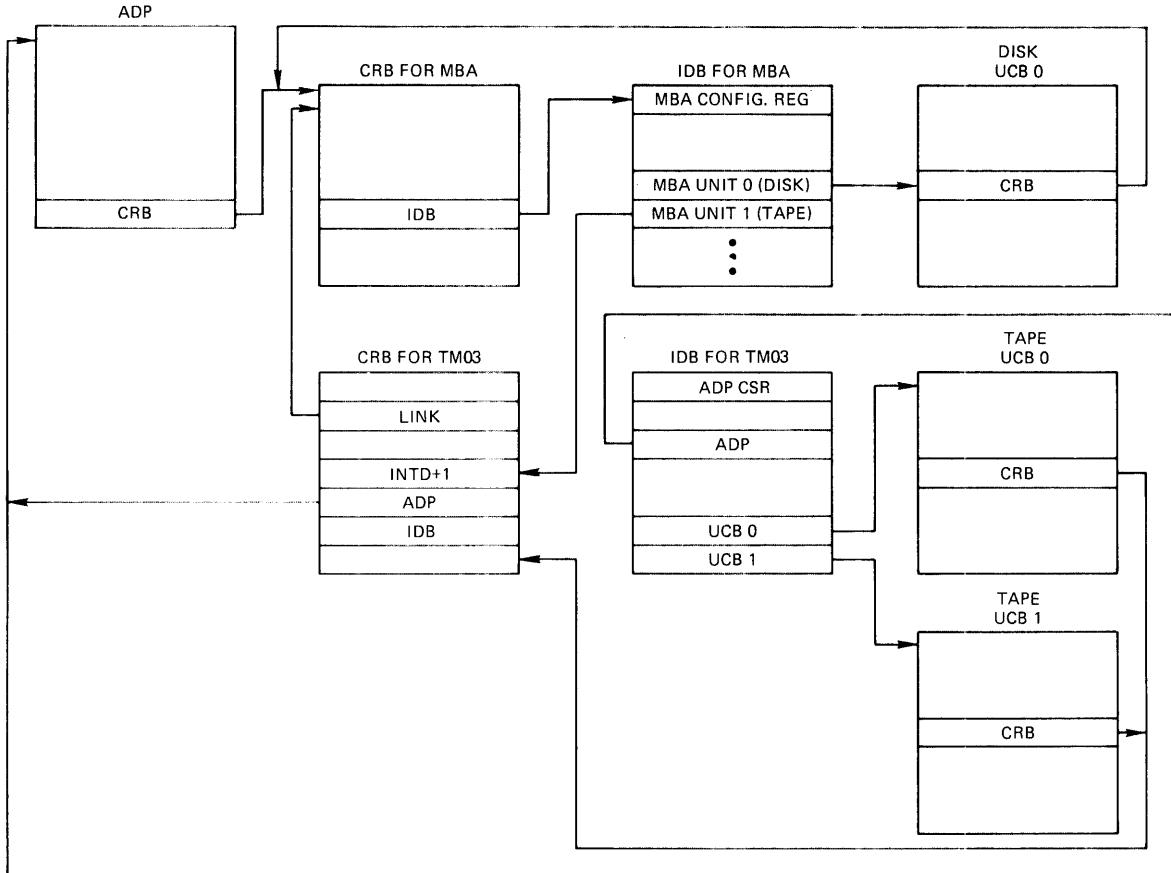


Figure F-6 I/O Data Structures Used in Dispatching an Interrupt

F.2 MBA CONSIDERATIONS FOR DRIVERS

MASSBUS adapter considerations affect a driver's device unit initialization routine, start I/O routines and, for subcontrollers only (magnetic tape), the driver's DPTAB macro. MBA considerations also affect interrupt handling as described in Section F.3.

F.2.1 Unit Initialization Routine

In order to perform unit initialization, the driver must refer to the device registers for the unit. The address of the unit initialization routine is specified in a field within the driver dispatch table

MASSBUS ADAPTER

(DDT\$L UNITINIT). For a single-unit controller, a driver obtains the information needed to refer to device registers in the following steps:

- Extracts the unit number from UCB\$W_UNIT and stores the unit number in UCB\$B_SLAVE
- Multiplies the slave number by 32 to derive the longword offset to the device registers for the drive and stores the result in UCB\$B_SLAVE+1; refer to the descriptions of these fields in Appendix A
- Assuming that the offset to the device registers is in R3, the driver loads the address of the device registers into a general register with the following instruction:

```
MOVAL    MBA$L_ERB(R4)[R3],R3
```

MBA\$L_ERB is a fixed offset to the start of the external registers.

R4 contains the address of the MBA configuration register. The configuration register is the first register in the MBA nexus space.

For a multiunit controller, a driver uses the following registers that were set up by the driver loading procedure:

```
R3 -- Address of the TM03 device registers
R4 -- Address of the MBA configuration register
R5 -- Address of unit control block
```

With this information, the driver locates the device registers in the following steps:

- Computes the MBA unit number of the TM03 controller by using R3 to determine the number of bytes from the start of the MBA external address space to the TM03 device registers and dividing the result by 128; stores the final result in UCB\$B_SLAVE
- Stores the drive offset constant (slave value multiplied by 32) in UCB\$B_SLAVE+1
- Performs initialization functions.

F.2.2 Start I/O Routine

The I/O data base contains the following information needed by a MASSBUS device driver start I/O routine:

- For a single-unit controller, the interrupt data block contains the address of the adapter's configuration register. For a multiunit controller, the interrupt data block contains the address of the controller's control/status register; that is, the first MBA external register for this controller.
- The unit control block contains the unit number in UCB\$B_SLAVE and the index to the address of the first device register in UCB\$B_SLAVE+1.

MASSBUS ADAPTER

The start I/O routine for a MASSBUS device performs the following basic functions:

- Requests controller data channel(s) as described in Section F.2.2.1
- Clears errors on the MASSBUS adapter by setting -1 in the MBA Status Register; this is a write-ones-to-clear register (MASSBUS device registers and MBA registers are a longword).
- Calls the LOADMBA macro to load map registers as described in Section F.2.2.2
- Sets up sector, track, and cylinder addresses (disk only)
- Clears drive errors and, if the medium is on line, starts the function
- Waits for device interrupt or timeout
- Releases controller data channel(s) as described in Section F.2.2.3
- Finishes the request like other drivers

F.2.2.1 Requesting a Controller Data Channel - For single-unit controllers, the MASSBUS adapter is the primary controller. For multiunit controllers, the subcontroller (device controller) is the primary controller, and the MBA is the secondary controller. Drivers for single-unit controllers must request the primary controller (MBA controller) data channel before they can load MBA map registers. Drivers request the primary controller channel by invoking the REQPCANL macro.

Drivers for units attached to a subcontroller must request both the primary controller data channel and the secondary controller data channel before they can load map registers. A tape driver requests both channels in the following steps:

- Invokes the REQPCANL macro to obtain the primary data channel
- Invokes the REQSCHANL or REQSCHANLH macro to obtain the secondary data channel

When a driver is performing a nontransfer function such as tape positioning operation, it does not require the MBA channel. The MBA channel is required only for data transfer operations.

F.2.2.2 Loading Map Registers - MASSBUS device drivers invoke the LOADMBA macro just prior to a transfer to load the MBA map registers. Drivers cannot modify these registers while a transfer is taking place.

LOADMBA expects the following register contents:

- R4 contains the address of the MBA configuration register.
- R5 contains the address of the unit control block

LOADMBA preserves the contents of R3. It uses R0 through R2.

MASSBUS ADAPTER

LOADMBA performs the following steps:

- Moves the negative value of the transfer byte count (UCB\$W_BCNT) into MBA\$L_BCR, which is the internal MBA byte counter register
- Moves the byte offset in the first page (UCB\$W_BOFF) of the transfer into MBA\$L_VAR, which is the internal MBA virtual address register
- Extracts a 21-bit page frame number from a page table entry, loads the page frame number into each map register needed, and sets the register's valid bit
- Loads a final map register as invalid so that a hardware fault does not modify memory
- Returns to the start I/O routine

F.2.2.3 Releasing Controller Data Channel(s) - A driver releases the controller data channels by invoking the RELCHAN macro. RELCHAN releases all controller channels (both primary and secondary) for the device. To release only the secondary controller channel, a driver can invoke the RELSCHAN macro.

F.2.3 DPTAB Macro

The device driver for a MASSBUS device attached to a subcontroller 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 interrupt data block to describe the subcontroller and to hold the address of the subcontroller's control/status register. It also causes creation of the second channel request block.

F.3 INTERRUPT HANDLING FOR MASSBUS DEVICES

The VAX/VMS MASSBUS interrupt handling routine (MBA\$INT) performs two functions:

- For single-unit controllers, it handles the interrupt and restores the driver in interrupt context at the instruction following the wait for interrupt
- For multiunit controllers, it calls the interrupt service routine for the subcontroller; for example, it calls the tape driver's interrupt service routine

MBA\$INT only preserves the contents of R2 through R5. Drivers wishing to use R0 and R1 must save and restore them.

MASSBUS ADAPTER

MBA\$INT handles interrupts in the following steps:

1. It obtains the address of the MBA interrupt data block from the stack.
2. From the interrupt data block, it obtains the field IDB\$L_OWNER. IDB\$L_OWNER contains either zeros or a UCB address. If it contains zeros, MBA\$INT proceeds as described in steps 1 through 4 of Section F.3.1.

If IDB\$L_OWNER contains a UCB address, it indicates the owner of the MBA controller.

3. MBA\$INT determines whether the device is expecting an interrupt. If it is not, MBA\$INT proceeds as described in steps 1 through 4 of Section F.3.1. If the interrupt status bit is set, MBA\$INT clears the bit to indicate that an interrupt has occurred.
4. If the interrupt is expected by the driver of the device that owns the channel, MBA\$INT takes the following steps:
 - a. Clears the Attention Summary bit for the MBA slave unit; that is, for the device or subcontroller that requested the interrupt. This bit is not cleared for subcontrollers; the drivers must clear it.
 - b. Obtains from the MBA IDB field IDB\$L_UCBLST (in which there are eight entries) the value stored for this device; this value is either a UCB address or the address of an instruction that transfers control to a subcontroller's interrupt service routine

If the value is not a UCB address, MBA\$INT proceeds as described in steps 1 through 3 of Section F.3.2.

5. If the value is a UCB address, MBA\$INT determines whether the interrupt is expected. If the interrupt is not expected, MBA\$INT proceeds as described in steps 1 through 3 of Section F.3.3.
6. If the interrupt is expected, MBA\$INT restores R3, R4, and PC and reactivates the driver. When the restored driver invokes IOFORK, control returns to this point.
7. MBA\$INT proceeds as described in steps 1 through 4 of Section F.3.1.

F.3.1 Looking for Another Request

Control transfers to this portion of MBA\$INT as a result of one of the following events:

- An interrupt was requested when no unit owned the MBA controller; that is, IDB\$L_OWNER was zero
- An interrupt occurred when the owner of the MBA controller was not expecting an interrupt
- When a driver has invoked IOFORK and that fork results in the execution of an RSB instruction that returns control to MBA\$INT

MASSBUS ADAPTER

MBA\$INT performs the following steps to dismiss the interrupt or handle the next request:

1. Clears the MBA status register
2. Examines the Attention Summary register for a device requesting attention
3. If no device is requesting attention, dismisses the interrupt
4. If a device is requesting attention, goes to step 4 above in Section F.3

The reason that MBA\$INT always checks the attention summary register when an interrupt service routine returns is to determine whether another device on the MASSBUS requested an interrupt while the MASSBUS owner device was transferring data or while this interrupt was being processed. Data transfer functions block the interrupts from nontransfer functions until the data transfer completes.

F.3.2 Transferring Control to a Subcontroller's Interrupt Service Routine

Control transfers to this portion of MBA\$INT when the device value for a MBA slave unit stored in IDB\$L_UCBLST is the address of an instruction that transfers control to a subcontroller's interrupt service routine, for example, a tape controller's interrupt service routine. MBA\$INT performs the following steps:

1. Moves the PSL onto the top of the stack
2. Executes a JSB instruction to the dispatch field of the subcontroller's CRB; the dispatch field contains a PUSH instruction that saves R2 through R5 and a JSB instruction to the subcontroller's interrupt service routine
3. The interrupt service routine executes, and after the driver forks, the interrupt service routine removes R2 through R5 from the stack and executes an REI instruction. The REI instruction removes the PSL and MBA\$INT's return address from the stack and returns control to MBA\$INT. MBA\$INT proceeds as described in steps 1 through 4 of Section F.3.1.

F.3.3 Handling Unsolicited Interrupts

When MBA\$INT finds that an unsolicited interrupt occurred (step 5 of Section F.3), it performs the following steps:

1. Obtains the address of the driver's unsolicited interrupt routine from the driver dispatch table
2. Calls the routine at that address
3. When the driver invokes IOFORK, MBA\$INT proceeds as described in steps 1 through 4 of Section F.3.1

GLOSSARY

ACP

See Ancillary Control Process.

adapter control block (ADP)

A structure in the I/O data base that describes either a UNIBUS or MASSBUS adapter.

ADP

See adapter control block.

allocate a device

To reserve a particular device unit for exclusive use. A user process can allocate a device only when that device is not allocated by any other process.

Ancillary Control Process (ACP)

A process that acts as an interface between user software and an I/O driver. An ACP provides functions supplemental to those performed in the driver, such as file and directory management. Three examples of ACPs are: the Files-11 ACP (F11ACP), the magnetic tape ACP (MTAACP), and the networks ACP (NETACP).

assign a channel

To establish the necessary software linkage between a user process and a device unit before a user process can communicate with that device. A user process requests the system to assign a channel and the system returns a channel number.

AST

See Asynchronous System Trap.

ASTLVL

See Asynchronous System Trap Level.

Asynchronous System Trap (AST)

A software-simulated interrupt to a user-defined service routine. ASTs enable a user process to be notified asynchronously with respect to its execution of the occurrence of a specific event. If a user process has defined an AST routine for an event, the system interrupts the process and executes the AST routine when that event occurs. When the AST routine exits, the system resumes the process at the point where it was interrupted.

Asynchronous System Trap Level (ASTLVL)

A value kept in an internal processor register that is the highest access mode for which an AST is pending. The AST does not occur until the current access mode drops in privilege (rises in numeric value) to a value greater than or equal to ASTLVL. Thus, an AST for an access mode will not be serviced while the processor is executing in a more privileged access mode.

base register

A general register used to contain the address of the first entry in a list, table, array, or other data structure.

buffered data path

A UNIBUS adapter data path that transfers 32 or 64 bits of data in a single SBI transfer. The UNIBUS adapter has 15 buffered data paths and one direct data path.

buffered I/O

See system buffered I/O.

bugcheck

The operating system's internal diagnostic check. The system logs the failure and crashes the system.

call instructions

The processor instructions CALLG (Call Procedure with General Argument List) and CALLS (Call Procedure with Stack Argument List).

CCB

See channel control block.

channel

A logical path connecting a user process to a physical device unit. A user process requests the operating system to assign a channel to a device so the process can communicate with that device. See also controller data channel.

channel control block (CCB)

A structure in the I/O data base maintained by the Assign I/O channel system service to describe the device unit to which a channel is assigned.

channel request block (CRB)

A structure in the I/O data base that describes the activity on a particular controller. The channel request block for a controller contains pointers to the wait queue of drivers ready to access a device through the controller.

configuration register

A control/status register for an adapter, for example a UNIBUS adapter. It resides in the adapter's I/O space.

connect-to-interrupt

A function by which a process connects to a device interrupt vector. To perform a connect-to-interrupt, the process must map to the program I/O space containing the vector.

console

The manual control unit integrated into the central processor. The console includes a serial line interface connected to a hard-copy terminal. This enables the operator to start and stop the system, monitor system operation, and run diagnostics.

console terminal

The hard-copy terminal connected to the central processor console.

context

The environment of an activity. See also process context, hardware context, and software context.

controller data channel

A logical path to which a driver for a device on a multiunit controller must be granted access before it can activate a device.

control/status register (CSR)

A control/status register for a device or controller. It resides in the processor's I/O space.

CRB

See channel request block.

CSR

See control/status register.

data base

- (1) All the occurrences of data described by a data base management system.
- (2) A collection of related data structures.

data structure

Any table, list, array, queue, or tree whose format and access conventions are well-defined for reference by one or more images.

DDB

See device data block.

DDT

See driver dispatch table.

device data block (DDB)

A structure in the I/O data base that identifies the generic device/controller name and driver name for a set of devices attached to the same controller.

device interrupt

An interrupt received on interrupt priority levels 20 through 23. Device interrupts can be requested only by devices, controllers, and memories.

device register

A location in device controller logic used to request device functions (such as I/O transfers) and/or report status.

device unit

One drive and its controlling logic, for example, a disk drive or terminal. Some controllers can have several device units connected to a single controller; for example, mass storage controllers.

diagnostic

A program that tests hardware, firmware, peripheral operation, logic, or memory and reports any faults it detects.

direct data path

A UNIBUS adapter data path that transfers 16 bits of data in a single SBI transfer. The UNIBUS adapter has one direct data path and 15 buffered data paths.

direct I/O

An I/O operation in which VAX/VMS locks the pages containing the associated buffer in physical memory for the duration of the I/O operation. The I/O transfer takes place directly from the process buffer. Contrast with system buffered I/O.

DPT

See driver prologue table.

drive

The electromechanical unit of a mass storage device system on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

driver

The set of code and tables that handles physical I/O operations to a device.

driver dispatch table (DDT)

A table in the I/O driver that lists the entry point addresses of standard driver routines and the sizes of diagnostic and error logging buffers for the device type.

driver fork level

The interrupt priority levels at which a driver fork process executes, that is, IPLs 8 through 11. Every unit control block indicates the driver fork level for its unit.

driver prologue table (DPT)

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

driver start I/O routine

See start I/O routine.

ECC

Error Correction Code.

error logger

A system process that empties the error log buffers and writes the error messages into the error file. Errors logged by the system include memory system errors, device errors and timeouts, and interrupts with invalid vector addresses.

exception

An event detected by the hardware or software (other than an interrupt or jump, branch, case, or call instruction) that changes the normal flow of instruction execution. An exception is always caused by the execution of an instruction or set of instructions (whereas an interrupt is caused by an activity in the system independent of the current instruction). There are three types of hardware exceptions: traps, faults, and aborts. Examples are: attempts to execute a privileged or reserved instruction, trace traps, compatibility mode faults, breakpoint instruction execution, and arithmetic traps.

executive

The generic name for the collection of procedures included in the operating system software that provide the basic control and monitoring functions of the operating system.

FDT

See function decision table.

FDT routines

Driver routines called by the Queue I/O Request system service to perform device-dependent preprocessing of an I/O request.

fork block

That portion of a unit control block that contains a driver's context while the driver is waiting for a resource. A driver awaiting the processor resource has its fork block linked into the fork queue.

fork dispatcher

A VAX/VMS interrupt service routine that is activated by a software interrupt at a fork interrupt priority level. Once activated, it dispatches driver fork processes from a driver fork queue until no processes remain in the queue for that IPL.

fork process

A fork process is a minimal context process that executes code under a series of constraints: it executes at raised interrupt priority levels; it uses R0 through R5 only (other registers must be saved and restored); it executes in system virtual address space; it is only allowed to refer to and modify static storage that is never modified by higher interrupt priority level code. VAX/VMS uses software interrupts and fork processes to synchronize executive operations.

fork queue

A queue of driver fork blocks that are awaiting activation at a particular IPL by the VAX/VMS fork dispatcher.

function code

See I/O function code.

function decision table (FDT)

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

function modifier

See I/O function modifier.

generic device name

A device name that identifies the type of device but not a particular unit; a device name in which the specific controller and/or unit number is omitted. When discussing device drivers, the generic device name contains neither the controller designation nor the unit number, for example, DB.

hardware context

The values contained in the following registers while a process is executing: the PC; the PSL; the 14 general registers (R0 through R13); the four processor registers (POBR, POLR, P1BR and P1LR) that describe the process virtual address space; the SP for the current access mode in which the processor is executing; plus the contents to be loaded in the SP for every access mode other than the current access mode. While a process is executing, its hardware context is continually being updated by the processor. While a process is not executing, its hardware context is stored in its hardware PCB.

hardware process control block (hardware PCB)

A data structure known to the processor that contains the hardware context when a process is not executing. A process's hardware PCB resides in its process header (PHD).

IDB

See interrupt data block.

interrupt

An event other than an exception or branch, jump, case, or call instruction that changes the normal flow of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs. See also device interrupt, software interrupt, and urgent interrupt.

interrupt data block (IDB)

A structure in the I/O data base that describes the characteristics of a particular controller and points to devices attached to that controller.

interrupt priority level (IPL)

The interrupt level at which a software or hardware interrupt is generated. There are 32 possible interrupt priority levels: IPL 0 is lowest, 31 is highest. The levels arbitrate contention for processor service. For example, a device cannot interrupt the processor if the processor is currently executing at an interrupt priority level greater than the interrupt priority level of the device's interrupt service routine.

interrupt service routine (ISR)

A routine executed when a device interrupt occurs.

interrupt stack (IS)

The system-wide stack used when executing in interrupt service context. At any time, the processor is either in a process context executing in user, supervisor, executive, or kernel mode, or in system-wide interrupt service context operating in kernel mode, as indicated by the interrupt stack and current mode bits in the PSL. The interrupt stack is not context switched.

interrupt stack pointer (ISP)

The stack pointer for the interrupt stack. Unlike the stack pointers for process context stacks, which are stored in the hardware PCB, the interrupt stack pointer is stored in an internal processor register.

interrupt vector

See vector.

I/O data base

A collection of data structures that describes I/O requests, controllers, device units, volumes, and device drivers in a VAX/VMS system. Examples are the driver dispatch table, driver prologue table, device data table, unit control block, channel request block, I/O request packet, and interrupt data block.

I/O driver

See driver.

I/O function

An I/O operation interpreted by the operating system and typically resulting in one or more physical I/O operations.

I/O function code

A 6-bit value specified in a Queue I/O Request system service that describes the particular I/O operation to be performed (such as, read, write, rewind).

I/O function modifier

A 10-bit value specified in a Queue I/O Request system service that modifies an I/O function code (for example, read terminal input no echo).

I/O lockdown

The state of a page such that it cannot be paged or swapped out of memory.

I/O request packet (IRP)

A structure in the I/O data base that describes an individual I/O request. The Queue I/O Request system service creates an I/O request packet for each I/O request. VAX/VMS and the driver of the target device use information in the I/O request packet to process the request.

I/O rundown

An operating system function in which the system cleans up any I/O in progress when an image exits.

I/O space

The regions of physical address space that contain the configuration registers, and device control/status and data registers. These regions are physically discontinuous.

I/O status block (IOSB)

A data structure associated with the Queue I/O Request system service. This service optionally returns a status code, number of bytes transferred, and device/function-dependent information in an I/O status block. The information returned is not returned from the service call, but filled in by VAX/VMS when the I/O request completes.

IPL

See interrupt priority level.

IRP

See I/O request packet.

ISP

See interrupt stack pointer.

ISR

See interrupt service routine.

limit

The size or number of given items requiring system resources (such as mailboxes, locked pages, I/O requests, or open files) that a job is allowed to have at any one time during execution, as specified by the system manager in the user authorization file. See also quota.

locking a page in memory

Making a page in an image ineligible for either paging or swapping. A page stays locked in physical memory until VAX/VMS specifically unlocks it.

logical I/O function

A set of I/O operations (for example, read and write logical block) that allow restricted direct access to device level I/O operations using logical block numbers.

mailbox

A software data structure that is treated as a record-oriented device for general interprocess communication. Communication using a mailbox is similar to other forms of device-independent I/O. Senders write to a mailbox; the receiver reads from that mailbox. Some system-wide mailboxes are defined: the error logger and OPCOM read from system-wide mailboxes.

MBA

MASSBUS Adapter.

offset

A fixed displacement from the beginning of a data structure. System offsets for items within a data structure normally have an associated symbolic name used instead of the numeric displacement. Where symbols are defined, programmers always reference the symbolic names for items in a data structure instead of using the numeric displacement.

page frame number (PFN)

The high-order 21 bits of the physical address of a page in physical memory.

page table entry (PTE)

The data structure that identifies the physical location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page frame number needed to map the virtual page to a physical page. When it is not in memory, the page table entry contains the information needed to locate the page on secondary storage (disk).

PCB

See Process Control Block.

PFN

See page frame number.

physical address

The address used by hardware to identify a location in physical memory or on directly-addressable secondary storage devices such as a disk. A physical memory address consists of a page frame number and the number of a byte within the page. A physical disk block address consists of a cylinder or track and sector number.

physical address space

The set of all possible 30-bit physical addresses that can be used to refer to locations in memory (memory space) or device registers (I/O space).

physical I/O functions

A set of I/O functions that allows access to all device level I/O operations except maintenance mode.

PID

See process identification.

process

The basic entity scheduled by the system software that provides the context in which an image executes. A process consists of an address space and both hardware and software context.

process context

The hardware and software contexts of a process.

process control block (PCB)

A data structure used to contain process context. The hardware PCB contains the hardware context. The software PCB contains the software context, which includes a pointer to the hardware PCB.

process identification (PID)

A 32-bit binary value that uniquely identifies a process. Each process has a process identification and a process name.

process I/O channel

See channel.

process page tables

The page tables used to describe process virtual memory.

process priority

The priority assigned to a process for scheduling purposes. The operating system recognizes 32 levels of process priority, where 0 is low and 31 high. Levels 16 through 31 are used for real-time processes. The system does not modify the priority of a real-time process (although the system manager or process itself may). Levels 0 through 15 are used for normal processes. The system may temporarily increase the priority of a normal process based on the activity of the process.

program section (psect)

A portion of a program with a given protection and set of storage management attributes. Program sections that have the same attributes are gathered together by the linker to form an image section.

PTE

See page table entry.

QIO

Queue I/O Request system service. The VAX/VMS system service that services \$QIO and \$QIOW requests. The Queue I/O Request system service prepares an I/O request for processing by the driver and performs device-independent preprocessing of the request. This system service also calls driver FDT routines.

quota

The total amount of a system resource, such as CPU time, that a job is allowed to use in an accounting period, as specified by the system manager in the user authorization file. See also limit.

return status code

See status code.

SBI

See Synchronous Backplane Interconnect.

small process

A system process that has no control region in its virtual address space and has an abbreviated context. Examples are the working set swapper and the null process. A small process is scheduled in the same manner as user processes, but must remain resident until it completes execution; that is, it cannot be swapped.

software context

The context maintained by VAX/VMS to describe a process. See software process control block (PCB).

software interrupt

An interrupt generated on interrupt priority level 1 through 15, which can be requested by software.

software process control block (software PCB)

The data structure used to contain a process's software context. The operating system defines a software PCB for every process when the process is created. The software PCB includes the following kinds of information about the process: current state; storage address if it is swapped out of memory; unique identification of the process; and address of the process header (which contains the hardware PCB). The software PCB resides in system region of virtual address space. It is not swapped with a process.

start I/O routine

The routine in a device driver that is responsible for obtaining necessary resources, for example, the controller data channel, and activating the device unit.

status code

A longword value that indicates the success or failure of a specific function. For example, system services always return a status code in R0 upon completion.

SVA

See system virtual address.

Synchronous Backplane Interconnect (SBI)

The part of the hardware that interconnects the processor, memory controllers, MASSBUS adapters, the UNIBUS adapter.

system buffered I/O

An I/O operation, such as terminal or mailbox I/O, in which an intermediate buffer from the system buffer pool is used instead of a process-specified buffer. Contrast with direct I/O.

System Page Table (SPT)

The data structure that maps the system virtual addresses, including the addresses used to refer to the process page tables. The SPT contains one PTE for each page of system virtual memory. The physical base address of the SPT is contained in a processor register called SBR.

system virtual address (SVA)

A virtual address identifying a location mapped to an address in system space.

timeout

The expiration of the time limit in which a device is to complete an I/O transfer. The driver's wait for interrupt request specifies the timeout limit.

timer

A system process that maintains the time of day and the date. It also scans for device timeouts and performs time-dependent scheduling upon request. The timer interrupt service routine creates the timer process.

UBA

UNIBUS Adapter.

UCB

See unit control block.

unit control block (UCB)

A structure in the I/O data base that describes the characteristics of and current activity on a device unit. The unit control block also holds the fork block for its unit's device driver; the fork block is a critical part of a driver fork process. The UCB also provides a static storage area for the driver.

unit initialization routine

The routine that readies controllers and device units for operation. Controllers and device units require initialization after a power fail and during the driver loading procedure.

urgent interrupt

An interrupt received on interrupt priority levels 24 through 31. These can be generated only by the processor for the interval clock, serious errors, and power fail.

vector

- (1) An interrupt or exception vector is a storage location known to the system that contains the starting address of a routine to be executed when a given interrupt or exception occurs. The system defines separate vectors for each interrupting adapter and for classes of exceptions. Each system vector is a longword.
- (2) For the purpose of exception handling, users can declare up to two software exception vectors (primary and secondary) for each of the four access modes. Each vector contains the address of a condition handler.
- (3) A one-dimensional array.

virtual I/O functions

A set of I/O functions that must be interpreted by an ancillary control process.

wait for interrupt request

A request made by a driver's start I/O routine after it activates a device. The request causes the driver fork process to be suspended until the device requests an interrupt or the device times out.

XDELTA

A tool for debugging operating systems and drivers.

INDEX

A

Aborting I/O after timeout, 12-5
Activating a fork process from
 fork queue, 5-14
Activating the device, 5-12, 9-2,
 9-6, 10-6
Adapter control block (ADP), 1-7,
 10-1, A-27
Address mapping for DMA transfers,
 4-2
ADP\$B_NUMBER, A-30
ADP\$B_TYPE, A-30
ADP\$L_CRB, A-31
ADP\$L_CSR, A-30
ADP\$L_DPQBL, A-31
ADP\$L_DPQFL, A-31
ADP\$L_INTD, A-32
ADP\$L_LINK, A-30
ADP\$L_MRQBL, A-32
ADP\$L_MRQFL, A-31
ADP\$L_VECTOR, A-31
ADP\$W_ADPTYPE, A-30
ADP\$W_DPBITMAP, A-32
ADP\$W_MRBITMAP, A-32
ADP\$W_SIZE, A-30
ADP\$W_TR, A-30
Allocation of an I/O request
 packet, 5-7
Allocation of map registers, 10-4
Assembling driver sources, 14-1
Assigning an I/O channel, 5-3
Asynchronous system traps (ASTs),
 kernel mode, 5-17
 user mode, 5-17
Autoconfiguration, 14-7
AUTOCONFIGURE command, 14-7

B

Base registers,
 setting, 15-11
Blocking interrupts, 9-6
Bootstrapping the system with
 XDELTA, 15-1
Breakpoints,
 clearing, 15-10
 displaying, 15-11
 inserting in the source code,
 15-3
 proceeding from, 15-11
 setting, 15-10
 setting complex, 15-13
Buffered data path, 4-5
 permanent allocation of, 10-3

Buffered data path, (Cont.)
 purging, 4-7, 10-7
 release of, 10-8
 requesting with no wait, 10-3
 requesting with wait, 10-2
Buffered I/O functions,
 selection of, 7-9
Buffered I/O operations, 1-15
 FDT routines for, 8-6
 implications for I/O post-
 processing, 8-7
 kernel mode AST for read, 8-8
Byte offset data transfers, 4-7

C

Calculating the base of driver
 code, 15-4
Cancel I/O on Channel system
 service, 13-4
Cancel I/O routine, 1-11, 13-4
 context for, 13-5
CASE macro, B-1
CCB\$B_AMOD, A-39
CCB\$B_STS, A-39
CCB\$L_DIRP, A-39
CCB\$L_UCB, A-39
CCB\$L_WIND, A-39
CCB\$W_IOC, A-39
Channel,
 process I/O channel vs. con-
 troller data channel, 5-5
 requesting the controller data
 channel, 9-3
Channel arbitration routine, 3-16
Channel assignment,
 process, 5-3
Channel control block (CCB),
 1-7, A-38
Channel request block (CRB), 1-6,
 5-5, A-22
 for MASSBUS devices, F-4
Checking for power failure, 9-6
Checking process I/O request
 quotas, 5-7
Clearing breakpoints, 15-10
Close and Display Next Location
 command, 15-9
Coding conventions, 6-1
Coding DMA transfers, 10-1
Coding driver tables, 7-1
Coding FDT routines, 8-1
Coding interrupt service routines,
 11-1
Coding start I/O routines, 9-1
COM\$DELATTNAST, C-1

INDEX

- COM\$DRVDEALMEM, C-2
 COM\$FLUSHATTNS, C-2
 Command files for booting with
 XDELTA,
 DBAXDT, 15-1
 DMAXDT, 15-1
 Competing for controller data
 channel, 3-15
 COM\$POST, 8-17, C-3
 COM\$SETATTNAST, C-4
 Completing the I/O request, 5-16,
 12-1
 Computing starting address of
 transfer, 10-6
 Computing transfer length, 9-5
 Configuration rules, 14-14
 CONNECT command, 14-3, 15-2
 Constraints for fork process
 execution, 3-14
 Context,
 for cancel I/O routine, 13-5
 for FDT routine execution, 8-1,
 8-2
 for fork process, 1-4
 for initialization routine,
 13-3
 for interrupt, 1-3, 11-3
 for start I/O routine, 9-1
 Controller data channel,
 competing for, 3-15
 releasing after transfer, 12-2
 requesting, 9-3
 requesting for MASSBUS device,
 F-8
 Controller initialization routine,
 13-1
 Conventions,
 coding, 6-1
 followed by FDT routines, 8-2
 register usage, 6-3
 register usage in I/O space,
 6-3
 terminology, xiii
 CRB\$B_MASK, A-24
 CRB\$B_TYPE, A-23
 CRB\$S_INTD, 7-2, 9-5, 10-3, 10-4,
 10-7, 13-2, 14-5, A-24
 CRB\$S_INTD+2, 11-3, A-24
 CRB\$S_INTD+4, 7-2
 CRB\$S_LINK, A-24
 CRB\$S_WQBL, A-23
 CRB\$S_WQFL, A-23
 CRB\$W_REFC, A-24
 CRB\$W_SIZE, A-23
 Creating a driver fork process
 for start I/O, 5-10
 CSR addresses,
 fixed and floating, 14-7
- D**
- Data channel, see controller data
 channel
 Data path, 4-3 to 4-8
 buffered data paths, 4-5
 byte offset data transfers, 4-7
 direct data path, 4-4
 longword-aligned 32-bit random
 access mode, 4-7
 purging buffered data paths,
 4-7
 DBAXDT command file, 15-1
 DDB\$B_TYPE, A-9
 DDB\$S_ACPD, A-9
 DDB\$S_DDT, 7-2, A-9
 DDB\$S_LINK, A-9
 DDB\$S_UCB, A-9
 DDB\$T_DRVNAME, A-9
 DDB\$T_NAME, A-9
 DDB\$W_SIZE, A-9
 DDTAB macro, 7-6
 DDT\$S_ALTSTART, A-35
 DDT\$S_CANCEL, A-34
 DDT\$S_FDT, A-33
 DDT\$S_REGDUMP, A-34
 DDT\$S_START, A-33
 DDT\$S_UNITINIT, A-35
 DDT\$S_UNSolINT, A-33
 DDT\$W_DIAGBUF, A-34
 DDT\$W_ERRORBUF, A-34
 Debugging a device driver, 15-1
 techniques, 15-14
 Defining device-specific function
 codes, 7-7
 DELTA debugging utility, 15-1
 commands, 15-14
 linking with user programs,
 15-14
 Destroying register contents,
 15-5
 Device activation, 2-6, 5-12,
 9-2, 9-6, 10-6
 Device activation bit mask, 9-5
 Device configuration, 14-9
 example of a UNIBUS config-
 uration, 14-15
 rules, 14-14
 Device data block (DDB), 1-6
 5-5, A-8
 Device dependence, 1-4
 Device-dependent cancel I/O
 routine, 13-5
 Device-dependent I/O post-
 processing, 12-3
 Device driver,
 functions of, 1-9

INDEX

- Device independence, 1-4
 - Device-independent cancel I/O routine, 13-5
 - Device interrupt,
 - delivering to a driver, 11-1
 - handling of, 5-13
 - responding to, 9-8
 - unsolicited, 11-5
 - waiting for, 9-6
 - Device IPLs, 3-6
 - Device registers,
 - incorrect references to, 15-15
 - opening with XDELTA, 15-15
 - reading and writing, 4-2
 - Device-specific function codes, 7-7
 - Device table for SYSGEN, 14-9
 - Device timeout,
 - wait for interrupt or, 9-6
 - Device timeout handler, 1-11, 12-4
 - Direct data path, 4-4
 - requesting a, 10-3
 - Direct I/O functions, 7-9
 - Direct I/O operations, 1-15
 - FDT routines for, 8-6
 - Direct memory access (DMA), 1-15
 - Direct memory access transfers, 4-1
 - Dispatching fork processes, 3-12
 - Displaying breakpoint list, 15-11
 - Display Range command, 15-9
 - DMA, see Direct memory access
 - DMAXDT command file, 15-1
 - DPTAB macro, 7-1
 - DPT\$B_ADPTYPE, A-36
 - DPT\$B_FLAGS, A-36
 - DPT\$B_REFC, A-36
 - DPT\$B_TYPE, A-36
 - DPT\$B_BLINK, A-36
 - DPT\$B_FLINK, A-36
 - DPT\$M_NOUNLOAD, 7-3
 - DPT\$M_SVP, 7-3
 - DPT_STORE macro, 7-3, B-2
 - DPT\$T_NAME, A-38
 - DPT\$W_INITTAB, A-37
 - DPT\$W_REINITTAB, A-37
 - DPT\$W_SIZE, A-36
 - DPT\$W_UCBSIZE, A-37
 - DPT\$W_UNLOAD, A-38
 - DPT\$W_VERSION, A-38
 - Driver debugging, 15-1
 - Driver dispatch table (DDT),
 - 7-1, A-33
 - creation of, 7-5
 - Driver fork IPLs (8 through 11), 3-7
 - Driver fork process, 1-3
 - context of, 9-1
 - creation for start I/O routine, 5-11
 - Driver linking, 14-1
 - Driver loading, 14-1, 15-2
 - initialization during, 13-2
 - Driver program sections, 14-1
 - Driver prologue table (DPT), 7-1, A-35
 - Driver routines,
 - cancel I/O routine, 1-11, 13-4
 - device timeout handler, 1-11, 12-4
 - error logging routine, 1-11, 13-6
 - FDT routines, 1-10, 8-1
 - initialization routines, 1-10, 13-1
 - interrupt service routine, 1-11, 11-1
 - I/O completion, 12-1
 - start I/O routine, 1-10, 9-1
 - Driver source assembly, 14-1
 - Driver tables,
 - coding of, 7-1
 - Driver template, 6-1, 6-5
 - DSBINT macro, 12-6, B-2
 - format of, 3-11
 - Dump routines (register), 13-6
- ## E
- ENBINT macro, 3-11, B-2
 - format of, 3-12
 - ERL\$DEVICERR, 13-6, C-6
 - ERL\$DEVICTMO, 13-6, C-6
 - ERL\$RELEASEMB, 12-3, C-6
 - Error logging routine, 1-11, 13-6
 - Examining the vector jump table, 15-4
 - Examining UCB, IPR, and PSL, 15-6
 - Example of bootstrapping the system with XDELTA, 15-2
 - Example of unsolicited interrupt handling, 11-6
 - Examples of table-generating macro invocations,
 - DDTAB macro, 7-6
 - DPTAB macro, 7-5
 - DPT_STORE macro, 7-5
 - FUNCTAB macro, 7-10
 - EXE\$ABORTIO, 8-6, 8-8, 8-13, C-7
 - EXE\$ALLOCBUF, C-8
 - use for buffered I/O, 8-6
 - EXE\$ALLOCI RP, C-9
 - EXE\$ALONONPAGED, C-9
 - EXE\$ALTQUEPK, 8-5, 8-17, C-9
 - EXE\$BUFFRQUOTA, C-10
 - EXE\$BUFQUOPRC, C-11
 - EXE\$CANCEL, 13-4

INDEX

Execute string command to XDELTA, 15-12
EXE\$DEANONPAGED, C-11
EXE\$FINISHIO, 8-5, 8-14, C-12
EXE\$FINISHIOC, 8-5, 8-6, 8-14, C-12
EXE\$FORK, C-12
EXE\$FORKDSPATH, C-13
EXE\$INSERTIRP, C-13
EXE\$INSIOQ, 8-16, C-14
EXE\$IOFORK, 10-7, 11-5, 15-6, C-15
 function of, 12-1
EXE\$MODIFY, C-16
EXE\$MODIFYLOCK, C-17
EXE\$MODIFYLOCKR, C-18
EXE\$ONEPARG, 8-9, C-20
EXE\$QIODRVPKT, 8-5, 8-13, C-20
EXE\$QIORETURN, C-21
EXE\$READ, 8-9, C-21
 use for buffered I/O, 8-4
EXE\$READCHK, 8-9, C-21
 use for buffered I/O, 8-6
EXE\$READCHKR, C-22
EXE\$READLOCK, 8-9, C-22
EXE\$READLOCKR, C-22
EXE\$SENSEMODE, 8-10, C-23
EXE\$SETCHAR, 8-11, C-24
EXE\$SETMODE, 8-11, C-25
EXE\$SNDEVMSG, C-26
EXE\$WRITE, 8-12, C-27
EXE\$WRITECHK, 8-12, C-27
 use for buffered I/O, 8-6
EXE\$WRITECHKR, C-27
EXE\$WRITELOCK, 8-12, C-28
EXE\$WRITELOCKR, C-29
EXE\$ZEROPARG, 8-13, C-29
EXIT command to DELTA, 15-14
Exiting from FDT routines, 8-4
External registers (MBA), F-1
FDT routines provided by VAX/VMS, 8-8
Floating CSR and vector address calculation, 14-14
Floating CSR and vector space, 14-8
Fork blocks, 1-3, 3-12
Fork dispatcher, 5-15
Fork dispatching, interrupt service routine for, 3-13
Fork IPL, 5-15
FORK macro, B-2
Fork process, activation from a fork queue, 5-15
 definition of, 1-3
 execution constraints, 3-14
 transferring control to, 3-4
Fork process context, 5-14
Fork process dispatching, 3-13
Fork queues, 1-8, 5-15
FUNCTAB Macro, 7-10, B-2
Function codes, definition of device-specific codes, 7-8
Function decision table (FDT), 1-2, 5-9, 7-1, 7-7
Functions of a device driver, 1-9
Functions of Queue I/O Request system service, 5-1

H

Hardware device IPLs, 3-8
Hardware interrupts, servicing of, 3-4
Hardware IPLs, 3-2

F

FDT bit mask, 5-9
FDT entry, 5-9
FDT processing, 5-8
FDT routines, 1-10, 9-1
 coding of, 8-1
 conventions followed by, 8-2
 execution context, 8-1
 exit methods, 8-5
 registers preset for, 8-1
 transferring to and from, 8-3
FDT routines for buffered I/O, 8-6
FDT routines for direct I/O, 8-6

I

IDB\$B_TYPE, A-28
IDB\$W_UNITS, A-29
IDB\$L_ADAP, A-29
IDB\$L_CSR, 9-3, A-28
IDB\$L_OWNER, 9-4, 9-7, 11-4, 13-1, A-28, F-10
IDB\$L_UCBLST, A-29, F-10
IDB\$W_SIZE, A-28
IFNOWRT macro, B-2
IFNORD macro, B-2
IFRD macro, B-3
Incorrect references to device registers, 15-15

INDEX

- Indirect command, 15-9
- INI\$BRK, 15-3
- Initialization data, 7-2
- Initialization during driver loading, 13-2
- Initialization during powerfail recovery, 13-2
- Initialization routines, 1-10, 13-1
 - execution context for, 13-3
- Inserting breakpoints in driver code, 15-3
- Internal registers (MBA), F-1
- Interrupt,
 - delivering to a driver, 11-1
 - requesting with XDELTA, 15-4
 - responding to, 9-8
 - solicited, 11-4
 - unsolicited, 11-5
 - waiting for an, 5-13, 9-6
- Interrupt context, 1-3, 5-13, 11-3
 - switching to fork process context from, 5-13
- Interrupt data block (IDB), 1-6, 5-5, A-27
 - for MASSBUS devices, F-5, F-6
- Interrupt handling, 2-7, 11-4
- Interrupt priority levels (IPLs), 1-7
 - conventions used during I/O postprocessing, 3-10
 - defined by VAX/VMS, 3-1
 - for device interrupts, 3-7
 - for driver fork processes, 3-7
 - hardware IPLs, 3-2
 - IPL\$ ASTDEL, 3-6
 - IPL\$ IOPOST, 3-7
 - IPL\$ MAILBOX, 3-9
 - IPL\$ POWER, 3-7
 - IPL\$ QUEUEAST, 3-8
 - IPL\$ SCHED, 3-8
 - IPL\$ SYNCH, 3-8
 - IPL\$ TIMER, 3-8
 - IPL\$ XDELTA, 3-9
 - lowering IPL, 3-3
 - modification of, 3-11
 - overview of use, 3-9
 - raising IPL, 3-3
 - software IPLs, 3-1
 - used during I/O processing, 3-6
- Interrupts,
 - blocking, 9-5
 - for MASSBUS devices, F-9
 - handling device interrupts, 5-14
- Interrupt service routine for fork dispatching, 3-13
- Interrupt service routines, 1-11, 3-2, 11-1
- IOC\$ALOUBAMAP(N), 10-4, C-30
- IOC\$ALTUBAMAP, C-31
- I/O cancellation routine, 13-4
- IOC\$APPLYECC, C-32
- IOC\$CANCELIO, 13-5, C-33
- IOC\$DIAGBUFILL, C-33
- IOC\$INITIATE, 8-16, C-35
- IOC\$IOPOST, C-36
- IOC\$LOADUBAMAP(A), 10-5, 10-6, C-37
- I/O completion, see I/O post-processing
- IOC\$PURGDATAP, C-38
- IOC\$RELCHAN, C-39
- IOC\$RELDATAP, 10-8, C-40
- IOC\$RELMAPREG, 10-8, C-41
- IOC\$RELSCHAN, C-42
- IOC\$REQCOM, C-42
 - function of, 12-3
- IOC\$REQDATAP(NW), 10-2, C-43
- IOC\$REQMAPREG, 10-4, C-44
- IOC\$REQPCHANH, C-45
- IOC\$REQPCHANL, C-46
- IOC\$REQSCHANH, C-47
- IOC\$REQSCHANL, 15-6, C-47
- IOC\$RETURN, 13-5, C-47
- IOC\$WFIKPC, 9-7, C-47
- IOC\$WFIRLCH, C-48
- I/O data base, 1-5, A-1
 - control blocks in, 1-6
 - for MASSBUS devices, F-4
 - locating a driver in, 5-3
- IOFORK macro, 10-7, 11-4, B-3, F-10, F-11
 - functions of, 5-14
- I/O function,
 - validation of, 5-7
- I/O function code, 5-9
 - conversion, 9-4
 - obtaining, 9-4
- I/O function modifier, 5-9
- I/O operations,
 - buffered vs. direct, 1-15
- I/O postprocessing, 5-17
 - by driver, 2-8, 12-1
 - by VAX/VMS, 2-8, 12-3
 - implications for buffered I/O, 8-7
- I/O postprocessing dispatcher, 5-17
- I/O preprocessing, 5-1
 - by Queue I/O Request system service, 2-3
 - by the driver, 2-4
- I/O request packet (IRP), 1-7, 5-16, A-1
 - allocation of, 5-7

INDEX

I/O request packet extension
 (IRPE), A-39
 I/O status block,
 validation of, 5-7
 IPL\$ ASTDEL, 3-6, 8-3, 8-13
 IPL\$ IOPOST, 3-7, 8-14
 IPL\$ MAILBOX, 3-9
 IPL\$ POWER, 3-7, 9-1, 9-5,
 12-6, 13-2
 IPL\$ QUEUEAST, 3-8
 IPL\$ SCHED, 3-8
 IPL\$ SYNCH, 3-8
 IPL\$ TIMER, 3-8
 IPL\$ XDELTA, 3-9
 IRP\$B CARCON, 8-9, 8-12, A-7
 IRP\$B EFN, A-4
 IRP\$B PRI, A-4
 IRP\$B RMOD, A-3
 IRP\$B TYPE, A-3
 IRPE\$B TYPE, A-41
 IRPE\$L BCNT1, A-41
 IRPE\$L BCNT2, A-41
 IRPE\$L SVAPTE1, A-41
 IRPE\$L SVAPTE2, A-41
 IRPE\$W BOFF1, A-41
 IRPE\$W BOFF2, A-41
 IRPE\$W SIZE, A-41
 IRPE\$W STS, A-41
 IRP\$L ARB, A-8
 IRP\$L AST, A-3
 IRP\$L ASTPRM, A-3
 IRP\$L DIAGBUF, A-7
 IRP\$L EXTEND, A-7
 IRP\$L IOQBL, A-3
 IRP\$L IOQFL, A-3
 IRP\$L IOSB, 8-14, A-5
 IRP\$L IOST1, A-6
 IRP\$L IOST2, A-7
 IRP\$L MEDIA, 8-5, 8-9, 8-11,
 8-13, 8-15, 12-3, A-6
 IRP\$L MEDIA+4, 8-5, 8-11, 8-15,
 12-3, A-7
 IRP\$L PID, 13-5, A-3, C-14
 IRP\$L SEGVBN, A-7
 IRP\$L SEQNUM, A-7
 IRP\$L SVAPTE, 8-7, 8-8, 8-10,
 9-2, A-6
 IRP\$L UCB, A-4
 IRP\$L WIND, A-4
 IRP\$W ABCNT, A-7
 IRP\$W BCNT, 8-6, 8-8, 8-9, 9-2,
 A-6
 IRP\$W BOFF, 8-7, 8-8, 8-16,
 9-2, A-6
 IRP\$W CHAN, 13-5, A-5
 IRP\$W FUNC, 8-9, 9-4, A-4
 IRP\$W OBCNT, A-7
 IRP\$W SIZE, A-3
 IRP\$W STS, 8-6, 8-8, 8-9, A-5

J

JIB\$L BYTCNT, 8-7
 Job information block, 8-7

K

Kernel mode AST, 5-17
 for buffered I/O read opera-
 tions, 8-8

L

Linking a driver, 14-1
 LOAD command, 14-2, 15-2
 Loading a driver, 1-16, 14-1,
 15-2
 Loading PC and Continuing
 (XDELTA), 15-11
 Loading MBA map registers, F-8
 Loading UBA map registers, 9-4,
 10-5
 LOADMBA macro, F-8
 LOADUBA macro, 10-5, B-3
 Longword-aligned 32-bit transfer
 mode, 4-7
 Lowering IPL, 3-3

M

Macros
 CASE, B-1
 DDTAB, 7-6, B-1
 DPTAB, 7-2, B-1
 DPT STORE, 7-3, B-2
 DSBINT, 3-11, B-2
 ENBINT, 3-12, B-2
 FORK, B-2
 FUNCTAB, 7-10, B-2
 IFNORD, B-2
 IFNOWRT, B-2
 IFRD, B-3
 IOFORK, B-3
 LOADUBA, B-3
 PURDPR, 1-7
 RELCHAN, B-3
 RELDPR, B-3
 RELMPR, B-3
 RELSCHAN, B-3
 REQCOM, B-3
 REQDPR, B-3
 REQMPR, B-3

INDEX

- Macros, (Cont.)
REQPCHAN, B-3
REQSCHAN, B-3
SAVIPL, B-4
SETIPL, 3-11, B-4
SOFTINT, 3-12, B-4
WFIKPCH, 9-7, B-4
WFIRLCH, 9-7, B-4
- Mapping addresses for DMA transfers, 4-2
- Map registers, 4-2
allocation of, 10-4
loading, 9-5, 10-5
loading for MBA transfers, F-8
permanent allocation of, 10-4
releasing, 10-8
- Map registers for MBA, F-2
- MASSBUS adapter, F-1
- MASSBUS device interrupt handling, F-9
- MASSBUS devices,
I/O data base for, F-4
- MBA\$INT, F-9 to F-11
- MBA map registers, F-2
- MBA registers,
external, F-1
internal, F-1
- Message to operator, 12-6
- Mixed direct and buffered data path transfers, 10-3
- MMG\$IOLOCK routine, 8-10, 8-12
- Modifying IPL, 3-11
- O**
- Obtaining the I/O function code, 9-4
- Open and Display command, 15-8
- Opening device registers in XDELTA, 15-15
- Operator,
sending a message to, 12-6
- Options file, 14-1
- P**
- Powerfail recovery,
initialization during, 13-2
- Power failure,
checking for, 9-6
- Preprocessing an I/O request, 5-1
- Proceeding from breakpoints, 15-11
- Process channel assignment, 5-3
- Process context, 1-3
- Process context conventions for FDT routines, 8-3
- Programmed I/O, 1-15
- Program sections in drivers, 14-1
- PSL (program status longword), 15-6
- PURDPR macro, 10-7
- Purging buffered data paths, 4-7, 10-7
- Q**
- Queue I/O Request system service, functions of, 5-1
- Queuing an I/O request packet, 2-5
- R**
- Raising IPL, 3-3
- Reading device registers, 4-2
- References to system addresses, 15-15
- REGDMP macro, 13-6
- Register content,
destruction of, 15-5, 15-6
- Register conventions for FDT routines, 8-2
- Register dump routine, 13-6
- Registers preset for FDT routines, 8-1
- Register use conventions,
for FDT routines, 6-3
for other driver routines, 6-3
in I/O space, 6-3
- Reinitialization data, 7-2
- RELCHAN macro, B-3
- RELDPR macro, 10-8, B-3
- Releasing a buffered data path, 10-8
- Releasing MBA map registers, F-9
- Releasing the controller, 12-2
- Releasing UBA map registers, 10-8
- RELMPR macro, 10-8, B-3
- RELOAD command, 14-5
- RELSCHAN macro, B-3
- REQCOM macro, 12-3, B-3
- REQDPR macro, B-3
- REQMPR macro, B-3
- REQPCHAN macro, B-3
- REQSCHAN macro, B-3

INDEX

- Requesting a buffered data path, 10-3
 - Requesting a direct data path, 10-3
 - Requesting an XDELTA interrupt, 15-4
 - Requesting the controller data channel, 9-2
 - Resource wait queues, 1-8, 3-14
 - Responding to a device interrupt, 9-8
 - Restrictions on register use in I/O space, 6-3
 - Retrying I/O operations, 12-4
 - Routines provided by VAX/VMS
 - COM\$DELATTNAST, C-1
 - COM\$DRVDEALMEM, C-2
 - COM\$FLUSHATTNS, C-2
 - COM\$POST, C-3
 - COM\$SETATTNAST, C-4
 - ERL\$DEVICERR, C-6
 - ERL\$DEVICTMO, C-6
 - ERL\$RELEASEMB, C-6
 - EXE\$ABORTIO, 8-13, C-7
 - EXE\$ALLOCBUF, C-8
 - EXE\$ALLOCIRP, C-9
 - EXE\$ALONONPAGED, C-9
 - EXE\$ALTQUEPKT, 8-5, 8-17, C-9
 - EX\$BUFFERQUOTA, C-10
 - EXE\$BUFQUOPRC, C-11
 - EXE\$DEANONPAGED, C-11
 - EXE\$FINISHIO, 8-14, C-12
 - EXE\$FINISHIOC, 8-14, C-12
 - EXE\$FORK, C-12
 - EXE\$FORKDSPH, C-13
 - EXE\$INSERTIRP, C-13
 - EXE\$INSIOQ, C-14
 - EXE\$IOFORK, 12-1, C-15
 - EXE\$MODIFY, C-16
 - EXE\$MODIFYLOCK, C-17
 - EXE\$MODIFYLOCKR, C-18
 - EXE\$ONEPARM, 8-9, C-20
 - EXE\$QIODRVPKT, 8-15, C-20
 - EXE\$QIORETURN, C-21
 - EXE\$READ, 8-9, C-21
 - EXE\$READCHK, C-21
 - EXE\$READCHKR, C-22
 - EXE\$READLOCK, C-22
 - EXE\$READLOCKR, C-22
 - EXE\$SENSEMODE, 8-10, C-23
 - EXE\$SETCHAR, 8-11, C-24
 - EXE\$SETMODE, 8-11, C-25
 - EXE\$SNDEVMSG, C-26
 - EXE\$WRITE, 8-12, C-27
 - EXE\$WRITECHK, C-27
 - EXE\$WRITECHKR, C-27
 - EXE\$WRITELOCK, C-28
 - EXE\$WRITELOCKR, C-29
 - EXE\$ZEROPARM, 8-13, C-29
 - IOC\$ALOUBAMAP(N), C-30
 - Routines provided by VAX/VMS, (Cont.)
 - IOC\$ALTUBAMAP, C-31
 - IOC\$APPLYECC, C-32
 - IOC\$CANCELIO, 13-5, C-33
 - IOC\$DIAGBUFILL, C-33
 - IOC\$INITIATE, C-35
 - IOC\$IOPOST, C-36
 - IOC\$LOADUBAMAP(A), C-37
 - IOC\$PURGDATAP, 10-7, C-38
 - IOC\$RELCHAN, C-39
 - IOC\$RELDATAP, C-40
 - IOC\$RELMAPREG, C-41
 - IOC\$RELSCHAN, C-42
 - IOC\$REQCOM, 12-3, C-42
 - IOC\$REQDATAP(NW), C-43
 - IOC\$REQMAPREG, C-44
 - IOC\$REQPCHANH, C-45
 - IOC\$REQPCHANL, C-46
 - IOC\$REQSCHANH, C-47
 - IOC\$REQSCHANL, C-47
 - IOC\$RETURN, C-47
 - IOC\$WFIKPCH, 9-7, C-47
 - IOC\$WFIRLCH, C-48
 - Rules for device configurations, 14-14
 - Running SYSGEN, 14-2
- ## S
- SAVIPL macro, B-4
 - SBI addresses,
 - mapping to UNIBUS, 4-2
 - Sending a message to the operator, 12-6
 - SETIPL macro, B-4
 - format of, 3-11
 - Setting an XDELTA base register, 15-5, 15-11
 - Setting base registers, 15-11
 - Setting breakpoints, 15-10
 - Setting complex breakpoints, 15-13
 - SHOW/DEVICE command, 14-6, 15-3
 - Show Value command, 15-10
 - SOFTINT macro, B-4
 - format of, 3-12
 - Software IPLs, 3-1
 - Solicited interrupt,
 - servicing an, 11-4
 - Start I/O routine, 1-10, 5-12
 - coding an, 9-1
 - execution context, 9-1
 - for MASSBUS device, F-7
 - Status,
 - saving, 12-3
 - Step Instruction command, 15-10
 - Synchronization, 1-7
 - fork queues, 1-8

INDEX

Synchronization, (Cont.)
 interrupt priority levels, 1-7
 resource wait queues, 1-8
 SYSGEN, 14-2
 SYSGEN commands,
 AUTOCONFIGURE, 14-7
 CONNECT, 14-3
 LOAD, 14-2
 RELOAD, 14-5
 SHOW/DEVICE, 14-6
 SYSGEN device table, 14-9
 SYSGEN's autoconfiguration, 14-8
 System buffer,
 allocation of, 8-6

T

Tables in drivers, 7-1
 Template for a driver, 6-1,
 6-5
 Timeouts,
 waiting for interrupt or, 9-6
 Transfer length,
 computation of, 9-5
 Transferring control to a driver
 fork process, 3-4
 Transferring to and from FDT
 routines, 8-3

U

UBA interrupt service routines,
 5-13
 UBA map registers,
 loading, 10-5
 releasing, 10-8
 UCB\$B_AMOD, A-17
 UCB\$B_CEX, A-21
 UCB\$B_DEVCLASS, 8-11, A-15
 UCB\$B_DEVTYPE, 8-11, A-16
 UCB\$B_DIPL, 7-2, 12-4, A-17
 UCB\$B_ERTCNT, 12-3, A-20, C-15
 UCB\$B_ERTMAX, A-20
 UCB\$B_FEX, A-21
 UCB\$B_FIPL, 7-2, 8-16, 12-2,
 A-12
 UCB\$B_OFFNDX, A-22
 UCB\$B_OFFRTC, A-22
 UCB\$B_SLAVE, A-21, F-7
 UCB\$B_SPR, A-21
 UCB\$B_TYPE, A-12
 UCB disk extension, A-21
 UCB error log extension, A-20
 UCB\$L_AMB, A-17
 UCB\$L_CRB, A-13
 UCB\$L_DDB, A-14

UCB\$L_DEVCHAR, 7-2, A-14
 UCB\$L_DEVDEPEND, 8-10, A-16
 UCB\$L_DPC, A-21
 UCB\$L_DUETIM, 12-4, A-18
 UCB\$L_EMB, 12-3, A-21
 UCB\$L_FPC, 11-4, 12-2, 12-4,
 A-12
 UCB\$L_FQBL, A-12
 UCB\$L_FQFL, A-12
 UCB\$L_FR3, 11-4, 12-4, A-13
 UCB\$L_FR4, 11-4, 12-4, A-13
 UCB\$L_IOQBL, A-17
 UCB\$L_IOQFL, A-16
 UCB\$L_IRP, 8-16, 12-3, A-17
 UCB\$L_LINK, A-14
 UCB\$L_MAXBLOCK, A-22
 UCB\$L_MEDIA, A-22
 UCB\$L_OPCNT, A-18
 UCB\$L_OWNUIC, A-13
 UCB\$L_PID, A-14
 UCB\$L_SVAPTE, 9-5, 10-5, 12-5,
 A-19
 UCB\$L_SVPN, 7-3, A-19
 UCB\$L_VCB, A-14
 UCB\$W_BCNT, 9-5, 10-4, 10-5,
 12-5, A-19
 UCB\$W_BCR, A-22
 UCB\$W_BOFF, 9-5, 10-4 to 10-6,
 12-5, A-19, F-9
 UCB\$W_BUFQUO, A-13
 UCB\$W_CHARGE, A-17
 UCB\$W_DEVBUSIZ, 8-11, A-16
 UCB\$W_DEVSTS, 12-3, A-18
 UCB\$W_DIRSEQ, A-22
 UCB\$W_EC1, A-22
 UCB\$W_EC2, A-22
 UCB\$W_ERRCNT, A-20
 UCB\$W_FUNC, A-21
 UCB\$W_OFFSET, A-22
 UCB\$W_REFC, 11-6, A-17
 UCB\$W_SIZE, A-12
 UCB\$W_STS, A-17
 UCB\$W_UNIT, A-17, F-7
 UCB\$W_VPROT, A-13
 UNIBUS, 4-1, 10-1
 UNIBUS adapter, 4-1, 10-1
 UNIBUS device configuration,
 an example, 14-15
 Unit control block (UCB), 1-6,
 5-3, 15-6, A-10
 Unit initialization routines,
 13-1
 for MASSBUS devices, F-6, F-7
 Unsolicited interrupt handling,
 11-5
 and busy devices, 11-5
 example of, 11-6
 User buffer,
 validation of user buffer for
 buffered I/O, 8-6

INDEX

V

Validating an I/O function, 5-7
Validating the I/O status block,
5-7
Vector jump table,
examining the, 15-5

W

Waiting for an interrupt, 2-6,
5-13, 9-6
Wait queues,
for resources, 3-14
WFIKPCH macro, 12-6, B-4
format of, 9-7
function of, 9-7
WFIRLCH macro, B-4
format of, 9-7
Writing device registers, 4-2

X

XDELTA,
and system failures, 15-5
booting the system with, 15-1

XDELTA base registers,
setting, 15-5
XDELTA commands,
Clearing Breakpoints, 15-10
Close and Display Next Location,
15-9
Displaying Breakpoint List,
15-11
Display Previous, 15-9
Display Range, 15-9
Indirect, 15-9
Loading PC and Continuing,
15-11
Open and Display, 15-8
Proceeding from Breakpoint,
15-11
Setting Base Registers, 15-11
Setting Breakpoints, 15-10
Setting Complex Breakpoints,
15-13
Show Value, 15-10
Step Instruction, 15-10
summary of, 15-7
XDELTA display mode control,
15-12
XDELTA interrupt,
requesting an, 15-4
XDELTA operators, 15-8
XDELTA special symbols, 15-8
XDELTA stored commands, 15-13
XDELTA values and expressions,
15-7

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 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

Please cut along this line.

Do Not Tear - Fold Here and Tape

digital



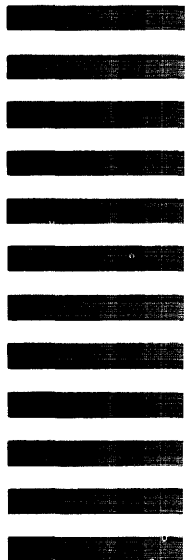
No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876



Do Not Tear - Fold Here