

VMS Device Support

Order Number: AA-LA88A-TE

April 1988

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

Revision/Update Information: This book supersedes the *Guide to Writing a Device Driver for VAX/VMS*, published April, 1986.

Operating System and Version: VMS Version 5.0

Software Version: VMS Version 5.0

**digital equipment corporation
maynard, massachusetts**

April 1988

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

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


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

Copyright ©1988 by Digital Equipment Corporation

All Rights Reserved.
Printed in U.S.A.

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

The following are trademarks of Digital Equipment Corporation:

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

ZK4490

**HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS**

USA & PUERTO RICO*

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

CANADA

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

INTERNATIONAL

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

In Continental USA, Puerto Rico, Alaska, and Hawaii call 800-DIGITAL.

In Canada call 800-267-6215.

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

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

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by DIGITAL. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use DIGITAL-supported devices, such as the LN03 laser printer and PostScript[®] printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

Contents

| | |
|---------|------|
| PREFACE | xxix |
|---------|------|

| | |
|--------------------------|--------|
| NEW AND CHANGED FEATURES | xxxiii |
|--------------------------|--------|

PART I THE VMS DEVICE DRIVER ENVIRONMENT

| | |
|--|-----|
| CHAPTER 1 INTRODUCTION TO DEVICE DRIVERS | 1-1 |
|--|-----|

| | |
|----------------------|-----|
| 1.1 DRIVER FUNCTIONS | 1-2 |
|----------------------|-----|

| | |
|-----------------------|-----|
| 1.2 DRIVER COMPONENTS | 1-2 |
|-----------------------|-----|

| | |
|---------------------|-----|
| 1.2.1 Driver Tables | 1-2 |
|---------------------|-----|

| | |
|-----------------------|-----|
| 1.2.2 Driver Routines | 1-3 |
|-----------------------|-----|

| | |
|----------------------|-----|
| 1.3 THE I/O DATABASE | 1-4 |
|----------------------|-----|

| | |
|---------------------|-----|
| 1.3.1 Driver Tables | 1-4 |
|---------------------|-----|

| | |
|-----------------------|-----|
| 1.3.2 Data Structures | 1-5 |
|-----------------------|-----|

| | |
|---------------------------|-----|
| 1.3.3 I/O Request Packets | 1-6 |
|---------------------------|-----|

| | |
|--|-----|
| 1.4 SYNCHRONIZATION OF DRIVER ACTIVITY | 1-7 |
|--|-----|

| | |
|--------------------|-----|
| 1.5 DRIVER CONTEXT | 1-7 |
|--------------------|-----|

| | |
|---|-----|
| 1.5.1 Example of Driver Context-Switching | 1-8 |
|---|-----|

| | |
|-----------------------------|-----|
| 1.6 HARDWARE CONSIDERATIONS | 1-9 |
|-----------------------------|-----|

| | |
|---|------|
| 1.6.1 Driver Dependency on VAX Processing Systems | 1-10 |
|---|------|

| | |
|---|------|
| 1.6.1.1 VAX-11/780, VAX-11/785, and VAX 8600/8650/8670 | 1-10 |
|---|------|

| | |
|--------------------|------|
| 1.6.1.2 VAX-11/750 | 1-11 |
|--------------------|------|

| | |
|-----------------------------------|------|
| 1.6.1.3 VAX-11/730 and VAX-11/725 | 1-12 |
|-----------------------------------|------|

| | |
|---|------|
| 1.6.1.4 VAX 8200/8250/8300/8350, VAX 8530/8550/8700/8800/8830/8840, and VAX 6200 Series | 1-12 |
|---|------|

| | |
|--|------|
| 1.6.1.5 MicroVAX 3600 Series and MicroVAX II | 1-15 |
|--|------|

| | |
|--------------------|------|
| 1.6.1.6 MicroVAX I | 1-16 |
|--------------------|------|

Contents

| | | |
|--|---|------------|
| 1.7 | PROGRAMMED-I/O AND DIRECT-MEMORY-ACCESS TRANSFERS | 1-16 |
| 1.7.1 | Programmed I/O | 1-17 |
| 1.7.2 | Direct-Memory-Access I/O | 1-17 |
| 1.8 | BUFFERED AND DIRECT I/O | 1-18 |
| 1.9 | EXAMPLE OF AN I/O REQUEST | 1-19 |
| CHAPTER 2 DISCUSSION OF A \$QIO REQUEST | | 2-1 |
| 2.1 | DRIVER CODE FOR THE LP11 WRITE FUNCTION | 2-1 |
| 2.2 | A USER PROCESS'S I/O REQUEST | 2-2 |
| 2.3 | DEVICE-INDEPENDENT I/O PREPROCESSING BY VMS | 2-3 |
| 2.4 | DEVICE-DEPENDENT I/O PREPROCESSING BY THE DRIVER | 2-3 |
| 2.5 | QUEUING THE I/O REQUEST PACKET TO THE DRIVER | 2-4 |
| 2.6 | ACTIVATING THE PRINTER | 2-5 |
| 2.7 | WAITING FOR A DEVICE INTERRUPT | 2-5 |
| 2.8 | HANDLING INTERRUPTS | 2-6 |
| 2.9 | I/O POSTPROCESSING BY THE DRIVER | 2-7 |
| 2.10 | I/O POSTPROCESSING BY VMS | 2-7 |
| CHAPTER 3 SYNCHRONIZATION OF I/O REQUEST PROCESSING | | 3-1 |

| | | |
|--------------|--|-------------|
| 3.1 | INTERRUPT PRIORITY LEVELS | 3-1 |
| 3.1.1 | Interrupt Service Routines | 3-3 |
| 3.1.2 | IPL Use During I/O Processing | 3-3 |
| 3.1.2.1 | IPL 2 (IPL\$_ASTDEL) • 3-4 | |
| 3.1.2.2 | IPL 4 (IPL\$_IOPOST) • 3-4 | |
| 3.1.2.3 | IPL 8 Through IPL 11 (Fork IPLs) • 3-5 | |
| 3.1.2.4 | IPL 20 Through IPL 23 (Device IPLs) • 3-5 | |
| 3.1.2.5 | IPL 31 (IPL\$_POWER) • 3-6 | |
| 3.1.3 | Additional IPLs | 3-6 |
| 3.1.3.1 | IPL 3 (IPL\$_RESCHED) • 3-7 | |
| 3.1.3.2 | IPL 6 (IPL\$_QUEUEAST) • 3-7 | |
| 3.1.3.3 | IPL 7 (IPL\$_TIMERFORK) • 3-7 | |
| 3.1.3.4 | IPL 8 (IPL\$_SYNCH) • 3-8 | |
| 3.1.3.5 | IPL 11 (IPL\$_MAILBOX) • 3-8 | |
| 3.1.3.6 | IPL 14 (XDELTA Entry IPL) • 3-8 | |
| 3.1.3.7 | IPL 22 or IPL 24 (Interval Clock IPLs) • 3-8 | |
| 3.1.4 | Modifying IPL in Driver Code | 3-8 |
| 3.1.4.1 | Raising IPL • 3-10 | |
| 3.1.4.2 | Lowering IPL • 3-11 | |
| 3.2 | SPIN LOCKS | 3-11 |
| 3.2.1 | Fork Locks | 3-14 |
| 3.2.2 | Device Locks | 3-15 |
| 3.3 | DEVICE DRIVER SYNCHRONIZATION | 3-15 |
| 3.3.1 | Overview of the Synchronization of an I/O Operation | 3-16 |
| 3.3.2 | Synchronizing the Device Database | 3-19 |
| 3.3.3 | Synchronizing at Driver Fork Level | 3-20 |
| 3.3.3.1 | Forking and the VMS Fork Dispatcher • 3-21 | |
| 3.3.3.2 | Restrictions on Fork Processes • 3-22 | |
| 3.4 | RESOURCE WAIT QUEUES | 3-23 |
| 3.4.1 | Competing for a Controller's Data Channel | 3-23 |

CHAPTER 4 OVERVIEW OF I/O PROCESSING **4-1**

| | | |
|--------------|---|------------|
| 4.1 | PREPROCESSING AN I/O REQUEST | 4-1 |
| 4.1.1 | Process I/O Channel Assignment | 4-3 |
| 4.1.2 | Locating a Device Driver in the I/O Database | 4-4 |
| 4.1.2.1 | Channel Request Block • 4-4 | |
| 4.1.2.2 | Interrupt Dispatch Block • 4-5 | |
| 4.1.2.3 | Device Data Block • 4-6 | |
| 4.1.3 | Validating the I/O Function | 4-7 |

Contents

| | | |
|-------|--|------|
| 4.1.4 | Checking Process I/O Request Quotas _____ | 4-7 |
| 4.1.5 | Validating the I/O Status Block _____ | 4-7 |
| 4.1.6 | Allocating and Setting Up an I/O Request Packet _____ | 4-7 |
| 4.1.7 | FDT Processing _____ | 4-9 |
| <hr/> | | |
| 4.2 | HANDLING DEVICE ACTIVITY | 4-12 |
| 4.2.1 | Creating a Driver Fork Process to Start I/O _____ | 4-12 |
| 4.2.2 | Activating a Device and Waiting for an Interrupt _____ | 4-13 |
| 4.2.3 | Handling a Device Interrupt _____ | 4-14 |
| 4.2.4 | Switching from Interrupt to Fork Process Context _____ | 4-14 |
| 4.2.5 | Activating a Fork Process from a Fork Queue _____ | 4-15 |
| <hr/> | | |
| 4.3 | COMPLETING AN I/O REQUEST | 4-17 |
| 4.3.1 | I/O Postprocessing _____ | 4-17 |

PART II WRITING A DEVICE DRIVER

CHAPTER 5 TEMPLATE FOR A DEVICE DRIVER 5-1

| | | |
|-------|---|-----|
| 5.1 | CODING CONVENTIONS | 5-1 |
| <hr/> | | |
| 5.2 | RESTRICTIONS ON THE USE OF DEVICE-REGISTER I/O SPACE | 5-3 |
| <hr/> | | |
| 5.3 | IMPLEMENTING CONDITIONAL CODE IN A DRIVER | 5-5 |
| <hr/> | | |
| 5.4 | DRIVER TEMPLATE | 5-6 |

CHAPTER 6 WRITING DEVICE-DRIVER TABLES 6-1

| | | |
|-------|------------------------------|-----|
| 6.1 | DRIVER PROLOGUE TABLE | 6-1 |
| <hr/> | | |
| 6.2 | DRIVER DISPATCH TABLE | 6-3 |

| | | |
|-------|---|-----|
| 6.3 | FUNCTION DECISION TABLE | 6-4 |
| 6.3.1 | Defining Buffered-I/O Functions | 6-7 |
| 6.3.2 | Defining Device-Specific Function Codes | 6-7 |

CHAPTER 7 WRITING FDT ROUTINES 7-1

| | | |
|---------|---|-----|
| 7.1 | CONTEXT OF FDT ROUTINE EXECUTION | 7-1 |
| 7.2 | FDT ROUTINES AND THEIR EXIT PATHS | 7-2 |
| 7.2.1 | FDT Exit Paths | 7-3 |
| 7.2.1.1 | RSB • 7-3 | |
| 7.2.1.2 | JMP G^EXE\$QIODRVPKT • 7-4 | |
| 7.2.1.3 | JMP G^EXE\$FINISHIO or JMP G^EXE\$FINISHIOC • 7-4 | |
| 7.2.1.4 | JMP G^EXE\$ABORTIO • 7-4 | |
| 7.2.1.5 | JSB G^EXE\$ALTQUEPKT • 7-5 | |
| 7.3 | FDT ROUTINES FOR VMS DIRECT I/O | 7-5 |
| 7.4 | FDT ROUTINES FOR VMS BUFFERED I/O | 7-6 |
| 7.4.1 | Checking Accessibility of the User's Buffer | 7-6 |
| 7.4.2 | Allocating the System Buffer | 7-6 |
| 7.4.3 | Buffered-I/O Postprocessing | 7-7 |
| 7.5 | FDT ROUTINES PROVIDED BY VMS | 7-8 |

CHAPTER 8 WRITING A START-I/O ROUTINE 8-1

| | | |
|-------|--|-----|
| 8.1 | TRANSFERRING CONTROL TO THE START-I/O ROUTINE | 8-1 |
| 8.2 | CONTEXT OF A DRIVER FORK PROCESS | 8-1 |
| 8.3 | FUNCTIONS OF A START-I/O ROUTINE | 8-2 |
| 8.3.1 | Obtaining Controller Access | 8-3 |
| 8.3.2 | Obtaining and Converting the I/O Function Code and Its Modifiers | 8-4 |
| 8.3.3 | Preparing the Device Activation Bit Mask | 8-4 |
| 8.3.4 | Synchronizing Access to the Device Database | 8-5 |
| 8.3.5 | Checking for a Local Processor Power Failure | 8-5 |
| 8.3.6 | Activating the Device | 8-5 |

Contents

| | | |
|-------------------|---|-------------|
| 8.4 | WAITING FOR AN INTERRUPT OR TIMEOUT | 8-6 |
| 8.4.1 | Expansion of WFIKPCH Macro | 8-6 |
| 8.4.2 | IOC\$WFIKPCH Routine | 8-7 |
| <hr/> | | |
| CHAPTER 9 | WRITING AN INTERRUPT SERVICE ROUTINE | 9-1 |
| 9.1 | INTERRUPT CONTEXT | 9-3 |
| 9.2 | SERVICING A SOLICITED INTERRUPT | 9-3 |
| 9.3 | SERVICING AN UNSOLICITED INTERRUPT | 9-4 |
| 9.3.1 | Examples of Unsolicited Interrupts | 9-6 |
| <hr/> | | |
| CHAPTER 10 | COMPLETING AN I/O REQUEST AND HANDLING TIMEOUTS | 10-1 |
| 10.1 | I/O POSTPROCESSING | 10-1 |
| 10.1.1 | EXE\$IOFORK | 10-1 |
| 10.1.2 | Completing an I/O Request | 10-2 |
| 10.1.2.1 | Releasing the Controller • 10-2 | |
| 10.1.2.2 | Saving Status, Count, and Device-Dependent Status • 10-3 | |
| 10.1.2.3 | Returning Control to the Operating System • 10-3 | |
| 10.2 | TIMEOUT HANDLING ROUTINES | 10-4 |
| 10.2.1 | Retrying an I/O Operation | 10-5 |
| 10.2.2 | Aborting an I/O Request | 10-6 |
| 10.2.3 | Sending a Message to the Operator | 10-6 |
| <hr/> | | |
| CHAPTER 11 | OTHER DRIVER ROUTINES | 11-1 |
| 11.1 | INITIALIZATION ROUTINES | 11-1 |
| 11.1.1 | Controller Initialization Routine | 11-1 |
| 11.1.2 | Unit Initialization Routine | 11-2 |
| 11.1.3 | Initialization During Driver Loading | 11-3 |
| 11.1.4 | Initialization During Recovery from a Power Failure | 11-4 |
| 11.1.5 | Forking from a Driver Initialization Routine | 11-5 |

| | | |
|-------------|---|--------------|
| 11.2 | CANCEL-I/O ROUTINE | 11-6 |
| 11.2.1 | Context of a Cancel-I/O Routine _____ | 11-7 |
| 11.2.2 | Drivers That Need No Cancel-I/O Routine _____ | 11-7 |
| 11.2.3 | Device-Independent Cancel-I/O Routine _____ | 11-8 |
| 11.2.4 | Device-Dependent Cancel-I/O Routine _____ | 11-8 |
| 11.3 | ERROR LOGGING ROUTINES | 11-8 |
| 11.3.1 | Error Logging Routines Supplied by VMS _____ | 11-9 |
| 11.3.2 | Register Dumping Routine _____ | 11-10 |
| 11.3.3 | Interpreting Error Log Entries _____ | 11-11 |
| 11.4 | CLONED UCB ROUTINE | 11-11 |

PART III BUS SPECIFIC CONSIDERATIONS AND ADVANCED TOPICS

CHAPTER 12 UNIBUS AND Q22 BUS DEVICE SUPPORT 12-1

| | | |
|-------------|--|--------------|
| 12.1 | FUNCTIONS OF THE UNIBUS ADAPTER AND Q22 BUS INTERFACE | 12-1 |
| 12.1.1 | Reading and Writing Device Registers _____ | 12-4 |
| 12.1.2 | Map Registers _____ | 12-4 |
| 12.1.3 | UNIBUS Adapter Data Transfer Paths _____ | 12-8 |
| 12.1.3.1 | Direct Data Path • 12-10 | |
| 12.1.3.2 | Buffered Data Paths • 12-11 | |
| 12.1.3.3 | Byte-Offset Data Transfers • 12-13 | |
| 12.1.3.4 | Purging a Buffered Data Path • 12-13 | |
| 12.1.3.5 | Longword-Aligned, 32-Bit, Random-Access Mode • 12-14 | |
| 12.2 | WRITING DRIVER CODE FOR UNIBUS/Q22 BUS DMA TRANSFERS | 12-15 |
| 12.2.1 | Selecting and Requesting a Data Path _____ | 12-17 |
| 12.2.1.1 | Requesting a Buffered Data Path • 12-17 | |
| 12.2.1.2 | Requesting a Permanent Buffered Data Path • 12-18 | |
| 12.2.1.3 | Requesting the Direct Data Path • 12-18 | |
| 12.2.1.4 | Mixed Use of Direct and Buffered Data Paths • 12-19 | |
| 12.2.2 | Requesting Map Registers _____ | 12-19 |
| 12.2.2.1 | Allocating Map Registers • 12-19 | |
| 12.2.2.2 | Permanently Allocating Map Registers • 12-20 | |
| 12.2.3 | Loading Map Registers _____ | 12-21 |

Contents

| | | |
|----------|--|-------|
| 12.2.4 | Computing the Starting Address of a Transfer | 12-22 |
| 12.2.5 | Computing the Transfer Length | 12-23 |
| 12.2.6 | Activating the Device | 12-23 |
| 12.2.7 | Completing a DMA Transfer | 12-24 |
| 12.2.7.1 | Purging the Data Path • 12-24 | |
| 12.2.7.2 | Releasing a Buffered Data Path • 12-25 | |
| 12.2.7.3 | Releasing Map Registers • 12-25 | |
| 12.2.8 | Considerations for MicroVAX I DMA Devices | 12-26 |

| | | |
|----------|---|-------|
| 12.3 | INTERRUPT DISPATCHING IN A UNIBUS/Q22 BUS SYSTEM | 12-27 |
| 12.3.1 | Direct-Vector and Non-Direct-Vector Interrupt Dispatching | 12-29 |
| 12.3.2 | Adapter Dispatch Table | 12-31 |
| 12.3.3 | Interrupt Transfer Vector and Interrupt Transfer Routine | 12-31 |
| 12.3.4 | Multilevel Device Interrupt Dispatching for Q22 Bus Devices | 12-34 |
| 12.3.4.1 | Ensuring That the Q22 Bus Is Properly Configured • 12-35 | |
| 12.3.4.2 | Effects of Enabling Multilevel Device Interrupt Dispatching on Device Drivers • 12-36 | |

CHAPTER 13 MASSBUS DEVICE SUPPORT

13-1

| | | |
|----------|---|-------|
| 13.1 | MASSBUS ADAPTER REGISTERS | 13-1 |
| 13.1.1 | Loading MASSBUS Adapter Registers | 13-3 |
| 13.1.2 | MASSBUS Adapter Registers and Offsets | 13-4 |
| 13.1.3 | Modifying MASSBUS Adapter Registers | 13-6 |
| 13.2 | I/O DATABASE FOR MASSBUS DEVICES | 13-6 |
| 13.3 | MASSBUS ADAPTER OPERATIONS | 13-8 |
| 13.4 | MASSBUS ADAPTER'S INTERRUPT DISPATCHING | 13-9 |
| 13.4.1 | Checking for MASSBUS Adapter Ownership | 13-9 |
| 13.4.2 | Dispatching a Device Interrupt | 13-10 |
| 13.5 | SPECIAL CONSIDERATIONS FOR MASSBUS DEVICE DRIVERS | 13-11 |
| 13.5.1 | Unit Initialization Routine | 13-11 |
| 13.5.2 | The MASSBUS Adapter and the I/O Database | 13-12 |
| 13.5.3 | Start-I/O Routine | 13-12 |
| 13.5.3.1 | Requesting Controller Data Channels • 13-12 | |
| 13.5.3.2 | Loading Map Registers • 13-13 | |

| | | |
|----------|---|-------|
| 13.5.3.3 | Releasing Controller Data Channels • 13–14 | |
| 13.5.4 | DPTAB Macro _____ | 13–14 |
| <hr/> | | |
| 13.6 | INTERRUPT SERVICE ROUTINES FOR MASSBUS DEVICES | 13–14 |
| 13.6.1 | Transferring Control to the Interrupt Service Routine _____ | 13–14 |
| 13.6.2 | Returning Control to MBA\$INT _____ | 13–15 |
| 13.6.3 | Considerations for Interrupt Service Routines _____ | 13–15 |

CHAPTER 14 GENERIC VAXBI DEVICE SUPPORT 14–1

| | | |
|----------|---|-------|
| 14.1 | OVERVIEW | 14–1 |
| <hr/> | | |
| 14.2 | VAXBI CONCEPTS | 14–1 |
| 14.2.1 | VAXBI Address Space _____ | 14–2 |
| 14.2.2 | Backplane Interconnect Interface Chip (BIIC) _____ | 14–5 |
| <hr/> | | |
| 14.3 | INITIALIZATION PERFORMED BY VMS | 14–5 |
| 14.3.1 | Data Structures _____ | 14–7 |
| 14.3.2 | System Control Block _____ | 14–9 |
| <hr/> | | |
| 14.4 | INITIALIZATION PERFORMED BY THE VAXBI DEVICE DRIVER | 14–9 |
| 14.4.1 | Examining BIIC Self-Test Status _____ | 14–11 |
| 14.4.2 | Clearing BIIC Errors, Setting Interrupts, and Enabling Interrupts _____ | 14–12 |
| 14.4.2.1 | Clearing the Bus Error Register • 14–12 | |
| 14.4.2.2 | Loading the Interrupt Destination Register • 14–12 | |
| 14.4.2.3 | Setting Interrupt Vectors • 14–13 | |
| 14.4.2.4 | Enabling Error Interrupts • 14–13 | |
| 14.4.2.5 | Enabling BIIC Options • 14–13 | |
| 14.4.3 | Mapping Window Space _____ | 14–14 |
| <hr/> | | |
| 14.5 | DMA TRANSFERS | 14–15 |
| 14.5.1 | Example: DMB32 Asynchronous/Synchronous Multiplexer _____ | 14–17 |
| <hr/> | | |
| 14.6 | UNIT INITIALIZATION ROUTINE | 14–19 |

Contents

| | | |
|---|--|-------------|
| 14.7 | REGISTER DUMPING ROUTINE | 14-19 |
| 14.8 | LOADING A VAXBI DEVICE DRIVER | 14-20 |
| 14.9 | BIIC REGISTER DEFINITIONS | 14-21 |
| CHAPTER 15 LOADING A DEVICE DRIVER | | 15-1 |
| 15.1 | PREPARING A DRIVER FOR LOADING INTO THE OPERATING SYSTEM | 15-1 |
| 15.2 | LOADING A DRIVER | 15-2 |
| 15.2.1 | LOAD Command | 15-2 |
| 15.2.2 | CONNECT Command | 15-3 |
| 15.2.3 | RELOAD Command | 15-7 |
| 15.2.4 | SHOW/ADAPTER Command | 15-8 |
| 15.2.5 | SHOW/CONFIGURATION Command | 15-9 |
| 15.2.6 | SHOW/DEVICE Command | 15-9 |
| 15.3 | LOADING UNIPROCESSING AND MULTIPROCESSING DRIVERS | 15-10 |
| 15.4 | THE SYSGEN AUTOCONFIGURATION FACILITY | 15-11 |
| 15.4.1 | SYSGEN Device Table | 15-12 |
| 15.4.2 | Device Driver Control of Autoconfiguration | 15-17 |
| 15.4.3 | Floating-Vector Address Calculation | 15-19 |
| 15.4.4 | Floating-CSR Address Calculation | 15-19 |
| 15.4.5 | Rules for Configuration | 15-19 |
| CHAPTER 16 DEBUGGING A DEVICE DRIVER | | 16-1 |
| 16.1 | BOOTSTRAPPING THE SYSTEM WITH XDELTA | 16-1 |
| 16.2 | PROCEEDING FROM THE INITIAL BREAKPOINTS | 16-5 |

| | | |
|----------------|---|--------------|
| 16.3 | LOADING THE DRIVER | 16-5 |
| 16.4 | INSERTING BREAKPOINTS IN DRIVER SOURCE CODE | 16-6 |
| 16.5 | CALCULATING THE BASE OF DRIVER CODE | 16-7 |
| 16.6 | REQUESTING AN XDELTA SOFTWARE INTERRUPT | 16-7 |
| 16.7 | EXAMINING THE VECTOR-JUMP TABLE | 16-8 |
| 16.8 | SETTING AN XDELTA BASE REGISTER | 16-9 |
| 16.9 | EXAMINING THE UCB, IRP, OR PSL | 16-10 |
| 16.10 | XDELTA COMMANDS | 16-10 |
| 16.10.1 | Values and Expressions | 16-12 |
| 16.10.2 | Special Symbols | 16-13 |
| 16.10.2.1 | Stored Base Registers • 16-13 | |
| 16.10.2.2 | Stored Command Strings • 16-13 | |
| 16.10.2.3 | Setting Base Registers • 16-14 | |
| 16.10.3 | Display Names and Locations of Loaded Executive Images | 16-14 |
| 16.10.4 | Set Display Mode | 16-14 |
| 16.10.5 | Open, Examine, and Close Location | 16-15 |
| 16.10.5.1 | Open and Display Value Command • 16-15 | |
| 16.10.5.2 | Display Instruction Command • 16-16 | |
| 16.10.5.3 | Close and Display Next Location Command • 16-16 | |
| 16.10.5.4 | Display Range Command • 16-16 | |
| 16.10.5.5 | Indirect Command • 16-17 | |
| 16.10.5.6 | Display Previous Location Command • 16-17 | |
| 16.10.6 | Breakpoints | 16-17 |
| 16.10.6.1 | Setting Breakpoints • 16-17 | |
| 16.10.6.2 | Clearing Breakpoints • 16-18 | |
| 16.10.6.3 | Displaying Breakpoint List • 16-18 | |
| 16.10.6.4 | Proceeding from Breakpoints • 16-18 | |
| 16.10.6.5 | Setting Complex Breakpoints • 16-18 | |
| 16.10.7 | Step, Set Location, and Execute Instruction Commands | 16-18 |
| 16.10.7.1 | Loading PC and Continuing • 16-18 | |
| 16.10.7.2 | Execute Instruction and Step Command • 16-19 | |
| 16.10.7.3 | Step Instruction Over Subroutine Command • 16-19 | |
| 16.10.8 | Execute String Command | 16-19 |

Contents

| | | |
|---|--|--------------|
| 16.11 | GUIDELINES FOR DEBUGGING DEVICE DRIVERS | 16-20 |
| 16.11.1 | Opening Device Registers in XDELTA | 16-20 |
| 16.11.2 | Adjusting the Device Timeout Value | 16-20 |
| 16.11.3 | XDELTA and System Failures | 16-20 |
| 16.12 | COMMON DRIVER ERRORS | 16-21 |
| 16.12.1 | References to System Addresses | 16-21 |
| 16.12.2 | Incorrect References to Device Registers | 16-21 |
| 16.12.3 | Destroying Register Contents | 16-21 |
| 16.13 | POOL CHECKING MECHANISM | 16-22 |
| 16.14 | DETECTING DRIVER PROBLEMS IN A MULTIPROCESSING SYSTEM | 16-24 |
| CHAPTER 17 TERMINAL CLASS AND PORT DRIVERS | | 17-1 |
| 17.1 | OVERVIEW | 17-2 |
| 17.2 | DATA STRUCTURES | 17-2 |
| 17.2.1 | Terminal UCB | 17-2 |
| 17.2.2 | Port Driver Vector Table | 17-4 |
| 17.2.3 | Class Driver Vector Table | 17-5 |
| 17.2.4 | Vector Table Generation Macros | 17-5 |
| 17.2.4.1 | \$VECINI Macro • 17-6 | |
| 17.2.4.2 | \$VEC Macro • 17-6 | |
| 17.2.4.3 | \$VECEND Macro • 17-6 | |
| 17.3 | STRUCTURE OF PORT AND CLASS DRIVERS | 17-6 |
| 17.3.1 | Binding Class and Port Drivers | 17-8 |
| 17.4 | PORT DRIVER ROUTINES | 17-8 |
| 17.4.1 | Port Startup Routines | 17-10 |
| 17.4.1.1 | Controller Initialization Routine • 17-11 | |
| 17.4.1.2 | Unit Initialization Routine • 17-11 | |
| 17.4.2 | Port Initiate Routines | 17-11 |
| 17.4.2.1 | PORT_DISCONNECT • 17-12 | |
| 17.4.2.2 | PORT_DS_SET • 17-12 | |
| 17.4.2.3 | PORT_FDT • 17-12 | |
| 17.4.2.4 | PORT_FORKRET • 17-13 | |
| 17.4.2.5 | PORT_MAINT • 17-13 | |

| | | |
|---------------|---|--------------|
| 17.4.2.6 | PORT_SET_LINE • 17-13 | |
| 17.4.2.7 | PORT_SET_MODEM • 17-14 | |
| 17.4.2.8 | PORT_STARTIO • 17-14 | |
| 17.4.3 | Port Service Routines | 17-15 |
| 17.4.3.1 | PORT_ABORT • 17-15 | |
| 17.4.3.2 | PORT_CANCEL • 17-15 | |
| 17.4.3.3 | PORT_RESUME • 17-15 | |
| 17.4.3.4 | PORT_STOP • 17-15 | |
| 17.4.3.5 | PORT_XOFF • 17-16 | |
| 17.4.3.6 | PORT_XON • 17-16 | |
| 17.4.3.7 | Port Interrupt Service Routines • 17-16 | |

| | | |
|-------------|------------------------------|--------------|
| 17.5 | CLASS DRIVER ROUTINES | 17-17 |
| 17.5.1 | CLASS_DDT | 17-18 |
| 17.5.2 | CLASS_DISCONNECT | 17-18 |
| 17.5.3 | CLASS_DS_TRANS | 17-18 |
| 17.5.4 | CLASS_FORK | 17-18 |
| 17.5.5 | CLASS_GETNXT | 17-19 |
| 17.5.6 | CLASS_PUTNXT | 17-19 |
| 17.5.7 | CLASS_SETUP_UCB | 17-20 |
| 17.5.8 | CLASS_POWERFAIL | 17-20 |
| 17.5.9 | CLASS_READERROR | 17-21 |

CHAPTER 18 MAPPING TO I/O SPACE AND THE CONNECT-TO-INTERRUPT FACILITY **18-1**

| | | |
|-------------|---|-------------|
| 18.1 | I/O ADDRESS SPACE | 18-1 |
| 18.2 | PFN MAPPING | 18-5 |
| 18.2.1 | Notes on PFN Mapping | 18-6 |
| 18.3 | CONNECTING TO AN INTERRUPT VECTOR | 18-7 |
| 18.3.1 | Performing the Connect-to-Interrupt | 18-8 |
| 18.3.2 | \$QIO Connect-to-Interrupt Request to Driver | 18-9 |
| 18.3.3 | The Connect-to-Interrupt Driver (CONINTERR.EXE) | 18-12 |
| 18.3.4 | Process-Specified Routines | 18-13 |
| 18.3.4.1 | Unit Initialization Routine • 18-14 | |
| 18.3.4.2 | Start-I/O Routine • 18-15 | |
| 18.3.4.3 | Interrupt Service Routine • 18-16 | |
| 18.3.4.4 | Cancel-I/O Routine • 18-17 | |
| 18.3.5 | AST Procedure | 18-18 |

Contents

| | | |
|--------|---|-------|
| 18.4 | REAL-TIME APPLICATIONS EXAMPLES | 18-18 |
| 18.4.1 | Example 1: KW11-W Watchdog Timer | 18-19 |
| 18.4.2 | Example 2: AD11-K, AM11-K A/D Converter with Multiplexer Connected to the UNIBUS | 18-20 |
| 18.4.3 | Example 3: KW11-P Real-Time Clock and AD11-K Converter Connected to the UNIBUS | 18-22 |

PART IV REFERENCE SECTION AND EXAMPLES

| | | |
|------------|-----------------|-----|
| APPENDIX A | DATA STRUCTURES | A-1 |
|------------|-----------------|-----|

| | | |
|------|-------------------------------------|------|
| A.1 | CONFIGURATION CONTROL BLOCK (ACF) | A-2 |
| A.2 | ADAPTER CONTROL BLOCK (ADP) | A-4 |
| A.3 | CHANNEL CONTROL BLOCK (CCB) | A-11 |
| A.4 | PER-CPU DATABASE (CPU) | A-12 |
| A.5 | CHANNEL REQUEST BLOCK (CRB) | A-17 |
| A.6 | DEVICE DATA BLOCK (DDB) | A-25 |
| A.7 | DRIVER DISPATCH TABLE (DDT) | A-27 |
| A.8 | DRIVER PROLOGUE TABLE (DPT) | A-30 |
| A.9 | INTERRUPT DISPATCH BLOCK (IDB) | A-34 |
| A.10 | I/O REQUEST PACKET (IRP) | A-36 |
| A.11 | I/O REQUEST PACKET EXTENSION (IRPE) | A-41 |

| | | |
|------|--------------------------------|------|
| A.12 | OBJECT RIGHTS BLOCK (ORB) | A-43 |
| A.13 | SPIN LOCK DATA STRUCTURE (SPL) | A-45 |
| A.14 | UNIT CONTROL BLOCK (UCB) | A-47 |

APPENDIX B VMS MACROS INVOKED BY DRIVERS B-1

| | |
|------------------------------|------|
| ADPDISP | B-2 |
| CASE | B-5 |
| CLASS_CTRL_INIT | B-6 |
| CLASS_UNIT_INIT | B-7 |
| CPUDISP | B-8 |
| DDTAB | B-10 |
| \$DEF | B-12 |
| \$DEFEND | B-13 |
| \$DEFINI | B-14 |
| DEVICELOCK | B-15 |
| DEVICEUNLOCK | B-17 |
| DPTAB | B-19 |
| DPT_STORE | B-22 |
| DSBINT | B-25 |
| ENBINT | B-26 |
| \$EQLST | B-27 |
| FIND_CPU_DATA | B-29 |
| FORK | B-30 |
| FORKLOCK | B-31 |
| FORKUNLOCK | B-33 |
| FUNCTAB | B-34 |
| IFNORD, IFNOWRT, IFRD, IFWRT | B-36 |
| INVALIDATE_TB | B-38 |
| IOFORK | B-40 |
| LOADALT | B-41 |
| LOADMBA | B-42 |
| LOADUBA | B-43 |
| LOCK | B-44 |
| PURDPR | B-46 |
| READ_SYSTIME | B-47 |
| RELALT | B-48 |
| RELCHAN | B-49 |
| RELDPR | B-50 |

Contents

| | |
|------------------|------|
| RELMPR | B-51 |
| RELSCHAN | B-52 |
| REQALT | B-53 |
| REQCOM | B-54 |
| REQDPR | B-55 |
| REQMPR | B-56 |
| REQPCHAN | B-57 |
| REQSCHAN | B-58 |
| SAVIPL | B-59 |
| SETIPL | B-60 |
| SOFTINT | B-62 |
| TIMEWAIT | B-63 |
| TIMEDWAIT | B-64 |
| UNLOCK | B-66 |
| \$VEC | B-67 |
| \$VECEND | B-68 |
| \$VECINI | B-69 |
| \$VIELD, _VIELD | B-70 |
| WFIKPCH, WFIRLCH | B-72 |

APPENDIX C OPERATING SYSTEM ROUTINES

C-1

| | |
|---|------|
| COM\$DELATTNAST | C-2 |
| COM\$DRVDEALMEM | C-3 |
| COM\$FLUSHATTNS | C-4 |
| COM\$POST | C-5 |
| COM\$SETATTNAST | C-6 |
| ERL\$DEVICERR, ERL\$DEVICTMO, ERL\$DEVICEATTN | C-8 |
| EXE\$ABORTIO | C-10 |
| EXE\$ALLOCBUF, EXE\$ALLOCIRP | C-12 |
| EXE\$ALONONPAGED | C-14 |
| EXE\$ALONPAGVAR | C-15 |
| EXE\$ALOPHYCNTG | C-16 |
| EXE\$ALTQUEPKT | C-17 |
| EXE\$CREDIT_BYTCNT, EXE\$CREDIT_BYTCNT_BYTLM | C-18 |
| EXE\$DEANONPAGED | C-19 |
| EXE\$DEBIT_BYTCNT(_NW), EXE\$DEBIT_BYTCNT_BYTLM(_NW) | C-20 |
| EXE\$DEBIT_BYTCNT_ALO, EXE\$DEBIT_BYTCNT_BYTLM_ALO | C-22 |

| | |
|--|------|
| EX\$FINISHIO, EX\$FINISHIOC | C-24 |
| EX\$FORK | C-26 |
| EX\$INSERTIRP | C-27 |
| EX\$INSIOQ, EX\$INSIOQC | C-28 |
| EX\$INSTIMQ | C-29 |
| EX\$IOFORK | C-30 |
| EX\$MODIFY | C-31 |
| EX\$MODIFYLOCK, EX\$MODIFYLOCKR | C-34 |
| EX\$ONEPARM | C-37 |
| EX\$QIODRVPKT | C-38 |
| EX\$QIORETURN | C-39 |
| EX\$READ | C-40 |
| EX\$READCHK, EX\$READCHKR | C-43 |
| EX\$READLOCK, EX\$READLOCKR | C-45 |
| EX\$SENSEMODE | C-48 |
| EX\$SETCHAR, EX\$SETMODE | C-49 |
| EX\$SNDEVMSG | C-51 |
| EX\$WRITE | C-53 |
| EX\$WRITECHK, EX\$WRITECHKR | C-55 |
| EX\$WRITELOCK, EX\$WRITELOCKR | C-57 |
| EX\$WRTMAILBOX | C-59 |
| EX\$ZEROPARM | C-60 |
| IOC\$ALOALTMAP, IOC\$ALOALTMAPN, IOC\$ALOALTMAPSP | C-61 |
| IOC\$ALOUBAMAP, IOC\$ALOUBAMAPN | C-63 |
| IOC\$APPLYECC | C-65 |
| IOC\$CANCELIO | C-66 |
| IOC\$DIAGBUFILL | C-67 |
| IOC\$INITIATE | C-68 |
| IOC\$IOPOST | C-70 |
| IOC\$LOADALTMAP | C-72 |
| IOC\$LOADMBAMAP | C-74 |
| IOC\$LOADUBAMAP, IOC\$LOADUBAMAPA | C-75 |
| IOC\$MOVFRUSER, IOC\$MOVFRUSER2 | C-77 |
| IOC\$MOVTOUSER, IOC\$MOVTOUSER2 | C-78 |
| IOC\$PURGDATAP | C-79 |
| IOC\$RELALTMAP | C-81 |
| IOC\$RELCHAN | C-83 |
| IOC\$RELDATAP | C-84 |
| IOC\$RELMAPREG | C-86 |
| IOC\$RELSCHAN | C-88 |

Contents

| | |
|--|-------|
| IOC\$REQALTMAP | C-89 |
| IOC\$REQCOM | C-91 |
| IOC\$REQDATAP, IOC\$REQDATAPNW | C-93 |
| IOC\$REQMAPREG | C-95 |
| IOC\$REQPCHANH, IOC\$REQPCHANL, IOC\$REQSCHANH, IOC\$REQSCHANL | C-97 |
| IOC\$RETURN | C-99 |
| IOC\$VERIFYCHAN | C-100 |
| IOC\$WFIKPCH, IOC\$WFIRLCH | C-101 |
| LDR\$ALLOC_PT | C-103 |
| LDR\$DEALLOC_PT | C-104 |
| MMG\$UNLOCK | C-105 |
| SMP\$ACQNOIPL | C-106 |
| SMP\$ACQUIRE | C-107 |
| SMP\$ACQUIREL | C-108 |
| SMP\$RELEASE | C-109 |
| SMP\$RELEASEL | C-110 |
| SMP\$RESTORE | C-111 |
| SMP\$RESTOREL | C-112 |

APPENDIX D DEVICE DRIVER ENTRY POINTS

D-1

| | |
|--|------|
| ALTERNATE START-I/O ROUTINE | D-2 |
| CANCEL-I/O ROUTINE | D-3 |
| CLONED UCB ROUTINE | D-5 |
| CONTROLLER INITIALIZATION ROUTINE | D-7 |
| DRIVER UNLOADING ROUTINE | D-9 |
| FDT ROUTINES | D-10 |
| INTERRUPT SERVICE ROUTINE | D-12 |
| REGISTER DUMPING ROUTINE | D-14 |
| START-I/O ROUTINE | D-15 |
| TIMEOUT HANDLING ROUTINE | D-17 |
| UNIT DELIVERY ROUTINE | D-19 |
| UNIT INITIALIZATION ROUTINE | D-21 |
| UNSOLICITED INTERRUPT SERVICE ROUTINE | D-23 |

| | | |
|-------------------|---|------------|
| APPENDIX E | SAMPLE DRIVER FOR THE RL11, RL01, AND RL02 | E-1 |
|-------------------|---|------------|

| | | |
|-------------------|--|------------|
| APPENDIX F | SAMPLE DRIVER FOR THE DR11-W AND DRV11-WA | F-1 |
|-------------------|--|------------|

| | | |
|-------------------|---|------------|
| APPENDIX G | VMS VERSION 5.0 AND KERNEL-MODE CODE | G-1 |
|-------------------|---|------------|

| | | |
|----------------|---|-------------|
| G.1 | UNIPROCESSOR AND MULTIPROCESSOR DEVICE DRIVERS | G-1 |
| G.1.1 | MULTIPROCESSING System Parameter _____ | G-2 |
| G.1.2 | Device Driver Loading _____ | G-3 |
| G.1.3 | VMS Synchronization Macros _____ | G-4 |
| G.2 | CHANGES REQUIRED OF ALL EXISTING DRIVERS UNDER VMS VERSION 5.0 | G-4 |
| G.2.1 | Specifying the Address of the Driver's Interrupt Service Routine in the DPT _____ | G-5 |
| G.2.2 | Checking, Debiting, and Crediting a Process's Byte Count Quota _____ | G-5 |
| G.2.3 | Referring to the Current PCB _____ | G-7 |
| G.2.4 | Allocating System Page-Table Entries _____ | G-7 |
| G.2.5 | Referring to a System Process Mailbox _____ | G-7 |
| G.2.6 | Reassembling and Relinking the Driver _____ | G-8 |
| G.3 | ADAPTING DEVICE DRIVERS TO RUN ON A VMS MULTIPROCESSING SYSTEM | G-8 |
| G.3.1 | Specifying the Fork Lock Index _____ | G-8 |
| G.3.2 | Synchronizing Access to the Device Database with the Interrupt Service Routine _____ | G-9 |
| G.3.2.1 | Synchronizing at Device IPL • G-9 | |
| G.3.2.2 | Raising IPL to IPL\$_POWER • G-10 | |
| G.3.2.3 | Synchronization Within the Interrupt Service Routine • G-11 | |
| G.3.3 | Controller and Unit Initialization Routines _____ | G-12 |
| G.3.3.1 | Permanently Allocating Map Registers and Buffered Data Paths • G-12 | |
| G.3.4 | Timeout Handling Routine _____ | G-13 |
| G.3.5 | General Methods for Synchronizing Kernel-Mode Code _____ | G-13 |
| G.3.5.1 | Using the Spin Lock Synchronization Macros • G-13 | |
| G.3.5.2 | Interlocking Access to Data Cells and Queues • G-14 | |
| G.3.6 | Miscellaneous Conversion Tasks _____ | G-15 |
| G.3.6.1 | Reading the System Time • G-15 | |
| G.3.6.2 | Calling the Driver Fork Process from a TQE • G-16 | |
| G.3.6.3 | Invalidating Translation Buffer Entries • G-16 | |

Contents

| | | |
|--------------|--|-------------|
| G.3.6.4 | Unsupported Use of the IRP • G-16 | |
| G.3.7 | Troubleshooting a Device Driver in a Multiprocessing System | G-17 |
| G.3.7.1 | Multiprocessing Bugchecks • G-17 | |
| G.3.7.2 | Analyzing a Multiprocessing System Failure • G-18 | |
| | G.3.7.2.1 Investigating the Status of Spin Locks • G-19 | |
| G.3.7.3 | Using XDELTA on SMP Systems • G-20 | |
| <hr/> | | |
| G.4 | MULTIPROCESSING IMPLEMENTATION DETAILS | G-20 |
| G.4.1 | Processor States | G-20 |
| G.4.2 | System Initialization | G-22 |
| G.4.3 | Scheduling in a VMS Multiprocessing Environment | G-24 |
| G.4.4 | Timekeeping in a VMS Multiprocessing Environment | G-25 |

GLOSSARY

Glossary-1

INDEX

EXAMPLES

| | | |
|------|--|------|
| 16-1 | Loading a Driver | 16-6 |
| 18-1 | Locating the Adapter Address Space of a DWBUA Adapter on a VAXBI Bus | 18-4 |

FIGURES

| | | |
|-----|---|------|
| 1-1 | The I/O Database | 1-5 |
| 1-2 | SBI-Based System Configurations | 1-11 |
| 1-3 | VAXBI-Based System Configurations | 1-13 |
| 1-4 | MicroVAX 3600-Series and MicroVAX II System Configuration | 1-15 |
| 1-5 | MicroVAX I System Configuration | 1-16 |
| 1-6 | Example of I/O Request Processing | 1-19 |
| 2-1 | A Printer Write Function | 2-2 |
| 3-1 | Synchronizing I/O Request Processing | 3-16 |
| 3-2 | Synchronizing I/O Request Completion | 3-18 |
| 3-3 | Processor-Specific Fork Queue Structure | 3-22 |
| 4-1 | Sequence of Driver Execution | 4-2 |
| 4-2 | Detailed Sequence of VMS I/O Processing | 4-3 |
| 4-3 | Data Structures for Three Devices on One Controller | 4-5 |

| | | |
|------|---|-------|
| 4-4 | I/O Database for Two Controllers _____ | 4-6 |
| 4-5 | Layout of a Function Decision Table _____ | 4-9 |
| 4-6 | FDT Routines and I/O Preprocessing _____ | 4-11 |
| 4-7 | Creating a Fork Process After an Interrupt _____ | 4-15 |
| 4-8 | Reactivation of a Driver Fork Process _____ | 4-16 |
| 5-1 | Driver Organization _____ | 5-2 |
| 7-1 | \$QIO Scan of a Function Decision Table _____ | 7-3 |
| 7-2 | Format of System Buffer for a Buffered-I/O Read Function _____ | 7-7 |
| 8-1 | Inserting a UCB into the Channel Wait Queue _____ | 8-3 |
| 9-1 | Flow of Interrupt Servicing _____ | 9-2 |
| 12-1 | UNIBUS and Q22 Bus Map Registers _____ | 12-6 |
| 12-2 | Mapping a UNIBUS Address to a Physical Address _____ | 12-7 |
| 12-3 | Mapping a Q22 Bus Address to a Physical Address _____ | 12-8 |
| 12-4 | UNIBUS Data Path Registers _____ | 12-9 |
| 12-5 | Direct-Vector Interrupt Dispatching _____ | 12-28 |
| 12-6 | Non-Direct-Vector Interrupt Dispatching _____ | 12-29 |
| 12-7 | VEC Structures Within a CRB _____ | 12-32 |
| 12-8 | Interrupt Transfer Vector Block (VEC) _____ | 12-33 |
| 13-1 | MASSBUS Configuration _____ | 13-2 |
| 13-2 | MASSBUS External-Register Longword _____ | 13-2 |
| 13-3 | Location of MASSBUS Registers in Physical Address Space _____ | 13-5 |
| 13-4 | I/O Database for MASSBUS Disk Unit _____ | 13-7 |
| 13-5 | I/O Database for MASSBUS Disk and Tape Units _____ | 13-7 |
| 13-6 | I/O Data Structures Used in Dispatching a MASSBUS Device Interrupt _____ | 13-8 |
| 14-1 | VAXBI Address Space _____ | 14-2 |
| 14-2 | Description of VAXBI I/O Address Space _____ | 14-3 |
| 14-3 | Physical Addresses in VAXBI I/O Address Space _____ | 14-4 |
| 14-4 | VAXBI Device Vectors _____ | 14-10 |
| 14-5 | Backplane Interconnect Interface Chip (BIIC) Registers _____ | 14-22 |
| 16-1 | Format of the POOLCHECK System Parameter _____ | 16-22 |
| 16-2 | Poisoned Pool Packet _____ | 16-24 |
| 17-1 | UCB Structure for Terminal Class/Port Drivers _____ | 17-3 |
| 17-2 | Port Driver Vector Table _____ | 17-4 |
| 17-3 | Class Driver Vector Table _____ | 17-5 |
| 17-4 | Port Driver Structure _____ | 17-7 |
| 17-5 | Class Driver Structure _____ | 17-7 |
| 17-6 | Terminal Class/Port Driver Binding _____ | 17-9 |
| 18-1 | Format of a Physical Address _____ | 18-4 |

Contents

| | | |
|------|---|------|
| A-1 | The I/O Database _____ | A-2 |
| A-2 | Configuration Control Block (ACF) _____ | A-3 |
| A-3 | Adapter Control Block (ADP) _____ | A-5 |
| A-4 | Channel Control Block (CCB) _____ | A-11 |
| A-5 | Per-CPU Database (CPU) _____ | A-13 |
| A-6 | Channel Request Block (CRB) _____ | A-18 |
| A-7 | Interrupt Transfer Vector Block (VEC) _____ | A-22 |
| A-8 | Device Data Block (DDB) _____ | A-26 |
| A-9 | Driver Dispatch Table (DDT) _____ | A-28 |
| A-10 | Driver Prologue Table (DPT) _____ | A-31 |
| A-11 | Interrupt Dispatch Block (IDB) _____ | A-34 |
| A-12 | I/O Request Packet (IRP) _____ | A-37 |
| A-13 | I/O Request Packet Extension (IRPE) _____ | A-42 |
| A-14 | Object Rights Block (ORB) _____ | A-44 |
| A-15 | Spin Lock Data Structure (SPL) _____ | A-46 |
| A-16 | Composition of Extended Unit Control Blocks _____ | A-49 |
| A-17 | Unit Control Block (UCB) _____ | A-50 |
| A-18 | UCB Error-Log Extension _____ | A-59 |
| A-19 | UCB Local Tape Extension _____ | A-60 |
| A-20 | UCB Local Disk Extension _____ | A-61 |
| A-21 | UCB Terminal Extension _____ | A-63 |
| G-1 | Multiprocessor State Transitions _____ | G-21 |

TABLES

| | | |
|------|---|-------|
| 3-1 | IPLs Defined by VMS _____ | 3-2 |
| 3-2 | VMS Macros That Change a Processor's IPL _____ | 3-9 |
| 3-3 | Static Spin Locks _____ | 3-12 |
| 6-1 | I/O Function Codes _____ | 6-5 |
| 7-1 | Registers Loaded by the \$QIO System Service _____ | 7-1 |
| 7-2 | FDT Routines Provided by VMS _____ | 7-8 |
| 12-1 | Features of the UNIBUS Adapters/Q22 Bus Interfaces of VAX Systems _____ | 12-2 |
| 12-2 | VAX System UNIBUS/Q22 Bus Interrupt Dispatching _____ | 12-30 |
| 13-1 | Major Offsets Defined by \$MBADEF _____ | 13-4 |
| 14-1 | Contents of the BIIC Registers _____ | 14-23 |
| 15-1 | Conventional Nexus Assignments _____ | 15-5 |
| 15-2 | SYSGEN Device Table _____ | 15-13 |
| 16-1 | Boot Flags That Control the Loading of XDELTA _____ | 16-2 |
| 16-2 | Recommended Methods for Bootstrapping with XDELTA _____ | 16-2 |

| | | |
|------|--|-------|
| 16-3 | Requesting an XDELTA Software Interrupt _____ | 16-8 |
| 16-4 | XDELTA Command Summary _____ | 16-10 |
| 16-5 | Settings of MULTIPROCESSING System Parameter _____ | 16-25 |
| 16-6 | Bugchecks Produced by Full-Checking Multiprocessing _____ | 16-25 |
| 17-1 | Port Driver Routines _____ | 17-10 |
| 17-2 | Class Driver Routines _____ | 17-17 |
| 18-1 | Symbols Defined by the \$IOxxxDEF Macros _____ | 18-2 |
| 18-2 | UNIBUS and Q22 Bus Adapter Address Space _____ | 18-3 |
| A-1 | Contents of the Configuration Control Block _____ | A-4 |
| A-2 | Contents of Adapter Control Block _____ | A-6 |
| A-3 | Contents of Channel Control Block _____ | A-11 |
| A-4 | Per-CPU Database (CPU) _____ | A-14 |
| A-5 | Contents of Channel Request Block _____ | A-19 |
| A-6 | Interrupt Dispatch Vector Block (VEC) _____ | A-23 |
| A-7 | Contents of Device Data Block _____ | A-27 |
| A-8 | Contents of Driver Dispatch Table _____ | A-29 |
| A-9 | Contents of Driver Prologue Table _____ | A-32 |
| A-10 | Contents of Interrupt Dispatch Block _____ | A-35 |
| A-11 | Contents of an I/O Request Packet _____ | A-38 |
| A-12 | Contents of the I/O Request Packet Extension _____ | A-43 |
| A-13 | Contents of Object Rights Block _____ | A-45 |
| A-14 | Contents of the Spin Lock Data Structure _____ | A-46 |
| A-15 | UCB Extensions and Sizes Defined in \$UCBDEF _____ | A-48 |
| A-16 | Contents of Unit Control Block _____ | A-51 |
| A-17 | UCB Error-Log Extension _____ | A-60 |
| A-18 | UCB Local Tape Extension _____ | A-61 |
| A-19 | UCB Local Disk Extension _____ | A-62 |
| A-20 | UCB Terminal Extension _____ | A-64 |
| G-1 | VMS Synchronization Images _____ | G-2 |
| G-2 | Settings of MULTIPROCESSING System Parameter _____ | G-3 |
| G-3 | Converting IPL Synchronization to Spin Lock Synchronization _____ | G-14 |
| G-4 | Bugchecks Produced Within Full-Checking Synchronization _____ | G-18 |
| G-5 | Multiprocessor States _____ | G-21 |

Preface

The *VMS Device Support Manual* provides information needed to write a device driver that runs under VMS Version 5.0 and to load it into the operating system. DIGITAL makes no guarantee that drivers written for earlier versions of VMS will execute without modification on this version of the operating system. Although the intent is to maintain the existing interface, some unavoidable changes might occur as new features are added.

Intended Audience

This manual is intended for system programmers who are already familiar with VAX processors and the VMS operating system.

Document Structure

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

- Chapter 1 describes the role of a device driver in the VMS operating system, introduces the components of a typical driver and the data structures it uses, and provides an overview of system concepts critical to driver operation. It concludes with an examination of the I/O subsystems of the VAX processing systems.
- Chapter 2 provides an example of a device driver—the VMS line printer driver, and illustrates the functions of the various components of this driver and describes the driver's interaction with VMS.
- Chapter 3 discusses VMS synchronization mechanisms: interrupt priority levels; spin locks, fork locks, and device locks; fork processes and fork queues; and resource-wait queues.
- Chapter 4 provides an overview of I/O processing and discusses the interaction of device drivers with VMS.

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

- Chapter 5 explains some general driver coding rules and conventions, and includes a template of a device driver.
- Chapter 6 describes how to create driver tables, including the driver prologue table, driver dispatch table, and function decision table (FDT).
- Chapter 7 explains how to write FDT routines, use VMS-supplied FDT routines, and transfer control out of I/O request preprocessing.
- Chapter 8 discusses the components of a driver's start-I/O routine.
- Chapter 9 discusses the functions performed by an interrupt service routine.
- Chapter 10 describes how to perform device-dependent I/O completion and write timeout handling routines.

Preface

- Chapter 11 describes unit and controller initialization routines, cancel-I/O routines, error logging routines, register dumping routines, and cloned UCB routines.

Part III contains discussions of bus-specific and processor-specific details that affect the composition and operation of a device driver. It also contains chapters that discuss advanced topics relating to the writing of specific types of drivers.

- Chapter 12 discusses I/O bus features that govern the operation of direct-memory-access (DMA) transfers and affect the code of DMA device drivers for UNIBUS and MicroVAX Q22 bus devices.
- Chapter 13 describes strategies for producing a MASSBUS device driver.
- Chapter 14 describes special coding considerations for generic VAXBI devices.
- Chapter 15 examines the methods by which a device is logically connected to the processor and by which a driver is loaded into the operating system.
- Chapter 16 describes the use of XDELTA as a device driver debugging tool.
- Chapter 17 discusses the components of terminal class and port drivers.
- Chapter 18 describes the connect-to-interrupt driver interface that is available to real-time users.

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

- Appendix A contains a set of figures and tables that describe the contents of each data structure and table in the I/O database.
- Appendix B lists the VMS macros usually invoked by drivers.
- Appendix C describes the context, synchronization, and I/O requirements of the executive routines used by drivers or called as the result of a driver macro invocation.
- Appendix D supplies a condensed description of the function and environment of each driver routine.
- Appendix E includes a sample driver that operates an RL01/RL02-type disk on the UNIBUS or Q22 bus.
- Appendix F contains a sample driver for two connected DR11 controllers on the UNIBUS or Q22 bus.
- Appendix G describes the differences between drivers intended for a VMS uniprocessing environment and those intended for a VMS multiprocessing environment. It further describes those changes that DIGITAL requires or recommends in all existing non-DIGITAL-supplied drivers because of the release of VMS Version 5.0 and also discusses the means by which a uniprocessing driver can be converted to a multiprocessing driver.

The glossary at the end of this manual defines the vocabulary that pertains to device drivers and their environment.

Associated Documents

Before reading the *VMS Device Support Manual* volume, you should have an understanding of the material discussed in the following documents:

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

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

- *VMS System Dump Analyzer Utility Manual*
- *Guide to Maintaining a VMS System*
- *VAX/VMS Internals and Data Structures*
- *VMS Delta/XDelta Utility Manual*

Conventions

This manual describes code transfer operations in three ways:

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

Typographical conventions used in this book include the following:

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

Under the VMS operating system, a *device driver* is a set of routines and tables that the system uses to process an I/O request for a particular device type.

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

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

- In examples, a symbol with a one- to six-character abbreviation indicates that you press a key on the terminal. For instance:

driver-base-address,0;X RET

Preface

- In examples, the symbol <CTRL/x> indicates that you must press the key labeled CTRL while you simultaneously press another key. For instance:

```
$ CREATE MYDRIVER.OPT
BASE=0
CTRL/Z
```

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

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

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

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

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

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

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

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

New and Changed Features

The *VMS Device Support Manual* reflects various enhancements and changes to VMS evident in VMS Version 5.0. It also incorporates information useful in the creation of device drivers for those VAX processors introduced since VMS Version 4.4, including the VAX 8250, VAX 8300, VAX 8530, VAX 8550, VAX 8670, VAX 8800, VAX 8830, VAX 8840, the VAX 6200 series, and the MicroVAX 3600 series.

Among the new features discussed in this manual are the following:

- VMS Version 5.0 incorporates support for symmetric multiprocessing (SMP) in VAX multiprocessing systems, such as the VAX 8300/8350, VAX 8800/8830/8840, and the VAX 6200 series. Drivers designed to run in a multiprocessing environment must supplement IPL synchronization with the additional synchronization mechanism of *spin locks*.

For a full discussion of the impact of VMS Version 5.0 on existing non-DIGITAL-supplied drivers, refer to Appendix G. Appendix G also discusses the means by which you can convert a driver designed to execute in a VMS uniprocessing system so that it also runs in a VMS multiprocessing system.

Chapter 3 supplies a full discussion of the rules for synchronizing drivers.

Appendix B includes descriptions of the following new synchronization macros:

| | |
|--------------|---|
| DEVICELOCK | Achieves synchronized access to a device's database as appropriate to the processing environment |
| DEVICEUNLOCK | Relinquishes synchronized access to a device's database as appropriate to the processing environment |
| FORKLOCK | Achieves synchronized access to a driver's fork database as appropriate to the processing environment |
| FORKUNLOCK | Relinquishes synchronized access to a driver's fork database as appropriate to the processing environment |
| LOCK | Achieves synchronized access to a system resource as appropriate to the processing environment |
| UNLOCK | Relinquishes synchronized access to a system resource as appropriate to the processing environment |

Additionally, Appendix B explains new requirements governing the use of the following existing macros:

| | |
|--------|--|
| DSBINT | Blocks interrupts from occurring on the local processor at or below a specified IPL |
| ENBINT | Lowers the local processor's IPL to a specified value and thus permits interrupts to occur at or beneath the current IPL |
| SETIPL | Sets the current IPL of the local processor |

New and Changed Features

Appendix B also describes the following new macros that synchronize certain tasks of privileged code in both a multiprocessing and uniprocessing environment:

| | |
|---------------|--|
| FIND_CPU_DATA | Obtains the starting virtual address of the current processor's per-CPU database structure |
| READ_SYSTIME | Obtains a consistent copy of the system time and places it into the specified quadword destination |
| INVALIDATE_TB | Flushes a single page-table entry or all page-table entries from the system's translation buffers |

Appendix C includes descriptions of all routines, invoked by the macros listed above, that obtain and release spin locks, fork locks, and device locks. It includes the following routines:

| | |
|---------------|---|
| SMP\$ACQNOIPL | Acquires a device lock, assuming that the local processor is already running at the IPL appropriate for the acquisition of the lock |
| SMP\$ACQUIRE | Acquires a fork lock or a spin lock and enforces the appropriate IPL synchronization on the local processor |
| SMP\$ACQUIREL | Acquires a device lock and enforces the appropriate IPL synchronization on the local processor |
| SMP\$RELEASE | Releases any and all acquisitions of a fork lock or a spin lock by the local processor and makes the lock available for acquisition by other processors |
| SMP\$RELEASEL | Releases any and all acquisitions of a device lock by the local processor and makes the lock available for acquisition by other processors |

Section 15.3 discusses the issues involved in loading a multiprocessing driver into the system, and Section 16.14 describes the VMS synchronization images and their role in testing a multiprocessing driver.

Finally, Appendix A includes full descriptions of two new structures that control the disposition of multiprocessing synchronization and record information regarding the members of the multiprocessing system: the spin lock data structure (SPL) and the per-CPU database structure (CPU).

- The ADPDISP macro, described in Appendix B, provides the ability to transfer control within a driver, based on a specified adapter characteristic. Such adapter characteristics include the presence of mapping registers, bus address size, and the capabilities of the data paths. You should replace most existing occurrences of the CPUDISP macro with ADPDISP.
- This manual now includes a separate part (Part III; Chapters 12 through 18) that discusses hardware-related driver issues and other advanced topics. This structure allows the manual to more clearly and concisely explain the general coding concepts and requirements for all VMS drivers in Parts I and II.
- VMS Version 5.0 allows Q22 bus systems to enable multilevel device interrupt dispatching. Section 12.3.4 describes this new feature as well as VMS interrupt dispatching in general.

New and Changed Features

- VMS Version 5.0 allows Q22 bus device drivers to allocate sets of registers from the entire set of 8,192 map registers. Appendix C describes the following routines that are used to allocate, request, load, and release these alternate map registers. These routines include:

| | |
|---|--|
| IOC\$ALOALTMAP, IOC\$ALOALTMAPN, IOC\$ALOALTMAPSP | Allocate a set of Q22 bus alternate map registers |
| IOC\$LOADALTMAP | Loads a set of Q22 bus alternate map registers |
| IOC\$RELALTMAP | Releases a set of Q22 bus alternate map registers |
| IOC\$REQALTMAP | Allocates sufficient Q22 bus alternate map registers to accommodate a DMA transfer and, if unavailable, place process in alternate-map-register wait queue |

- The executive routines that perform buffer quota checking (EXE\$BUFRQUOTA and EXE\$BUFQUOPRC) have been replaced in VMS Version 5.0 by similar routines that check and debit (or credit) a job's byte count quota and, optionally, its byte count limit. Versions of these routines exist that also allocate a requested buffer.

Appendix C includes discussions of the following buffer quota checking and adjusting routines:

| | |
|---|---|
| EXE\$CREDIT_BYTCNT, EXE\$CREDIT_BYTCNT_BYTLM | Return credit to a job's buffered-I/O byte count quota and byte count limit |
| EXE\$DEBIT_BYTCNT(_NW), EXE\$DEBIT_BYTCNT_BYTLM(_NW) | Determine whether a job's buffered I/O byte count quota usage permits the process to be granted additional buffered I/O and, if so, adjust the job's byte count quota and byte count limit |
| EXE\$DEBIT_BYTCNT_ALO, EXE\$DEBIT_BYTCNT_BYTLM_ALO | Determine whether a job's buffered I/O byte count quota usage permits the process to be granted additional buffered I/O and, if so, allocate the requested amount of nonpaged pool and adjust the job's byte count quota and byte count limit |

- Appendix C includes discussion of the following routines:

| | |
|-----------------|---|
| ERL\$DEVICEATTN | Allocates an error message buffer and logs information relevant to an error that occurs on a device, independent of the I/O request currently being processed |
| LDR\$ALLOC_PT | Allocates system page-table entries |
| LDR\$DEALLOC_PT | Deallocates system page-table entries |

- The SYSGEN CONNECT and LOAD commands have been modified to load a driver image from either SYS\$LOADABLE_IMAGES or SYS\$SYSTEM. DIGITAL recommends that all drivers be placed in the SYS\$LOADABLE_IMAGES directory.

New and Changed Features

- This manual incorporates a description of the VMS terminal driver's class/port interface. Chapter 17 details this architecture and each of its vector entry points; Table A-20 and Figure A-21 illustrate the contents of the UCB terminal extension; and Appendix B describes the CLASS_CTRL_INIT, CLASS_UNIT_INIT, \$VEC, \$VECINI, and \$VECEND macros.
- In VMS Version 5.0, the XDELTA entry IPL has become IPL 14 on all VAX processors. Formerly, it was IPL 5 on VAX uniprocessing systems and IPL 15 on VAX multiprocessing systems.
- The new BREAKPOINTS system parameter, by default, inserts a breakpoint at the end of system initialization. This is in addition to the breakpoint at the beginning of system initialization controlled by the boot flags described in Chapter 16.
- Special pool checking code has been added to the VMS memory allocation and deallocation routines to facilitate detection of pool corruption problems. You can enable this code to troubleshoot problems of this sort by setting the POOLCHECK system parameter, as discussed in Section 16.13.
- The VMS connect-to-interrupt facility, as detailed in Chapter 18, now supports UNIBUS device operations on the VAX 8200/8250/8300/8350 and VAX 8530/8550/8700/8800 processors.
- Section 11.4 and Appendix D describe the role of a driver's cloned UCB routine in I/O processing.
- Appendix A includes a description of all fields that have been added, moved, or modified since VMS Version 4.4.
- Other sections of this manual have been reorganized, corrected, and rewritten as necessary to accurately reflect VMS Version 5.0.

Part I The VMS Device Driver Environment

1

Introduction to Device Drivers

Under the VMS operating system, a *device driver* is a set of routines and tables that the system uses to process an I/O request for a particular device type.

The VMS operating system's approach to I/O is that the 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 VMS operating system provides drivers with the following services:

- A Queue I/O request (\$QIO) system service that preprocesses an I/O request by performing those functions and checks that are common to all devices; for example, validating those arguments of the I/O request that are not device specific
- Many operating system routines that drivers can call to perform I/O preprocessing, allocate and deallocate resources, and synchronize driver execution
- Macros that drivers can invoke to accomplish tasks that would otherwise require many lines of code
- A VMS I/O postprocessing routine that performs device-independent I/O postprocessing for all I/O requests

Thus, drivers can leave the device-independent I/O processing to the operating system and concentrate on servicing those aspects of an I/O operation that vary from device type to device type. In addition, drivers can call VMS system routines to perform many functions that are common to several, but not all, devices.

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

This section first defines the general functions and purposes of a VMS device driver. It then introduces VMS concepts crucial to an understanding of how device drivers work within the operating system and integral to the process of successfully writing one. It concludes with a brief description of the flow of driver activity in servicing an I/O request, using the VMS line printer driver as an example.

Introduction to Device Drivers

1.1 Driver Functions

1.1 Driver Functions

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

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

1.2 Driver Components

Normally, a device driver module can consist of the routines and tables discussed in this section. With a few exceptions, which are noted throughout Chapter 6, the order of the various routines and tables within the driver module is not important.

1.2.1 Driver Tables

The following tables appear in every driver.

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

Section 6.1 describes the procedure for creating a DPT and further discusses its functions. Figure A-10 illustrates the DPT and Table A-9 describes its contents.

The *driver dispatch table* (DDT) lists the addresses of the entry points of standard routines within the driver, and records the size of the diagnostic and error message buffers for drivers that perform error logging. You can find additional information and instructions on how to specify a DDT in Section 6.2. An illustration of the DDT appears in Figure A-9; Table A-8 describes its contents.

Introduction to Device Drivers

1.2 Driver Components

The *function decision table* (FDT) lists all valid function codes for the device, and associates valid codes with the addresses of I/O preprocessing routines, called *FDT routines*. The driver contains device-dependent FDT routines, and the VMS operating system itself provides routines (described in Section 7.5) that perform request preprocessing common to many I/O functions.

When a user process calls the \$QIO system service, the system service uses the I/O function code specified in the request to traverse the FDT and select one or more of these preprocessing routines for execution, as appropriate to the function. To prepare for the actual I/O operation, FDT routines perform such tasks as allocating buffers in system space, locking pages in memory, and validating the device-dependent arguments (**p1** through **p6**) of the \$QIO request. Section 6.3 provides further discussion of the FDT, and Chapter 7 details strategies and rules for writing, specifying, and exiting from an FDT routine.

1.2.2 Driver Routines

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

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

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

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

Details on writing a start-I/O routine appear in Chapter 8. A description of a driver interrupt service routine appears in Chapter 9.

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

The *unit initialization routine* and *controller initialization routine* prepare a device or controller for operation when the VMS driver-loading procedure loads the driver into memory and when the VMS system recovers from a power failure. The amount and type of initialization needed by devices and controllers varies according to the device type and the I/O bus to which the device or controller is attached. Section 11.1 provides additional information about device driver initialization routines.

Introduction to Device Drivers

1.2 Driver Components

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

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

Timeout handling routines are discussed in Section 10.2. Register dumping routines and driver error handling are discussed in Section 11.3.

The VMS operating system calls a driver's *cancel-I/O routine* when a user process issues a Cancel I/O on Channel (\$CANCEL) system service for the device. It may also call the routine when the device's reference count goes to zero, which occurs when all users with assigned channels to the device have deassigned them. The discussion of the cancel-I/O routine appears in Section 11.2.

1.3 The I/O Database

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

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

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

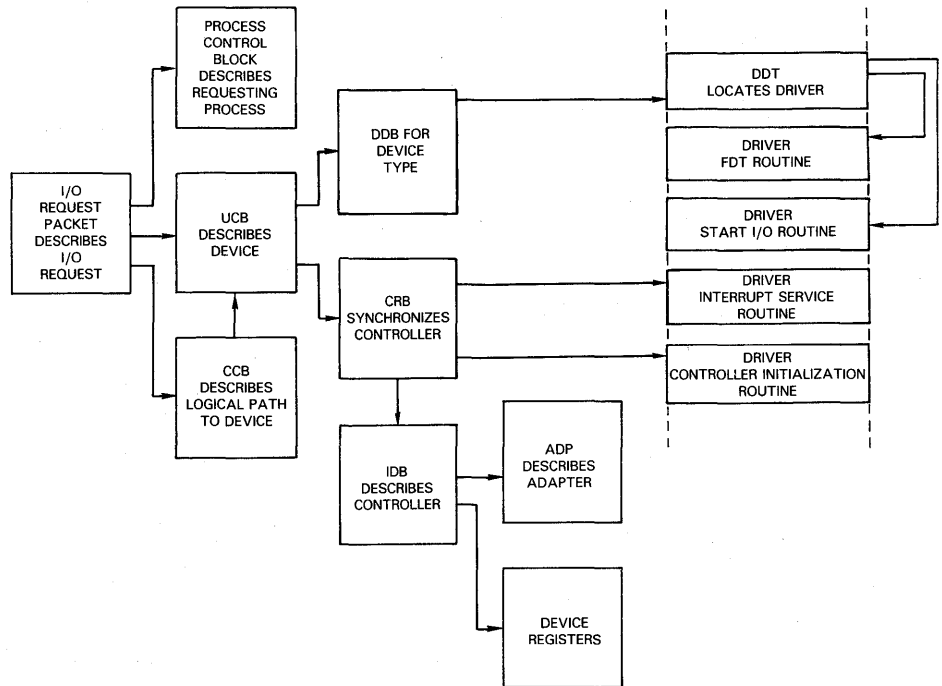
1.3.1 Driver Tables

The three driver tables—driver prologue table, driver dispatch table, and function decision table—are defined in every driver. Section 1.2 lists these tables among the other components of a device driver, and Chapter 6 discusses their contents.

Introduction to Device Drivers

1.3 The I/O Database

Figure 1-1 The I/O Database



ZK-1766-84

1.3.2 Data Structures

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

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

UCBs are the focal point of the I/O database. When a driver is suspended or interrupted, the UCB keeps the context of the driver in a set of fields collectively known as a *fork block*.¹ In addition, the UCB contains the listhead for the queue of pending I/O request packets for the unit.

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

¹ Other structures, such as the CRB, also include a fork block. The discussion of fork blocks and fork processes in Section 1.5 explains the role of fork blocks in driver processing.

Introduction to Device Drivers

1.3 The I/O Database

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

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

An *adapter control block* (ADP) defines the characteristics and current state of an I/O adapter, such as the VAX UNIBUS and MASSBUS adapters, the Q22 bus interface of the MicroVAX 3600-series and MicroVAX II systems, or a device attached to the VAXBI bus. An ADP contains the queues and allocation bit maps necessary to allocate the adapter's resources. VMS provides routines that drivers can call to interface with the appropriate adapter.

The *channel control block* (CCB) describes the logical path between a process and the UCB of a specific device unit.² Each process owns a number of CCBs. When a process issues the Assign I/O Channel (\$ASSIGN) system service, the system writes a description of the assigned device to the CCB.

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

1.3.3 I/O Request Packets

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

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

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

² Channel request blocks and channel control blocks are two separate data structures. To help distinguish the two, it may be helpful to think of the channel request block as the "controller request" block because it describes the hardware controller. In contrast, the channel control block helps manage the logical channel (the **channel** argument to the \$ASSIGN and \$QIO system services) by means of which a process and a device unit accomplish I/O operations.

Introduction to Device Drivers

1.4 Synchronization of Driver Activity

1.4 Synchronization of Driver Activity

Device drivers and other kernel-mode code must maintain synchronization with other priority operating system activities. The term *synchronization* refers to the means by which such code accesses shared data in a consistent, orderly, and predictable fashion. Because there may be more than one processor active in a VMS system, system-level code must synchronize its actions with other code threads it may have preempted on the same (or *local*) processor, as well as with those that are active (or to be activated) on other processors in the system. The VMS operating system uses hardware and software interrupt priority levels (IPLs) to order system events on each local processor in a VAX system. The VAX hardware defines 32 interrupt priority levels (IPLs). The higher numbered IPLs (16 through 31) are reserved for hardware interrupts, such as those posted by devices. The VMS operating system uses the lower numbered IPLs (0 through 15). Code that executes at a higher IPL takes precedence over code that executes at a lower IPL.

A driver, in concert with the operating system, ensures that it maintains system synchronization by performing certain activities and accessing certain data only at the appropriate IPL. In a VMS multiprocessing system, the driver extends the synchronization it achieves by executing locally at a given IPL by acquiring ownership of the spin lock associated with the operation it is performing. (IPL, spin locks, and other forms of synchronization in a VMS system are discussed fully in Chapter 3.)

1.5 Driver Context

As indicated in Section 1.2.2, a driver may have several routines to which the VMS operating system may pass control in certain situations. The context in which any one routine receives control from VMS may differ substantially from that in which another receives control. It is essential that a driver routine not attempt to exceed the limitations of the context in which it executes.

In general, context is characterized by the following factors:

- The current IPL of the executing processor
- The IPL at which the thread of execution that resulted in the call to the driver began
- The currently owned spin locks of the executing processor
- The data structures available to the routine
- Data available to the routine in registers, in data structure fields, and on the stack
- The condition of the registers, data structure fields, and stack when the routine exits
- The ability or inability to access process space

A complete description of the context of each driver routine appears in Appendix D. The following are some general observations:

- All device driver routines execute in kernel mode at an elevated IPL.
- Only driver FDT routines execute within process context and can access process space (P0 and P1).

Introduction to Device Drivers

1.5 Driver Context

- The majority of driver routines execute in *interrupt* (or *system* context): that is, in the sequence of execution that follows a processor's grant of an interrupt request at a given IPL. Such code can refer only to system (S0) space. Moreover, it cannot incur exceptions, including page faults, without causing a fatal bugcheck. Code executing in interrupt context is serviced on the interrupt stack, and must synchronize its execution with other priority code threads by using IPLs, spin locks, and resource wait queues, all of which are described in Chapter 3.

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

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

- Two general purpose registers (R3 and R4)
- The program counter (PC)
- A fork block (usually the unit control block, the address of which is presumed to be in R5 at the time of the suspension) that can contain additional fork process context

VMS places the fork block of a suspended fork process in either a processor-specific fork queue or a resource wait queue where it waits to be resumed. When it resumes the fork process, VMS ensures that the fork context is restored. Fork blocks, fork processes, and fork queues are discussed fully in Section 3.3.3.

1.5.1 Example of Driver Context-Switching

Because a device driver consists of a number of routines that are activated by VMS, the operating system for the most part determines the context in which the routines execute.

As an example, consider the following write request that occurs without error:

- 1 A user process executing in user mode calls the \$QIO system service to write data to a device.
- 2 The \$QIO system service gains control in process context but in kernel mode. It performs device-independent preprocessing of the I/O request.
- 3 The system service uses the driver's function decision table to call the appropriate FDT routines to perform device-dependent preprocessing. These FDT routines execute in full process context in kernel mode.
- 4 When preprocessing is complete, a VMS routine creates a fork process to execute the driver's start-I/O routine in kernel mode.
- 5 The start-I/O routine activates the device unit and suspends itself. At this point, 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.

Introduction to Device Drivers

1.5 Driver Context

- 6 When the device completes the data transfer, it requests an interrupt. The interrupt causes the system to activate the driver's interrupt service routine.
- 7 The interrupt service routine executes to handle the device interrupt. It then causes the start-I/O routine to resume in interrupt context.
- 8 The start-I/O routine regains control in interrupt context but almost immediately issues a request to the operating system to transform its context to that of a fork process. This action dismisses the interrupt.
- 9 When reactivated in fork process context, the start-I/O routine performs device-specific I/O completion and passes control to the system for additional I/O postprocessing.
- 10 VMS I/O postprocessing runs in interrupt context at a lower IPL and issues a special kernel-mode asynchronous system trap (AST) for the user process requesting I/O.
- 11 When the special kernel-mode AST is delivered, the AST routine executes in full process context in kernel mode to deliver data and status to the process. If the original request specified a user-mode AST, the special kernel-mode AST queues it.
- 12 When the user process gains control, the user's AST routine executes in full process context in user mode.

1.6 Hardware Considerations

The VMS operating system runs on any of the following VAX systems:

- VAX 6200 series
- VAX 8530/8550/8700/8800/8830/8840
- VAX 8600/8650/8670
- VAX 8200/8250/8300/8350
- VAX-11/785 and VAX-11/780
- VAX-11/750
- VAX-11/730 and VAX-11/725
- MicroVAX 3600 series
- VAXstation 2000/MicroVAX 2000
- MicroVAX II
- MicroVAX I

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

Introduction to Device Drivers

1.6 Hardware Considerations

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

1.6.1 Driver Dependency on VAX Processing Systems

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

1.6.1.1 VAX-11/780, VAX-11/785, and VAX 8600/8650/8670

The VAX-11/780, VAX-11/785, VAX 8600, VAX 8650, and VAX 8670 systems, from the viewpoint of I/O architecture, are SBI-based systems. That is, the *synchronous backplane interconnect* (SBI) is the bus by which I/O adapters communicate with main memory and the central processor (see Figure 1-2). I/O adapters supported by the SBI include the UNIBUS adapter (UBA), MASSBUS adapter (MBA), and the DR780 interface adapter. Correspondingly, peripheral devices attach to either the UNIBUS, MASSBUS or DR32 device interconnect (DDI) of the DR780 adapter. Main memory shares the SBI with the I/O adapters on the VAX-11/780 and VAX-11/785. The VAX 8600, VAX 8650, and VAX 8670 employ a separate bus to which main memory is attached and can each be configured with up to two SBIs for I/O adapters.

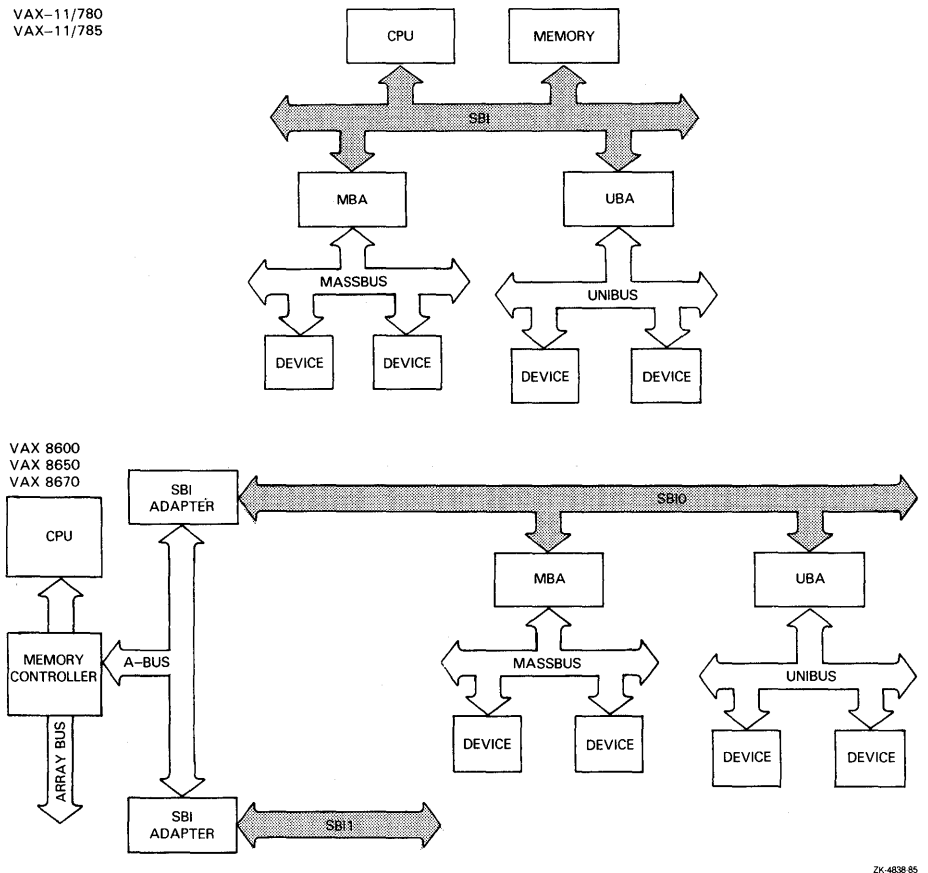
For these systems, nonstandard devices are commonly attached to the UNIBUS, although some nonstandard devices connect to the MASSBUS and DDI. The components of UNIBUS and MASSBUS drivers are nearly identical and the strategies for producing driver code are similar; writers of either type of driver will profit from reading the bulk of this manual. Writers of UNIBUS drivers can find specific information about the UNIBUS adapter and VMS support for UNIBUS drivers in Chapter 12. MASSBUS driver writers should refer to Chapter 13 for similar information about the MASSBUS. DIGITAL supplies a device driver and an application library for DDI devices; the *VMS I/O User's Reference Manual: Part II* discusses the DR32 interface driver in detail.

A final note on terminology regarding these systems is pertinent. For the purposes of the discussion in this book, the term *VAX-11/780* refers to the family of VAX systems that includes the VAX-11/780 and VAX-11/785; the term *VAX 8600* refers to the VAX 8600, VAX 8650, and VAX 8670.

Introduction to Device Drivers

1.6 Hardware Considerations

Figure 1-2 SBI-Based System Configurations



1.6.1.2 VAX-11/750

The VAX-11/750 system resembles the VAX-11/780-type systems in that it supports UNIBUS, MASSBUS, and DDI peripheral devices (see Figure 1-2). The backplane, or CPU-to-memory interconnect (CMI), by which I/O adapters communicate with the central processor and main memory, is integral to the processor, as are the UNIBUS interface (UBI) and MASSBUS adapter (MBA). The DR750 interface adapter connects the CMI to the DDI subsystem. Peripheral devices connect to the UNIBUS, MASSBUS, and DDI. A separate memory interconnect provides an interface between main memory and the rest of the system.

For the VAX-11/750, nonstandard devices are commonly connected to the UNIBUS, although some nonstandard devices attach to the MASSBUS. The components of UNIBUS and MASSBUS drivers are identical, and the strategies for developing driver code are similar. Writers of either type of driver will profit from reading this manual. Writers of UNIBUS drivers can find specific information about the UNIBUS adapter and VMS support for UNIBUS drivers in Chapter 12. MASSBUS driver writers should refer to Chapter 13 for similar information about the MASSBUS. DIGITAL supplies a device driver and an application library for DDI devices; the *VMS I/O User's Reference Manual: Part II* discusses the DR32 interface driver in detail.

Introduction to Device Drivers

1.6 Hardware Considerations

1.6.1.3 VAX-11/730 and VAX-11/725

The VAX-11/730 and VAX-11/725 systems, like the VAX-11/750, incorporate an integral UNIBUS adapter to control transactions between UNIBUS peripheral devices, the processor, and the main memory interface. The VAX-11/730 and VAX-11/725, however, do not support MASSBUS devices. Writers of UNIBUS drivers can find specific information about the UNIBUS adapter and VMS support for UNIBUS drivers in Chapter 12. For the purposes of the discussion in this book, the term *VAX-11/730* refers to both the VAX-11/730 and the VAX-11/725.

1.6.1.4 VAX 8200/8250/8300/8350, VAX 8530/8550/8700/8800/8830/8840, and VAX 6200 Series

The VAX 8200/8250/8300/8350, VAX 8530/8550/8700/8800/8830/8840 and VAX 6200 series are VAXBI-based systems; that is, the VAXBI is the bus by which I/O adapters communicate with main memory and the central processor (see Figure 1-3).

In a VAX 8200/8250/8300/8350 configuration, main memory, the DWBUA, and other devices are all connected directly to the VAXBI bus. By contrast, the VAX 8530/8550/8700/8800/8830/8840 and VAX 6200-series configurations employ separate memory interconnects (known as the NMI, PBI, or XMI, as illustrated in Figure 1-3, to service main memory. The VAX 8530/8550/8700/8800 provides multiple VAXBI buses to which I/O adapters and devices can be attached. The VAX 8300, VAX 8350, VAX 8800/8830/8840, and VAX 6200 series are multiprocessor systems.

The VAXBI bus supports UNIBUS peripherals by means of the BI-to-UNIBUS adapter (DWBUA). Writers of UNIBUS drivers can find specific information about the UNIBUS adapter and VMS support for UNIBUS drivers in Chapter 12.

The VAXBI also supports non-DIGITAL-supplied devices designed according to specifications established by DIGITAL and a license granted by DIGITAL. Writers of drivers for such devices, referred to as *generic VAXBI devices* in this manual, can find specific discussion in Chapter 14.

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

Introduction to Device Drivers

1.6 Hardware Considerations

Figure 1-3 VAXBI-Based System Configurations

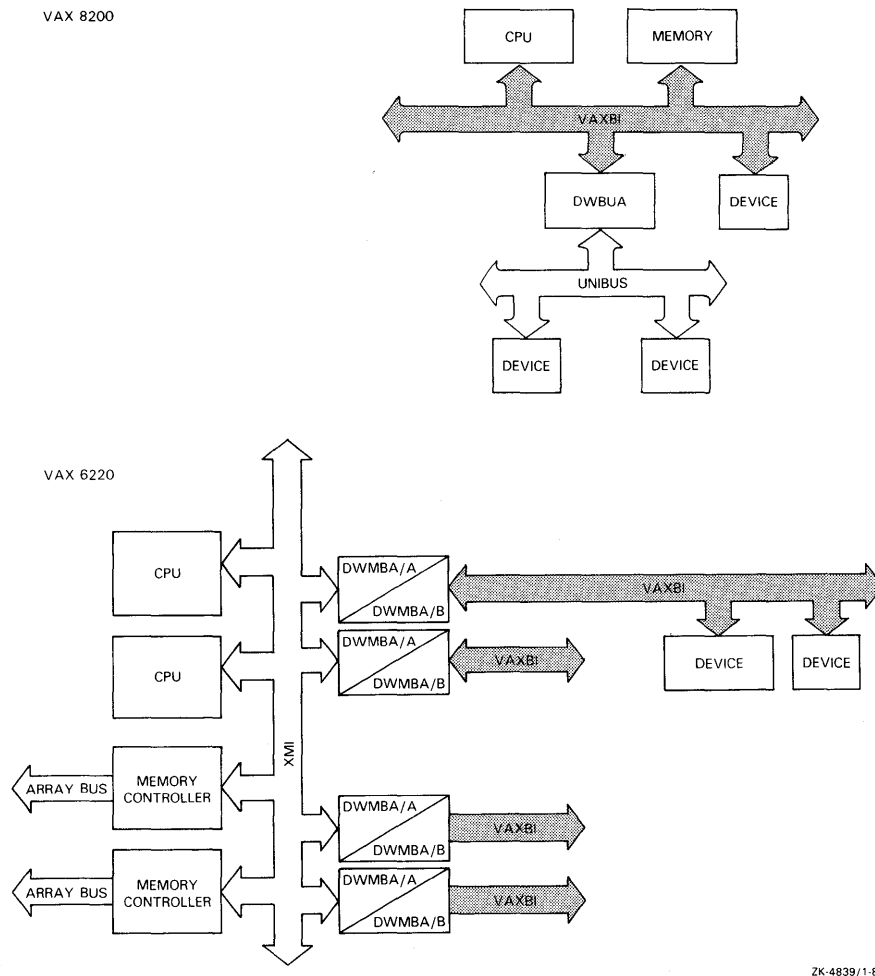
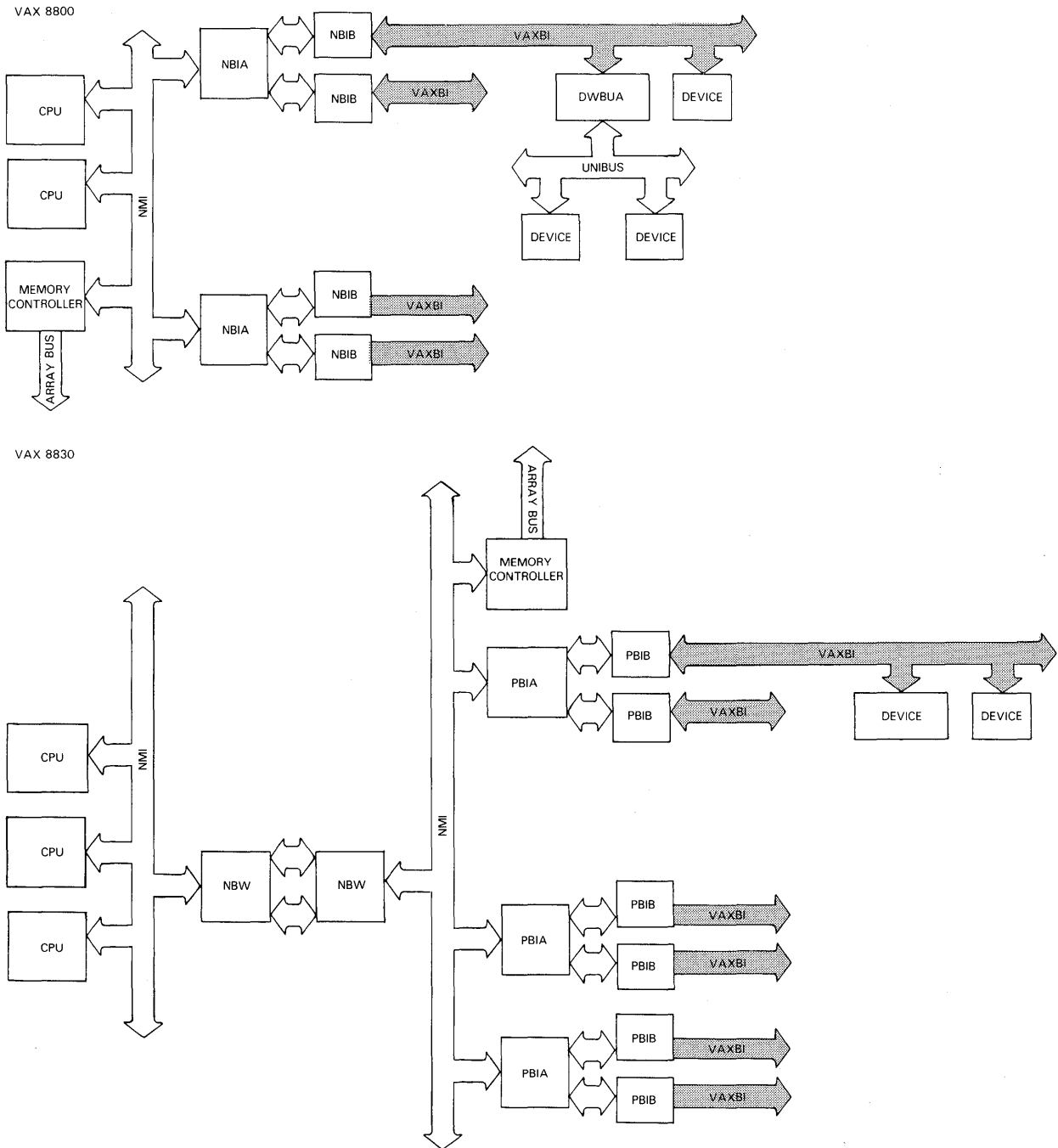


Figure 1-3 Cont'd. on next page

Introduction to Device Drivers

1.6 Hardware Considerations

Figure 1-3 (Cont.) VAXBI-Based System Configurations



ZK-4839/2-85

Introduction to Device Drivers

1.6 Hardware Considerations

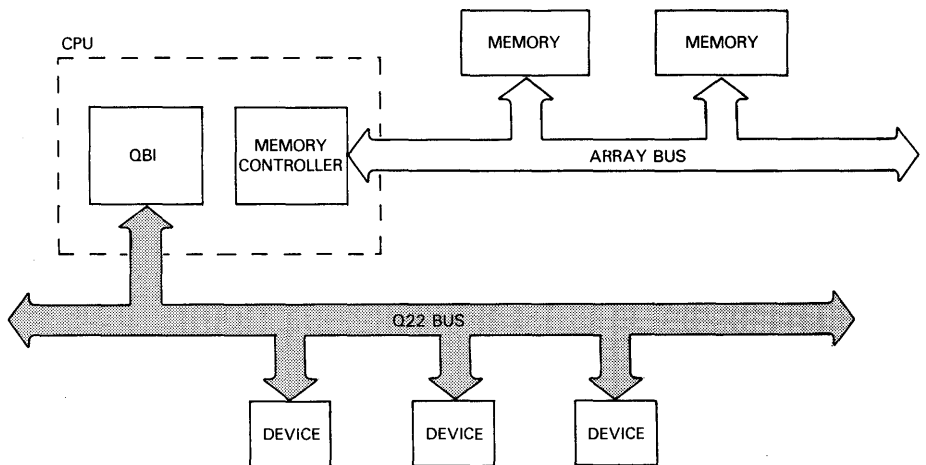
1.6.1.5 MicroVAX 3600 Series and MicroVAX II

The MicroVAX 3600 series and MicroVAX II are Q22 bus-based systems. On these systems, the Q22 bus is the bus by which peripheral devices communicate with main memory and the processor.³ Q22 bus device drivers are sufficiently similar to those that drive UNIBUS devices that most of the discussion of UNIBUS drivers in this book can equally pertain to the writing of Q22 bus device drivers (see Chapter 12 for a discussion of the similarities and differences).

As you can see in Figure 1-4, in these systems main memory and I/O devices reside on separate interconnects. The MicroVAX 3600-series and MicroVAX II systems implement a scatter-gather map containing 8,192 map registers that allows devices to perform multiple-block direct-memory-access (DMA) transfers.⁴

For the purposes of discussion in this manual, the term *backplane interconnect* represents the Q22 bus in the MicroVAX 3600-series and MicroVAX II systems. The term *Q22 bus interface* represents those functions performed by these processors that resemble those performed by the UNIBUS adapter of other VAX systems. In most instances, you can assume that discussions of the UNIBUS adapter apply as well to the Q22 bus.

Figure 1-4 MicroVAX 3600-Series and MicroVAX II System Configuration



ZK-4840-85

³ DMA controllers attached to the Q22 bus must be capable of 22-bit addressing.

⁴ In these systems, the 4MB of Q22 bus memory is located from physical address 30000000_{16} to $303F0000_{16}$. If you must install controllers that contain local memory on the Q22 bus, it is best to install them in the upper 3 3/4 MB of Q22 bus memory (after physical address 30040000_{16}). The first 1/4 MB of Q22 bus memory contains 496 map registers, 127 of which must be free for use by VMS in booting. If you must place a controller containing memory in this address region, it cannot occupy more than 369 pages. If the controller exceeds this space, VMS will probably boot but will not be able to take crash dumps.

Introduction to Device Drivers

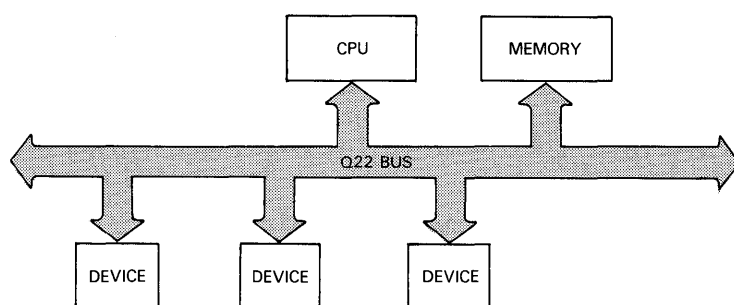
1.6 Hardware Considerations

1.6.1.6 MicroVAX I

The MicroVAX I is a Q22 bus-based system; that is, the Q22 bus is the bus by which peripheral devices communicate with main memory and the processor. Q22 bus device drivers are sufficiently similar to those that drive UNIBUS devices that most of the discussion of UNIBUS drivers in this book can equally pertain to the writing of Q22 bus device drivers (see Chapter 12 for a discussion of the similarities and differences).

MicroVAX I main memory and I/O devices exist together on the same bus (see Figure 1-5). The effects of the absence of a scatter-gather map on DMA device drivers are discussed in Section 12.2.8.⁵

Figure 1-5 MicroVAX I System Configuration



ZK 4853-95

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

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

⁵ The MicroVAX I uses the 22-bit Q22 bus to address both main memory and Q22 bus memory. Because MicroVAX I main memory shares the Q22 bus with I/O devices, the maximum amount of address space available for main memory (4MB at most) is correspondingly decreased whenever controllers containing memory are attached to the Q22 bus. For instance, if a controller containing a 256K bit map is installed on the Q22 bus, 3 3/4 MB would remain for main memory. VMS is effectively prevented from using as main memory those locations addressable as controller memory by the appropriate setting of the PHYSICALPAGES system parameter. In the preceding example, PHYSICALPAGES would be set to 7680 to prevent the double mapping of the 256K bit map as both main memory and controller memory.

Introduction to Device Drivers

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

1.7.1 Programmed I/O

Drivers for relatively slow devices, such as printers, card readers, terminals, and some disk and tape drives, must transfer data to a device register a byte or a word at a time. These drivers must themselves keep a record of the location of the data buffer in memory, as well as a running count of the amount of data that has been transferred to or from the device. Thus, these devices perform *programmed I/O* (PIO) in that the transfer is largely conducted by the driver program.

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

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

Drivers performing PIO transfers are generally not concerned with the operation of I/O adapters. However, drivers that perform direct-memory-access (DMA) transfers must take into account I/O adapter functions, as discussed in Section 1.7.2.

1.7.2 Direct-Memory-Access I/O

Devices that perform *direct-memory-access* (DMA) transfers do not require the central processor so frequently. Once the driver activates the device, the device can transfer a large amount of data without requesting an interrupt after each of the smaller amounts. The responsibilities of a driver for a DMA device involve supplying a device register with the starting address of the buffer containing the data to be transferred, a byte offset into the buffer, and the size of the transfer. By setting the appropriate bit or bits in the device control and status register (CSR), the driver activates the device. The device then automatically transfers the specified amount of data to or from the specified address. The VMS drivers DLDRIVER and XADRIVER are examples of DMA drivers, and appear in full in Appendixes E and F, respectively.

For DMA transfers, UNIBUS drivers and MicroVAX 3600-series/MicroVAX II drivers must first map the transfer from main memory to I/O bus memory space. The result of this mapping is a set of contiguous addresses in UNIBUS or Q22 bus space that the DMA device can access to successfully perform a DMA transfer. To accomplish this, a driver must first obtain map registers, and, optionally for UNIBUS drivers, a buffered data path. The driver calls VMS routines that interface with the I/O adapter to allocate these resources on behalf of the driver. Chapter 12 discusses the operation of the UNIBUS adapter and the Q22 bus. Section 12.2 provides instructions on how to write a DMA driver for UNIBUS and Q22 bus devices.

Introduction to Device Drivers

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

The MicroVAX I Q22 bus has no map registers, so no mapping of physical bus addresses to virtual memory addresses is possible. As a result, a driver for a device attached to the MicroVAX I Q22 bus that performs DMA transfers must include special logic that either allocates a physically contiguous buffer from nonpaged pool for use in the transfer or segments the transfer at page boundaries. Section 12.2.8 discusses the strategies for producing MicroVAX I DMA drivers.

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

The method by which a generic VAXBI device capable of DMA transfers accomplishes such a transfer depends upon the characteristics of the device. Several methods are discussed in Section 14.5.

1.8 Buffered and Direct I/O

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

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

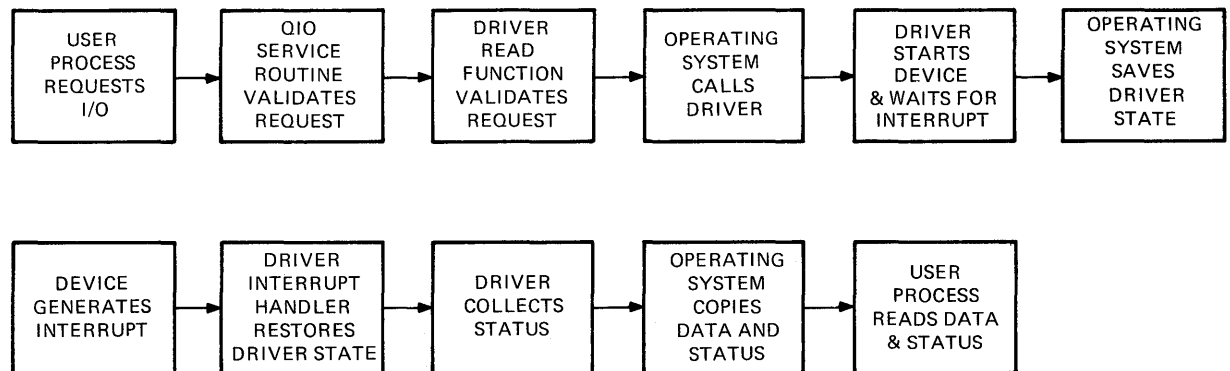
- *Direct I/O*, the technique used most commonly by drivers of DMA devices, locks the user buffer in memory as well as the page-table entries that map it. The function decision table (FDT) of such a driver calls a VMS-supplied FDT routine that prepares the user buffer for direct I/O.
- *Buffered I/O*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

The trade-off between buffered I/O and direct I/O is the time required to move the data into the user's buffer as against the time required to lock the buffer pages in memory. Sections 6.3.1 and 7.4 provide additional information.

1.9 Example of an I/O Request

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

Figure 1-6 Example of I/O Request Processing



ZK-909-82

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

1 A process requests an I/O operation.

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

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

2 The operating system performs I/O preprocessing.

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

3 The driver performs I/O preprocessing.

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

4 VMS creates a driver's fork process.

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

Introduction to Device Drivers

1.9 Example of an I/O Request

5 The driver readies the I/O adapter.

For DMA transfers, the driver's fork process calls VMS routines that enable the I/O adapter hardware to map I/O bus addresses into physical addresses for the transfer. (Note that the MicroVAX I system does not have this capability, as discussed in Section 12.2.8.)

6 The driver activates the device.

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

7 The driver waits for an interrupt.

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

8 The device requests an interrupt.

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

9 The driver services the interrupt.

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

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

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

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

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

12 The driver completes the I/O operation.

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

13 VMS completes the I/O operation.

The VMS I/O postprocessing routines copy the I/O status into process address space, general registers, or both, and return control to the user process.

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

Introduction to Device Drivers

1.9 Example of an I/O Request

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

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

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

2

Discussion of a \$QIO Request

This chapter outlines the series of activities performed by the VMS operating system and a simple device driver in order to process an I/O request. The LP11 line printer driver (LPDRIVER) was selected for this discussion because it is a simple driver but still illustrates many driver principles. The first-time reader of this document might not understand all of the points made in this chapter; however, the chapter should provide some insight into driver flow and I/O processing.

The LP11 printer is a PIO device (see Section 1.7.1). Although the LP11 is usually spooled, this discussion assumes that it is not.

A user process can request the following functions on this printer:

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

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

- The portions of the line printer driver that are used in servicing a write request
- The VMS components with which the driver interacts to process the write request

Figure 2-1 illustrates the flow of execution through the VMS executive routines and printer driver code that satisfies an I/O request. The unshaded boxes in Figure 2-1 indicate the processing performed by driver subroutines. Boxes shown above the solid line indicate processing in the context of the user process. Boxes below the line indicate processing in fork or interrupt context.

2.1 Driver Code for the LP11 Write Function

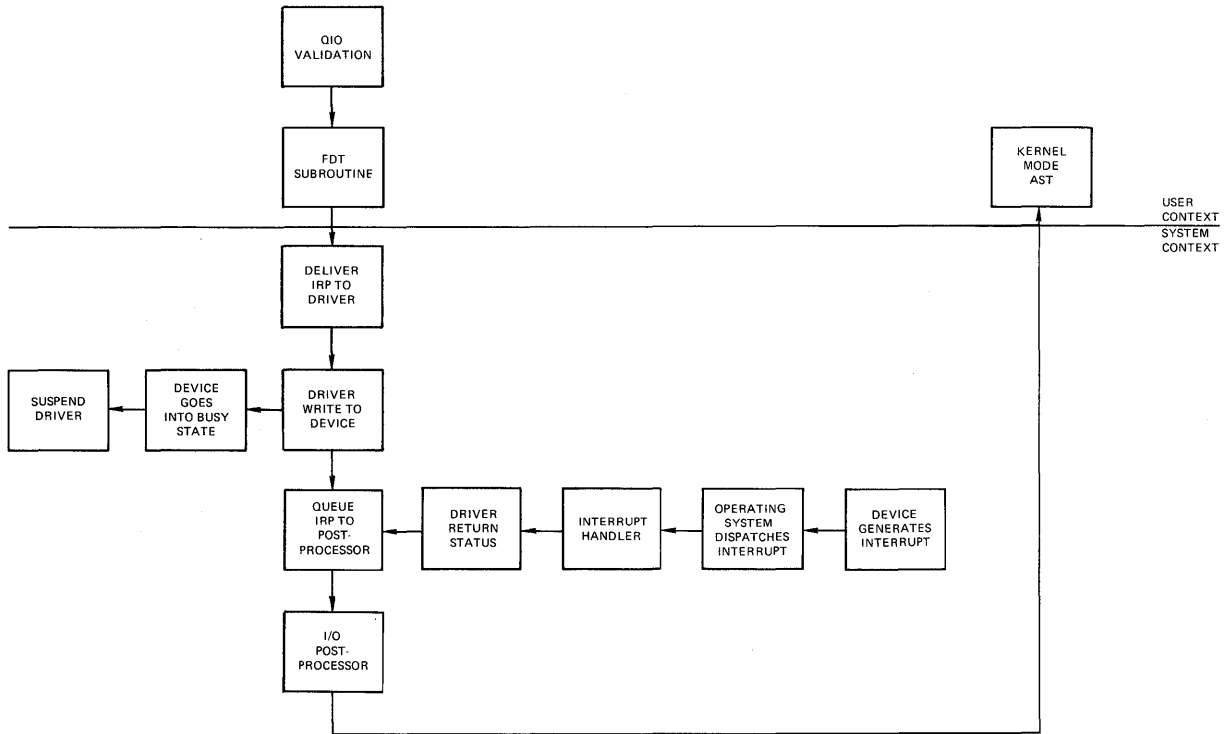
The VMS device driver for an LP11 printer implements a write function using the following parts of the driver:

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

Discussion of a \$QIO Request

2.1 Driver Code for the LP11 Write Function

Figure 2-1 A Printer Write Function



Z6-911-82

2.2 A User Process's I/O Request

A user process writes a line to the printer by calling the Queue I/O Request (\$QIO) system service, specifying the write-virtual-block function code as follows:

```
$QIO_S chan = CHANNEL_NUMBER, -
func = #IO$WRITEVBLK, -
efn = #6, -
iosb = STATUS_BLOCK, -
p1 = BUFFER_ADDRESS, -
p2 = #BUFFER_SIZE, -
p4 = #^X30
```

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

Discussion of a \$QIO Request

2.3 Device-Independent I/O Preprocessing by VMS

2.3 Device-Independent I/O Preprocessing by VMS

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

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

CCB → UCB → DDT → FDT

- The driver FDT must list IO\$_WRITEVBLK as a valid function for the device.
- The event flag number must be valid.
- The process's remaining buffered I/O count (BIOCNT) must permit the \$QIO system service to perform a buffered-I/O request.
- The process must have write access to location STATUS_BLOCK, specified in the request for use as an I/O status block.

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

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

2.4 Device-Dependent I/O Preprocessing by the Driver

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

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

- It calculates the length of the required system space buffer.
- If the job byte count quota for buffered I/O (JIB\$_L_BYTCNT) permits, the routine allocates a buffer from system address space, stores the address of the buffer in the IRP, and decreases the current job byte count quota.
- It then synchronizes access to the printer's UCB by obtaining its mutex (UCB\$_L_LP_MUTEX) for write access. It can thus reliably preprocess the write request, depending upon information contained in the UCB.

Discussion of a \$QIO Request

2.4 Device-Dependent I/O Preprocessing by the Driver

By obtaining the line printer mutex, the driver FDT routine effectively prevents processes active in a VMS multiprocessing system from initiating simultaneous functions on the printer. Also, in a VMS uniprocessing system, this action prevents contention between a process that has allocated the printer (and has been preempted in the midst of a write function) and any of its subprocesses that, when scheduled, may attempt to start a concurrent function that alters device characteristics.

- It reads the description of the printer's current line and page position from the device's UCB.
- It reformats the data from the process buffer into the system buffer, adding carriage control characters, as specified in argument **p4** to the I/O request, before and after the data.

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

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

All of the I/O processing described to this point occurs in the context of the user's process. The user address space is mapped, and the processor's IPL is still low enough to permit process scheduling and paging. Subsequent queuing of the transfer request to the driver and all resulting driver processing occur at higher IPLs—and with ownership of the appropriate fork lock and device lock in a VMS multiprocessing environment—that synchronize the driver's handling of the device. (See Chapter 3 for a discussion of the concept of synchronization.)

2.5 Queuing the I/O Request Packet to the Driver

Before queuing the IRP to the printer driver, the VMS queuing routine raises the IPL to the driver's fork level and obtains the associated fork lock in a VMS multiprocessing environment. These actions synchronize access to those fields of the UCB referenced by driver routines at fork IPL.

If the device is idle, which is to say that if the busy bit in the UCB status longword (UCB\$V_BSY in UCB\$L_STS) of the UCB is clear, VMS can transfer control to the driver. The driver dispatch table (DDT) contains the entry point to the driver's start-I/O routine. To find the proper entry point, the queuing routine chains through the I/O database to the DDT, as follows:

UCB → DDT → start-I/O routine

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

Discussion of a \$QIO Request

2.6 Activating the Printer

2.6 Activating the Printer

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

The driver routine delivers characters to the printer according to the following sequence:

- 1 The driver locates the LP11 device registers using a chain of pointers starting at the device's UCB.

UCB → CRB → IDB → CSR address

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

- 2 The driver examines the device's CSR to see if the device is ready to accept characters.
- 3 If the device is ready, the driver writes a byte of data to the printer's data buffer register. The printer controller moves the byte from the register to the controller's internal print silo.
- 4 The driver decreases the count of bytes to transfer and repeats step 2.
- 5 If the device is not ready (that is, its print silo is full), the driver raises IPL to device IPL and obtains the corresponding device lock in a VMS multiprocessing system. These actions allow it to set the interrupt-enable bit in the device's CSR in synchronization with other routines in the driver that may access the CSR.

After setting the interrupt-enable bit, the driver invokes a VMS wait-for-interrupt macro to release the device lock and suspend driver processing until the printer requests an interrupt or the device times out.

2.7 Waiting for a Device Interrupt

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

- Saving driver context (R3, R4, and the address of the next instruction in the driver) in the device's UCB
- Calculating the time at which the device will time out
- Setting bits in the device's UCB to indicate that the driver expects a device interrupt within a specified time period
- Releasing the device lock in a VMS multiprocessing system, restoring IPL to fork level, and returning control to the caller of the driver's start-I/O routine

Discussion of a \$QIO Request

2.7 Waiting for a Device Interrupt

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

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

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

2.8 Handling Interrupts

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

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

- 1 Restores the address of the UCB in R5
- 2 Obtains the appropriate device lock to ensure synchronization in a VMS multiprocessing environment
- 3 Confirms that the interrupt was expected by examining bits in the device's UCB
- 4 Restores the saved registers (R3 and R4) from the device's UCB
- 5 Transfers control to the driver PC address stored in the device's UCB

Rather than execute in interrupt context, the reactivated driver routine calls a VMS routine to create a fork process. As a result of this action, VMS again suspends driver processing by performing the following steps:

- 1 Saving driver context (R3, R4, and the driver PC address) in the device's UCB
- 2 Inserting the UCB address in the appropriate fork queue in the local processor's CPU database

The driver suspension allows the operating system to reschedule driver processing at its fork IPL and permits higher priority code to execute and device interrupts to be serviced while driver processing of the I/O request concludes. The VMS fork dispatcher reactivates the driver when the IPL of the local processor drops to fork level.

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

Discussion of a \$QIO Request

2.9 I/O Postprocessing by the Driver

2.9 I/O Postprocessing by the Driver

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

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

2.10 I/O Postprocessing by VMS

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

- 1 Increases the use count (PHD\$_L_BIOCNT) of the process's buffered I/O requests because the current operation is complete. The use count is maintained for accounting purposes.
- 2 Decreases the process's buffered I/O count (PCB\$_W_BIOCNT) to reflect a completed buffered I/O operation. This operation restores buffered-I/O quota to the process.
- 3 Deallocates the system buffer used for the reformatted user data.
- 4 Increases the job's byte count quota.
- 5 Sets an event flag to indicate that the I/O operation is complete.
- 6 Queues a special kernel-mode AST routine that will deallocate the IRP and stores I/O status in the user's I/O status block.

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

3 Synchronization of I/O Request Processing

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

This chapter contains the following discussions:

- Section 3.1 discusses the interrupt priority levels, focusing on those IPLs and interrupt service routines that participate in the processing of an I/O request. It briefly examines the roles of the other IPLs in the operating system. Whether you are writing a driver for a VMS uniprocessor or multiprocessor environment, you must adhere to the synchronization rules discussed in this section.
- Section 3.3 illustrates how system synchronization is maintained during the processing of an I/O request on any VAX system. As part of this discussion, this section describes the driver fork process and the activity of forking. Finally, it examines the methods by which a driver synchronizes at fork level and device interrupt level.
- Section 3.4 discusses the mechanism by which driver code stalls to wait for an available adapter or controller resource on any VAX system.

3.1 Interrupt Priority Levels

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

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

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

Synchronization of I/O Request Processing

3.1 Interrupt Priority Levels

Table 3-1 IPLs Defined by VMS

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

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

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

- Power failure
- Processor errors
- Device interrupts
- Device driver fork processing
- I/O postprocessing
- Process rescheduling
- AST delivery

As a result of blocking events on and ordering the activities of a single VAX processor, VMS use of IPLs ensures that kernel-mode code accesses data in memory in a cooperative and predictable manner. The mechanism by which synchronized access to data is ensured is twofold. First, VMS associates a given IPL with the access of one or more data structures or databases.

Synchronization of I/O Request Processing

3.1 Interrupt Priority Levels

Secondly, VMS defines an ordered set of semaphores, called *spin locks*, that extend IPL synchronization throughout a VMS multiprocessing system. A processor must obtain one or more of these spin locks before executing any code thread that must make use of the resources the spin lock protects. Spin locks thus allow each processor in a VMS multiprocessing system to share common system data and block events systemwide.

For example, consider a code thread running at IPL 8 that intends to access the memory management database. To do so, it raises IPL to IPL\$_MMG. This action gives it the exclusive right to access the database from the local processor, effectively preventing access by other code threads on the same processor. After raising IPL, this code thread requests the memory management (MMG) spin lock. Ownership of the MMG spin lock gives the processor executing this thread the exclusive right to access the database systemwide, and bars access from any other code thread running on any other processor in the VAX system.

Although discussions in this book treat IPL and spin lock synchronization as conceptually separate tasks for a device driver, the set of VMS synchronization macros, described in Table 3-2, makes adjustment of IPL and disposition of spin locks appear as a single operation.

A full description of spin locks appears in Section 3.2.

3.1.1 Interrupt Service Routines

VMS associates certain IPLs with the execution of certain tasks. Moreover, when a processor in a VAX system grants an interrupt at a given IPL, the grant actually triggers the execution of a specific piece of code, called an interrupt service routine, that performs the task.

Device drivers themselves contain an interrupt service routine which handles device interrupts at an appropriate device IPL (IPLs 20 through 23). In addition, drivers rely heavily upon the VMS interrupt service routine known as the *fork dispatcher* which runs at several IPLs, including driver fork IPLs 8 through 11. When the local processor's IPL drops to fork IPL, it is the fork dispatcher that restores the context of the driver fork process and places it into execution. (See Section 3.1.2.4 and 3.3.2 for discussions of the device IPLs and interrupt dispatching, respectively. Sections 3.1.2.3 and 3.3.3 discuss the fork IPLs and driver fork processes.)

3.1.2 IPL Use During I/O Processing

The activities essential to the processing of an I/O request occur only at certain IPLs. VMS performs some of these tasks in system routines and interrupt service routines; drivers perform others. This section describes those IPLs and interrupt service routines that are most involved in I/O processing. Section 3.1.3 discusses the IPLs at which other system activities transpire that may influence the coding of a driver. For additional information on the pattern of synchronization throughout the servicing of an I/O request, see Section 3.3.

Synchronization of I/O Request Processing

3.1 Interrupt Priority Levels

3.1.2.1 IPL 2 (IPL\$_ASTDEL)

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

When an AST is specified for delivery to a process, the AST queuing routine (SCH\$QAST) queues the AST to the specified process's process control block (PCB).¹ When an AST is delivered is determined by the mode of the AST, the current mode of the processor, and the mode contained in the processor's ASTLVL register. The VAX hardware, by means of the REI instruction, requests a software interrupt on the local processor at IPL\$_ASTDEL whenever the processor's mode becomes less privileged than that specified as its ASTLVL.²

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

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

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

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

3.1.2.2 IPL 4 (IPL\$_IOPOST)

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

After it has completed whatever device-dependent postprocessing is required, a driver fork process requests I/O postprocessing by calling a VMS routine (COM\$POST) that inserts an IRP in the local processor's postprocessing queue (at CPU\$_PSBL) and requests a software interrupt at IPL\$_IOPOST. When IPL drops below IPL 4, the IPL\$_IOPOST interrupt service routine

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

² In the event that a processor queues an AST to a process currently executing on another processor in a multiprocessing system, the local processor generates an interprocessor interrupt to the other processor to change its ASTLVL.

Synchronization of I/O Request Processing

3.1 Interrupt Priority Levels

dequeues an IRP from the I/O postprocessing queue at CPU\$_PSFL, performs all I/O-completion tasks that can occur without reference to the device's unit control block (UCB) and, thus, at an IPL lower than fork IPL.

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

3.1.2.3 IPL 8 Through IPL 11 (Fork IPLs)

On each processor in a VAX system—for each of the IPLs from 8 to 11—there exists a queue for fork blocks waiting to be processed. Each fork block contains the context of a suspended fork process. The interrupt service routine that executes at each of these IPLs (EXE\$FORKDSPATH) is known as the *fork dispatcher*. The fork dispatcher dequeues a fork block, obtains the appropriate fork lock, restores the context of the fork process, and resumes its execution at the PC location saved in the fork block (at FKB\$_FPC). (Refer to Section 3.3.3 for a discussion of fork blocks and fork processes.)

All driver routines, except most FDT routines, execute at fork IPL or higher. Usually driver routines should not read or alter UCB fields without taking steps to ensure synchronization. Because such UCB fields can be shared among driver fork processes and VMS system tasks executing on other processors in a VMS multiprocessing system, a processor must first secure the corresponding fork lock to execute at that fork IPL. Furthermore, the drivers for all devices on a single I/O adapter must use the same fork lock if they actively compete for shared I/O adapter resources such as map registers and data paths. The VMS routine that initiates an I/O request on an idle device unit, as well as the fork dispatcher, transfers control to the driver with the appropriate synchronization.

A driver places a fork lock index in UCB\$_FLCK using the DPT_STORE macro. (See Section 6.1.) VMS determines the appropriate fork IPL from the contents of the SPL\$_IPL field in the fork lock's structure. (See Section 3.2 for a discussion of spin locks.)

3.1.2.4 IPL 20 Through IPL 23 (Device IPLs)

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

The *interrupt dispatcher* routes interrupts from devices to the appropriate device driver's interrupt service routine. A driver specifies the address of its interrupt service routine in the driver prologue table (DPT). The interrupt dispatcher's routing mechanism works differently depending upon the VAX processor and I/O subsystem in use. (For additional information on device interrupt dispatching, see the general discussion of interrupt dispatching in Chapter 9. Information specific to a given I/O subsystem configuration

Synchronization of I/O Request Processing

3.1 Interrupt Priority Levels

appears in Sections 12.3 (UNIBUS/Q22 bus), 13.4 (MASSBUS), and 14.3.1 (VAXBI bus.)

Data in a device's registers and in various fields of the UCB that record device status is synchronized on the local processor at device IPL, at which its driver's interrupt service routine executes. This value is stored by the driver in the UCB\$B_DIPL field of the UCB. It is the responsibility of the interrupt service routine to secure the corresponding device lock. This action allows it to synchronize with other code threads that access the same resources in a VMS multiprocessing system.

The driver's start-I/O routine is one such code thread and must similarly synchronize. In a VMS uniprocessing environment, the routine raises IPL to device IPL before writing data in device registers and database fields. In a VMS multiprocessor environment, the start-I/O routine must secure the appropriate device lock to achieve systemwide synchronization of the device database. The act of acquiring the device lock automatically sets IPL to device IPL.

Because code executing at IPLs 20 through 23 blocks most other hardware interrupts and all software interrupts on the local processor, driver code lowers its IPL as soon as possible. Interrupts from devices on a MicroVAX 3600-series, MicroVAX II, MicroVAX I, VAX 8200/8250/8300/8350, VAX 8530/8550/8700/8800/8830/8850, and VAX 6200-series system, in fact, can block hardware interrupts from the processor's interval clock if these device interrupts occur at or above IPL 22. To prevent the loss of an interval clock interrupt, these drivers, when executing at IPL 22 or above, should lower IPL below 22 as soon as possible (within 9 milliseconds).

3.1.2.5 IPL 31 (IPL\$_POWER)

The highest IPL, IPL\$_POWER (IPL 31) locks out all other interrupts on the local processor. Many 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. In a VMS multiprocessing system, these routines often need to acquire additional synchronization, as described in Section 3.2.

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

3.1.3 Additional IPLs

In addition to the IPLs that are directly involved in the processing of an I/O request, VMS defines the IPLs described in this section.

Synchronization of I/O Request Processing

3.1 Interrupt Priority Levels

3.1.3.1 IPL 3 (IPL\$_RESCHED)

When an event occurs that requires that a process be rescheduled, a VMS routine requests a software interrupt on the local processor at IPL\$_RESCHED. The scheduler interrupt service routine (SCH\$RESCHED) gains control at this IPL, but immediately obtains the SCHED spin lock (as a result, raising IPL to IPL\$_SYNCH). This action synchronizes the processor's access to the scheduler's database with other system activities.

Drivers never explicitly reference IPL\$_RESCHED. Most driver processing occurs at higher IPLs. When a process raises IPL to or above IPL\$_RESCHED, the scheduler cannot reschedule the process. The process runs until an interrupt occurs at a higher IPL or the process lowers IPL below IPL\$_RESCHED.

3.1.3.2 IPL 6 (IPL\$_QUEUEAST)

IPL\$_QUEUEAST is a fork-level IPL used predominantly by drivers written before Version 4.0 of the VMS operating system. A driver fork process originating at an IPL between 8 and 11 would use IPL\$_QUEUEAST when it needed to synchronize access to the scheduler's database at IPL\$_SYNCH—for instance, to queue an AST. Prior to VMS Version 4.0, the only way that such a fork process could maintain proper synchronization was to first call a system routine that creates yet another fork process to be dispatched at IPL\$_QUEUEAST. Once the fork dispatcher dequeued the fork block and resumed execution of the driver, the driver fork process could then raise IPL to IPL\$_SYNCH and access the system database.

Because versions of the VMS operating system after Version 4.0 implement IPL\$_SYNCH as a fork IPL, a driver fork process can fork directly to IPL\$_SYNCH. In VMS Version 5.0, the fork dispatcher obtains the IPL 8 fork lock (IOLOCK8), dequeues the driver fork block, restores driver context, and resumes execution of the driver. To maintain synchronization in a VMS multiprocessing environment, the driver then must obtain the spin lock that corresponds to the data structure it is accessing.

3.1.3.3 IPL 7 (IPL\$_TIMERFORK)

The interval clock's interrupt service routine (EXE\$HWCLKINT), executing at IPL 22 or IPL 24 depending upon the VAX system, posts interrupts at IPL\$_TIMERFORK. A processor requests such an interrupt when the current process has exceeded its processor time quantum. The software timer interrupt service routine (EXE\$SWTIMINT) gets control when the IPL drops below IPL\$_TIMERFORK, services quantum end events by immediately raising IPL to IPL\$_SYNCH (obtaining the SCHED spin lock, if needed), and calls the appropriate scheduler routine.

The primary processor in a VMS multiprocessor system, when executing the interval clock's interrupt service routine, requests an IPL\$_TIMERFORK interrupt when the first entry in the timer queue (EXE\$GQ_1ST_TIME) is due. The software timer interrupt service routine contains special code that allows the primary processor to service the expiration of a timer queue element (TQE). The routine raises IPL to IPL\$_SYNCH, synchronizes access to the timer queue (except for the first TQE) by obtaining the TIMER spin lock, and secures the interval clock database (the system time at EXE\$GQ_SYSTIME and the expiration time of the first TQE at EXE\$GQ_1ST_TIME) by obtaining the HWCLK spin lock. Thus synchronized, it determines which TQEs have expired, dequeues them, and transfers control to the appropriate timeout handlers. Device timeouts are dispatched in this manner.

Synchronization of I/O Request Processing

3.1 Interrupt Priority Levels

3.1.3.4 IPL 8 (IPL\$_SYNCH)

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

3.1.3.5 IPL 11 (IPL\$_MAILBOX)

IPL\$_MAILBOX is the highest fork IPL. When a VMS or driver routine writes into a mailbox, the executing processor must be at IPL\$_MAILBOX holding the MAILBOX spin lock. Because other readers or writers to the mailbox must similarly pursue synchronization, these actions prevent other writers from modifying incomplete data in the mailbox and readers from reading invalid data.

3.1.3.6 IPL 14 (XDELTA Entry IPL)

For debugging purposes, you can halt a processor from the console terminal and request a software interrupt to invoke the XDELTA debugger. You accomplish this by depositing $0E_{16}$ in the processor's Software Interrupt Request Register (PR\$_SIRR). (The procedure for requesting a software interrupt to invoke XDELTA is described in Table 16-3.)

After you issue the console's CONTINUE command and return to program mode, the processor grants an interrupt at IPL 14. The processor must be executing below the requested IPL for the interrupt to take effect.

3.1.3.7 IPL 22 or IPL 24 (Interval Clock IPLs)

Every ten milliseconds, the interval clock interrupts at IPL 22 or 24, depending upon the VAX system. A system cell points to the IPL field (SPL\$_IPL) in the HWCLK spin lock, identifying the IPL at which the processor's interval clock interrupts.

The interval clock's interrupt service routine performs the functions described in Section 3.1.3.3. Note that the interval clock's interrupt service routine obtains the HWCLK spin lock to synchronize its operations on the system time quadword (EXE\$GQ\$_SYSTIME) and the quadword containing the due time of the first timer queue element (EXE\$GQ\$_1ST_TIME).

3.1.4 Modifying IPL in Driver Code

Kernel-mode code can modify the IPL of the local processor by either explicitly setting the processor's IPL to a specific value or by requesting a software interrupt at a specific level. Driver code can change the IPL at which it executes by invoking a VMS-supplied macro to request a change in IPL. Because the DEVICELOCK, FORKLOCK, and LOCK macros (and their counterparts) only raise (or lower) IPL in a VMS uniprocessing environment, but achieve full synchronization in a VMS multiprocessing system, DIGITAL recommends their use instead of the SETIPL, DSBINT, and ENBINT macros.

³ IPL\$_TIMER, IPL\$_SCHED, IPL\$_JIB, IPL\$_MMG, IPL\$_FILSYS, and IPL\$_IOLOCK8 are all synonyms for IPL\$_SYNCH (see Table 3-3).

Synchronization of I/O Request Processing

3.1 Interrupt Priority Levels

Table 3-2 lists the macros that set, store, or restore a processor's IPL. See Appendix B for a further explanation of the functions of these macros and a full description of their arguments.

Table 3-2 VMS Macros That Change a Processor's IPL

| Macro | Function |
|--|--|
| Raising IPL | |
| DEVICELock [<i>lockaddr</i>] [<i>,lockipl</i>] [<i>,savipl</i>] [<i>,condition</i>] [<i>,preserve=YES</i>] | Raises IPL on the local processor to the device IPL associated with the device lock's <i>lockaddr</i> , obtains the device lock, and saves the current IPL at <i>savipl</i> ¹ |
| DSBINT [<i>ipl=31</i>] [<i>,dst=-(SP)</i>] [<i>,environ=MULTIPROCESSOR</i>] | Raises IPL on the local processor to the specified <i>ipl</i> , saving the current IPL at <i>dst</i> ² |
| FORKLOCK [<i>lock</i>] [<i>,lockipl</i>] [<i>,savipl</i>] [<i>,preserve=YES</i>] [<i>,fip=NO</i>] | Raises IPL on the local processor to <i>lockipl</i> , obtains the fork lock, and saves the current IPL at <i>savipl</i> ¹ |
| LOCK <i>lockname</i> [<i>,lockipl</i>] [<i>,savipl</i>] [<i>,condition</i>] [<i>,preserve=YES</i>] | Raises IPL on the local processor to the <i>lockipl</i> , obtains the lock indicated by <i>lockname</i> , and saves the current IPL at <i>savipl</i> ¹ |
| SETIPL [<i>ipl=31</i>] [<i>,environ=MULTIPROCESSOR</i>] | Raises IPL on the local processor to the specified <i>ipl</i> ² |
| Lowering IPL | |
| DEVICEUNLOCK [<i>lockaddr</i>] [<i>,newipl</i>] [<i>,condition</i>] [<i>,preserve=YES</i>] | Releases or restores the device lock indicated by <i>lockaddr</i> , lowering the local processor's IPL to <i>newipl</i> , thus permitting interrupts to occur at or beneath the current IPL ¹ |
| ENBINT [<i>src=(SP+)</i>] | Lowers the local processor's IPL to <i>src</i> , thus permitting interrupts to occur at or beneath the current IPL ² |
| FORKUNLOCK [<i>lock</i>] [<i>,newipl</i>] [<i>,condition</i>] [<i>,preserve=YES</i>] | Releases or restores the fork lock indicated by <i>lock</i> , lowering the local processor's IPL to <i>newipl</i> , thus permitting interrupts to occur at or beneath the current IPL ¹ |
| UNLOCK <i>lockname</i> [<i>,newipl</i>] [<i>,condition</i>] [<i>,preserve=YES</i>] | Releases or restores the spin lock indicated by <i>lockname</i> and lowers IPL to <i>newipl</i> , thus permitting interrupts to occur at or beneath the current IPL ¹ |
| Miscellaneous Functions | |
| SAVIPL [<i>dst=-(SP)</i>] | Saves the local processor's IPL at the specified location |
| SOFTINT <i>ipl</i> | Requests a software interrupt on the local processor at the specified <i>ipl</i> |

¹When used in a uniprocessing environment, the DEVICELock, DEVICEUNLOCK, FORKLOCK, FORKUNLOCK, LOCK, and UNLOCK macros generate only the code that manipulates IPL.

²Use of the SETIPL, ENBINT, and DSBINT macros is not sufficient to guarantee systemwide synchronization of events and data in a VMS multiprocessing system. The DEVICELock, FORKLOCK, and LOCK macros have been designed to achieve appropriate synchronization in either a uniprocessing or multiprocessing environment.

Synchronization of I/O Request Processing

3.1 Interrupt Priority Levels

3.1.4.1 Raising IPL

To block certain activities on a local processor in a VAX system, it is sometimes useful to raise IPL explicitly. Driver code should not raise IPL for more than a few instructions, for doing so prevents the local processor from servicing interrupts at the current IPL and all lower IPLs.

In a uniprocessor environment, raising IPL provides sufficient systemwide synchronization to both block events and also protect data customarily accessed at a given IPL. Drivers typically raise a processor's IPL to check for a local processor power failure, send a message to a mailbox, or access device registers. For instance, a driver running exclusively in a VMS uniprocessor environment can set IPL to its device IPL (UCB\$B_DIPL) to access device registers. While the driver executes at device IPL, no other code thread can execute at the same device IPL and thereby read or write the same device registers. (See the discussion in Section 3.1.4.2 for a description of the rules for lowering IPL that enforce the synchronization.) VMS supplies the SETIPL and DSBINT macros to effect the change in IPL.

In a multiprocessing environment, as in a uniprocessing environment, a driver can block activities on the local processor by raising IPL. However, in a multiprocessing environment, simply raising IPL is not sufficient to protect shared data structures from other processors that may attempt to access them concurrently. To achieve synchronization in a VMS multiprocessing system, VMS associates a series of semaphores, called *spin locks*, with such shared databases. (See Section 3.2 for a further discussion of spin locks.)

A processor that must access a shared structure must first secure a corresponding spin lock. The acquisition of a spin lock often involves the raising of IPL to the IPL associated with the spin lock and the database it protects. Spin lock acquisition code can elevate IPL automatically if called from a code thread executing at an IPL lower than the synchronization IPL of the lock. A processor that has properly obtained a spin lock can thus proceed to access the associated database at the appropriate IPL. If necessary, it is free to further raise IPL, but should not lower IPL below the spin lock's allocation IPL without first releasing the spin lock.

For example, a driver running in a VMS multiprocessing system can set IPL to IPL\$_POWER to block the servicing of a power failure on the local processor. However, while executing at IPL\$_POWER (or at device IPL), the driver cannot safely access device registers unless it has first secured the spin lock associated with the device: that is, its device lock. Similarly, a driver's fork process, although it executes at a fork IPL with a corresponding fork lock held, may raise a processor's IPL by obtaining an additional spin lock. Sending a message to the OPCOM mailbox (obtaining the MAILBOX spin lock at IPL 11) and accessing device registers (obtaining the appropriate device lock at device IPL) are two such activities.

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

Synchronization of I/O Request Processing

3.1 Interrupt Priority Levels

3.1.4.2 Lowering IPL

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

One of the most fundamental coding rules in VMS is that *a code thread cannot explicitly lower IPL below the level at which its execution has been initiated.*

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

Driver code uses the following methods to lower IPL:

- Issuing a DEVICEUNLOCK, FORKUNLOCK, or UNLOCK macro (paired with an earlier invocation of a DEVICELOCK, FORKLOCK, or LOCK macro) or a ENBINT macro (paired with an earlier invocation of an DSBINT macro) to restore IPL to a previously saved value.
- Invoking the IOFORK (or FORK) macro to preserve its context in a fork block, insert the block in a fork queue, and request a software interrupt at the driver's fork IPL. See Section 3.3.3.1 for a complete discussion of forking.
- Issuing an REI instruction at the end of its interrupt service routine that dismisses the interrupt.

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

3.2 Spin Locks

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

A *spin lock* is a semaphore associated with a set of system structures, fields, or registers whose integrity is critical to the performance of a specific operating system task. The scheduler and the memory management subsystem thus have their own spin locks, as does each fork processing level and each device controller. Because a spin lock can be owned by only one processor in the system at a time, other processors attempting to acquire the same spin lock are prevented from reading from or writing into the database it protects. The structure of a spin lock is pictured in Figure A-15 and described in Table A-14.

Synchronization of I/O Request Processing

3.2 Spin Locks

There are two categories of spin lock:

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

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

Table 3-3 Static Spin Locks

| Lock Name | Lock Index | Synchronization IPL | Description |
|-----------|----------------|--------------------------------|---|
| QUEUEAST | SPL\$_QUEUEAST | 6 (IPL\$_QUEUEAST) | Fork lock for executing a fork process at IPL 6 |
| FILSYS | SPL\$_FILSYS | 8 (IPL\$_FILSYS) ¹ | Lock on file system structures |
| IOLOCK8 | SPL\$_IOLOCK8 | 8 (IPL\$_IOLOCK8) ¹ | Fork lock for executing a fork process at IPL 8 |
| PR_LK8 | SPL\$_PR_LK8 | 8 (IPL\$_IOLOCK8) ¹ | Primary CPU's private lock for IPL 8 |
| TIMER | SPL\$_TIMER | 8 (IPL\$_TIMER) ¹ | Lock for adding and deleting timer queue entries and searching the timer queue ² |
| JIB | SPL\$_JIB | 8 (IPL\$_JIB) ¹ | Lock for manipulating job nonpaged pool quotas as reflected by the fields JIB\$_BYTCNT and JIB\$_BYTLM in the job information block |
| MMG | SPL\$_MMG | 8 (IPL\$_MMG) ¹ | Lock on VMS memory management, PFN database, swapper, modified page writer, and creation of per-CPU database structures |
| SCHED | SPL\$_SCHED | 8 (IPL\$_SCHED) ¹ | Lock on process control blocks, scheduler database, and mutex acquisition and release structures |
| IOLOCK9 | SPL\$_IOLOCK9 | 9 (IPL\$_IOLOCK9) | Fork lock for executing a fork process at IPL 9 |
| PR_LK9 | SPL\$_PR_LK9 | 9 (IPL\$_IOLOCK9) | Primary CPU's private lock for IPL 9 |

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

²The HWCLK spin lock implicitly locks the timer queue element at the head of the timer queue by locking the quadword representing its due time (EXE\$GQ_1ST_TIME).

Synchronization of I/O Request Processing

3.2 Spin Locks

Table 3-3 (Cont.) Static Spin Locks

| Lock Name | Lock Index | Synchronization IPL | Description |
|------------|------------------|-----------------------|--|
| IOLOCK10 | SPL\$_IOLOCK10 | 10 (IPL\$_IOLOCK10) | Fork lock for executing a fork process at IPL 10 |
| PR_LK10 | SPL\$_PR_LK10 | 10 (IPL\$_IOLOCK10) | Primary CPU's private lock for IPL 10 |
| IOLOCK11 | SPL\$_IOLOCK11 | 11 (IPL\$_IOLOCK11) | Fork lock for executing a fork process at IPL 11 |
| PR_LK11 | SPL\$_PR_LK11 | 11 (IPL\$_IOLOCK11) | Primary CPU's private lock for IPL 11 |
| MAILBOX | SPL\$_MAILBOX | 11 (IPL\$_MAILBOX) | Lock for sending messages to mailboxes |
| POOL | SPL\$_POOL | 11 (IPL\$_POOL) | Lock on nonpaged pool database |
| PERFMON | SPL\$_PERFMON | 15 (IPL\$_PERFMON) | Lock for I/O performance monitoring |
| INVALIDATE | SPL\$_INVALIDATE | 19 (IPL\$_INVALIDATE) | Lock system space translation buffer (TB) invalidation |
| VIRTCONS | SPL\$_VIRTCONS | 20 (IPL\$_VIRTCONS) | Lock for ownership of the virtual console |
| HWCLK | SPL\$_HWCLK | 22 or 24 | Lock on interval clock database, including the quadword containing the due time of the first timer queue element and the quadword containing the system time |
| MEGA | SPL\$_MEGA | 31 (IPL\$_MEGA) | Lock for serializing access to fork and wait queue |
| MCHECK | SPL\$_MCHECK | 31 (IPL\$_MCHECK) | Lock for synchronizing certain machine error handling |
| EMB | SPL\$_EMB | 31 (IPL\$_EMB) | Lock for allocating and releasing error logging buffers |

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

VMS provides a set of macros, listed in Table 3-2 and described in full in Appendix B, that call the system's spin lock acquisition and releasing routines.

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

IPL

The processor must be executing at an IPL equal to or below the spin lock's synchronization IPL (SPL\$_IPL). In keeping with the rules discussed in Section 3.1.4.2, a processor should not lower the IPL of its thread of execution in the process of acquiring a spin lock. Thus, in acquiring a spin lock, a processor may or may not raise its IPL, depending upon whether it is executing already at the spin lock synchronization IPL. VMS supplies spin lock acquisition macros (DEVICELOCK, FORKLOCK, and LOCK) that, in calling appropriate VMS routines, raise IPL automatically in the course of obtaining the requested spin lock. Once it owns the spin lock, the processor

Synchronization of I/O Request Processing

3.2 Spin Locks

can raise its IPL above the IPL at which the spin lock was acquired, but it should not lower it below that level.

Rank

A processor can own multiple spin locks simultaneously, but must obtain these spin locks in increasing order of rank. (Table 3-3 lists the spin locks in order of rank.) In other words, a processor that owns one or more spin locks should not attempt to acquire a spin lock whose logical rank⁴ is less than a spin lock it already holds. It does not need to acquire all spin locks of intervening rank. This rule is meant to avoid potential deadlocks in the acquisition of system spin locks and fork locks, and does not pertain to device locks. The processor may release spin locks in any order, as long as any attempt to reacquire those spin locks acquires them in ascending order.

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

Ownership

The spin lock must not be owned by any other processor. If the spin lock is currently owned by another processor, a requesting processor *spin waits* for the lock to become available. That is, it executes in a loop, waiting for the processor that owns the spin lock to release it. If a spin lock is owned, its owner field (SPL\$L_OWN_CPU) contains an identifier that indicates which processor in the multiprocessor system owns the spin lock.

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

3.2.1 Fork Locks

In its simplest form, a fork lock is a static spin lock that synchronizes the right of a fork process to execute at a specified IPL in a VMS multiprocessing system. Fork locks exist for each of the fork IPLs from IPL 8 to 11. A driver indicates the fork lock under which it processes, and by implication its fork IPL, by specifying a fork lock index in its driver prologue table (using the DPT_STORE macro as described in Section 6.1).

Those code threads that must synchronize with another fork thread use the same fork lock. For instance, the fork processes of drivers whose devices share the resources of a common adapter *must* synchronize themselves by means of a common fork lock. These code threads fork not necessarily to lower IPL, but rather to wait for the availability of a common resource such as a controller data channel or map registers (see Section 3.4). The VMS routines that acquire and release these resources ensure that the fork lock is acquired and released as necessary.

⁴ The physical rank of a spin lock is the inverse of its logical rank. See the description of the SPL\$B_RANK field in Table A-14 for additional information.

Synchronization of I/O Request Processing

3.2 Spin Locks

Drivers rarely need to obtain a fork lock explicitly. VMS places the driver fork process into execution (originally by EXE\$INSIOQ and, by implication, by IOC\$REQCOM) at fork IPL holding the appropriate fork lock. In addition, the fork dispatcher obtains the fork lock associated with the driver fork process before it restores its context and resumes its execution. (Section 3.3.3 describes these concepts in greater detail.)

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

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

3.2.2 Device Locks

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

UCBs are protected by a device lock common to all units on the same adapter or common to the entire system, depending upon the type of device. A device lock is dynamically created by the System Generation Utility (SYSGEN) when it creates a channel request block (CRB). SYSGEN stores the address of the device lock in the CRB (CRB\$_DLCK) and later copies it to the unit control block (UCB\$_DLCK) as a UCB is created for each unit on the controller.

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

3.3 Device Driver Synchronization

This section describes how VMS and driver processing maintain synchronization during the processing of a general I/O request. It later focuses on the specific strategies drivers employ to synchronize at the device and fork levels.

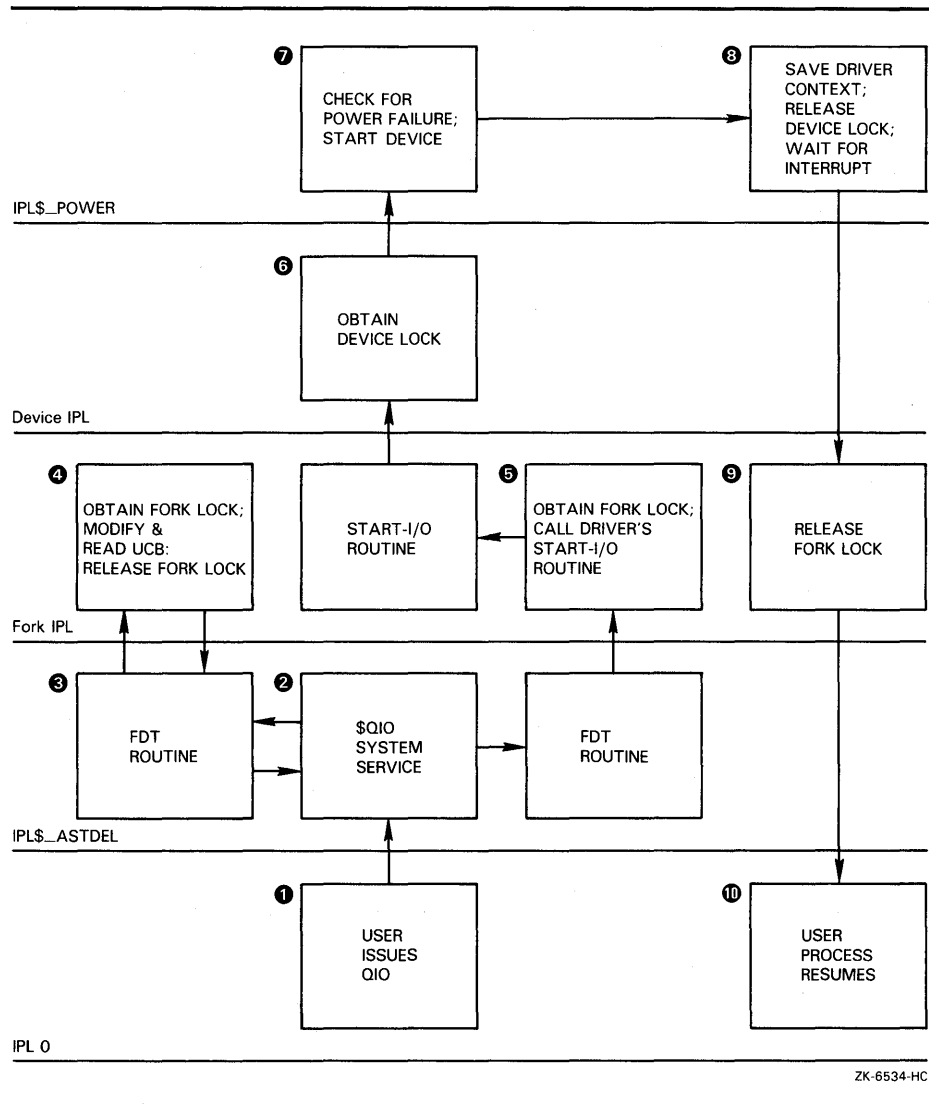
Synchronization of I/O Request Processing

3.3 Device Driver Synchronization

3.3.1 Overview of the Synchronization of an I/O Operation

Figure 3-1 diagrams the general flow of the processing of a single I/O request, as synchronization is achieved by raising and lowering IPL, and, in a multiprocessing environment, by also obtaining and releasing the necessary spin locks.

Figure 3-1 Synchronizing I/O Request Processing



ZK-6534-HC

Figure 3-1 illustrates the following events:

- 1 The user program, executing at IPL 0, issues a \$QIO system service call.
- 2 The \$QIO system service raises IPL to IPL\$_ASTDEL to prepare the I/O request according to the arguments included in the call.
- 3 The driver's FDT routines execute, mainly at IPL\$_ASTDEL.

Synchronization of I/O Request Processing

3.3 Device Driver Synchronization

Note that during IPL 0 processing and FDT routine activity at IPL\$_ASTDEL, the process requesting the I/O is susceptible to being rescheduled. In a multiprocessing environment, such an event could cause I/O processing to resume on a different processor from that on which it was started.

- ④ In certain rare circumstances, an FDT routine must read or modify the device's UCB. Because most fields in the UCB may be shared by fork processes running systemwide it is important that, if the FDT routine must use them, it issue the FORKLCK macro to obtain the appropriate fork lock and raise to fork IPL. (When finished, it relinquishes this synchronization by issuing the FORKUNLOCK macro.)
- ⑤ The continuation of VMS preprocessing of the I/O request—or the completion of a previous I/O request on the device unit—ensures that the driver's start-I/O routine is placed into execution at fork IPL and, in a multiprocessing system, holding the corresponding fork lock. The start-I/O routine accesses various UCB fields and contends for adapter resources synchronized systemwide by the fork lock.
- ⑥ Once it has further prepared the I/O request and obtained the required resources, it generally must access device registers. Device registers and those UCB fields that record their status are synchronized at device IPL. A processor in a VMS multiprocessing system must hold the appropriate device lock to access the device database.
- ⑦ While executing a critical code sequence, such as those instructions that start a device, the start-I/O routine raises the IPL of the local processor to IPL\$_POWER to check for a processor power failure. In a VMS multiprocessing environment, the executing processor retains the device lock during this sequence.
- ⑧ After it activates the device, the start-I/O routine calls a VMS routine that saves the driver's context in the UCB fork block, suspends driver processing, releases the device lock, if held, and restores IPL to a previous level.
- ⑨ VMS at this point returns control to the code that initiated the fork thread where, in a VMS multiprocessing system, the fork lock is released.
- ⑩ After VMS services interrupts at intervening IPLs, the user process resumes.

Figure 3-2 illustrates the synchronization involved in the completion of an I/O request from the point of the device interrupt to the delivery of ASTs to the user program. There is little linear flow involved in the completion of an I/O request. The servicing of interrupts, represented by jagged lines in the figure, the requesting of software interrupts, and the REI instruction contribute to the flow that completes an I/O request.

Synchronization of I/O Request Processing

3.3 Device Driver Synchronization

Figure 3-2 Synchronizing I/O Request Completion

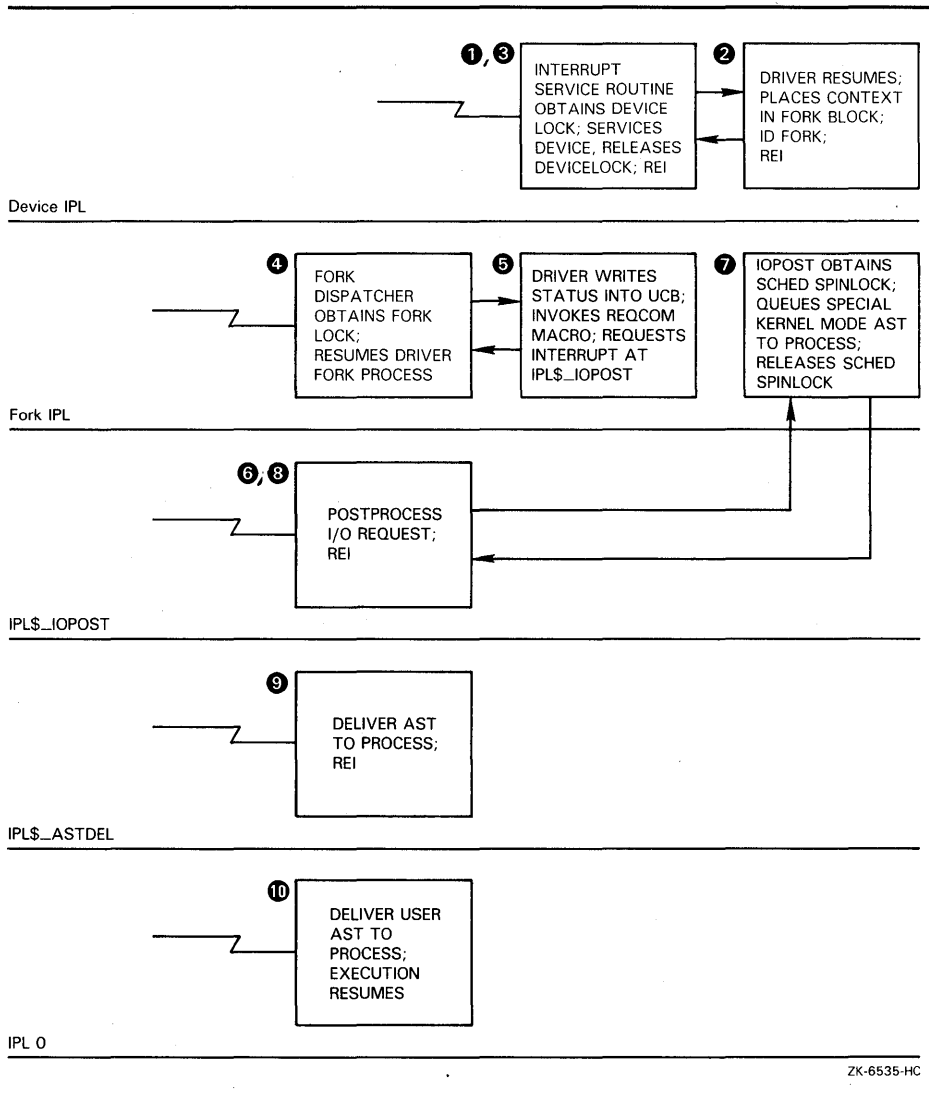


Figure 3-2 illustrates the following events:

- ① A device interrupt in the range of IPL 20 through IPL 23 triggers the execution of the driver's interrupt service routine. The interrupt service routine locates the device unit's UCB and, in a VMS multiprocessing system, immediately obtains the appropriate device lock. After it analyzes the interrupt and determines that it is expected, it reactivates the driver, still at device IPL and holding any acquired device lock.
- ② The driver briefly examines and/or saves the contents of the device's registers, but, in order to permit other device interrupts to be serviced and to allow other high priority system tasks to proceed, it lowers its own priority. The driver accomplishes this by requesting VMS to save some driver context in the UCB fork block and place it into one of the processor-specific fork queues at IPLs 8 through 11 serviced by the fork dispatcher. When it does so, VMS returns control to the driver's interrupt service routine.

Synchronization of I/O Request Processing

3.3 Device Driver Synchronization

- ③ The interrupt service routine releases any acquired device lock and issues an REI instruction to dismiss the device interrupt.
- ④ When IPL drops below the driver's fork IPL, the fork dispatcher restores the context of the driver and resumes its execution. In a VMS multiprocessing system, the fork dispatcher obtains the necessary fork lock prior to placing the driver into execution.
- ⑤ Still synchronized at fork level, the driver fork process analyzes the success of the I/O operation and writes status into R0 and R1. VMS then inserts the IRP into the local processor's I/O postprocessing queue, requests a software interrupt at IPL\$_IOPOST, and starts any I/O request that may be waiting for the device. Eventually, VMS returns to the fork dispatcher and, if no other fork processes are queued for that IPL, issues an REI instruction to dismiss the software interrupt.
- ⑥ When the processor's IPL falls below IPL\$_IOPOST, the I/O postprocessing routine removes the IRP from the I/O postprocessing queue, adjusts process quota usage, and deallocates system buffers for write functions.
- ⑦ When the routine finishes processing the IRP, it queues a special kernel-mode AST to the process that issued the original \$QIO request. To accomplish this, it obtains the SCHED spin lock (raising to IPL\$_SYNCH in the process) and calls another VMS routine that queues the AST to the process's PCB. It then releases the SCHED spin lock.
- ⑧ The I/O postprocessing routine continues execution at IPL\$_IOPOST until it has serviced all entries in the postprocessing queue. It then issues an REI to dismiss this software interrupt.
- ⑨ The special kernel-mode AST routine executes at IPL\$_ASTDEL. It completes the transfer of the results and status of the I/O request to the user process.
- ⑩ The special kernel-mode AST routine can queue a user-mode AST routine to the user process. When the user process has been rescheduled and its context reloaded, the user-mode AST routine executes at IPL 0.

3.3.2 Synchronizing the Device Database

A *device database* ordinarily consists of the device or adapter registers, plus some storage in the UCB (or in another data structure) that reflects the status of the device. Routines that access data in the device database must do so at device IPL (UCB\$_DIPL) in order to maintain synchronization. Generally, only three driver routines contend for access to the device database.

- Interrupt service routine
- Start-I/O routine when loading or reading device registers
- Timeout handling routine

In a VMS uniprocessing environment, the start-I/O routine raises its IPL to device IPL using the DSBINT macro. VMS calls the driver's timeout handling routine at device IPL. Because the interrupt dispatcher invokes it at device IPL, the driver's interrupt service routine does not need to acquire additional synchronization.

Synchronization of I/O Request Processing

3.3 Device Driver Synchronization

In a VMS multiprocessing environment, these routines must also hold the appropriate device lock (UCB\$L_DLCK). The device lock protecting the device database is a dynamic spin lock, created by SYSGEN when the device is configured and its channel request block (CRB) is created. The address of the device lock is first stored in CRB\$L_DLCK and is moved to UCB\$L_DLCK as corresponding UCBs are allocated for each unit on the controller. VMS calls the driver's timeout handling routine at device IPL with the device lock held. The start-I/O routine and the interrupt service routine must explicitly obtain such synchronization by invoking the DEVICELock macro.

The start-I/O routine and timeout handling routine are additionally synchronized at driver fork level. VMS raises IPL to fork level and obtains the corresponding fork lock before transferring control to them. This is not the case, however, with a driver's interrupt service routine. A device's interrupt service routine usually does not hold the fork lock. However, it may have preempted a thread holding the fork lock, or a fork thread may be running in parallel on another processor. Therefore, an interrupt service routine must not change any fields in the UCB that are protected by the fork lock. To access these fields, an interrupt service routine must fork, as described in Section 3.3.3.1.

3.3.3 Synchronizing at Driver Fork Level

A large part of driver code executes in the context of a *fork process*. As a fork process, driver code that must access data in its fork database does so at a single, specific fork IPL (from IPL 8 to IPL 11) and—in a VMS multiprocessing environment—holding a single, specific fork lock (see Section 3.2.1). The *fork database* consists of those fields in the unit control block (UCB) not explicitly synchronized at device level and such adapter or controller resources as map registers or data paths.

The system routine EXE\$INSIOQ initially creates a driver fork process as it attempts to deliver a preprocessed I/O request to the driver's start-I/O routine. If the device unit is busy (that is, a fork process is already active servicing a prior request for that device), EXE\$INSIOQ inserts the IRP into the UCB's pending-I/O queue. If the device unit is not busy, EXE\$INSIOQ calls IOC\$INITIATE to transfer control to the driver's start-I/O routine. The start-I/O routine begins to execute at fork IPL holding the associated fork lock, if necessary.

When the driver fork process later calls IOC\$REQCOM to complete processing of a prior I/O request, IOC\$REQCOM executes within the driver fork process, dequeues the next IRP on the pending-I/O queue, and begins processing it.

Like other processes, fork processes can be interrupted or suspended. The local processor interrupts a fork process when the processor receives a request for an interrupt at a higher priority level. To minimize the number of interruptions, fork processes sometimes execute at raised IPLs, and even raise their IPL to block all other interrupts, if necessary.

VMS stalls a driver's fork process when the process requests an unavailable resource such as a controller's data path (see Section 3.4). When suspended, a driver fork process, like other processes, preserves some context information. As VMS preserves some of the context of a normal process in its hardware PCB, so it preserves a driver fork process's context—however abbreviated—in a fork block. *Fork context* consists of the following:

Synchronization of I/O Request Processing

3.3 Device Driver Synchronization

- Two general purpose registers (R3 and R4)
- The program counter (PC)
- A fork block (usually the UCB), the address of which is in R5 at the time of the suspension

Minimal context helps ensure that, when a driver fork process is ready to be resumed, the resulting context-switching occurs swiftly.

3.3.3.1 Forking and the VMS Fork Dispatcher

Forking allows high IPL code to do the following:

- Continue executing a particular code thread at a lower IPL than the IPL at which the code thread was initiated
- Synchronize with other code executing at the lower IPL

Usually, a driver forks after servicing a device interrupt at an IPL from 20 through 23. By forking, the driver lowers the IPL at which it continues to process the device interrupt from device IPL to fork IPL (8 through 11). Forking not only allows the driver to process efficiently that part of interrupt request processing that is not time critical, but it allows the driver to synchronize its execution with other fork process code threads initiating I/O. For example, forking helps the driver synchronize its use of a device unit's UCB with other code threads interested in the structure. Moreover, the driver, by forking after completing the initial servicing of a device interrupt, allows other device interrupts to occur at that device IPL.

To fork, either the driver's interrupt service routine or the start-I/O routine, when resumed by the interrupt service routine, invokes the VMS macro IOFORK. The IOFORK macro saves fork process context in the UCB fork block, places the fork block in the local processor's fork queue for the specific fork IPL, and requests a software interrupt for that IPL. When that interrupt is ultimately serviced, driver fork processing resumes at the lower level.

There are other specialized instances in which a device driver may fork. As discussed in Section 11.1.5, the driver's unit initialization routine or controller initialization routine, while executing at IPL 31, may fork in order to permanently allocate controller resources, system nonpaged dynamic memory, or system page-table entries. To fork, these routines use the VMS macro FORK. The FORK macro allows a driver to fork, utilizing the fork block, the address of which is placed in R5. Because the channel request block (CRB) is available to these routines and contains a fork block, they invoke the VMS macro FORK with the address of the CRB in R5.

One interrupt service routine (EXE\$FORKDSPTH) handles all fork-process dispatching on each processor in a VAX system. When the processor grants an interrupt at fork IPL, the fork dispatcher saves R0 through R5 on the stack and processes the local 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 from the fork block, obtains ownership of the fork lock specified in the fork block (in a VMS multiprocessing system), and reactivates the suspended fork process.

When that fork process is completed, the dispatcher releases the fork lock and examines the fork queue. If an entry exists on the queue, the fork dispatcher removes it, restores the context of the fork process, secures the fork lock specified in the fork block, and reactivates the fork process. This sequence is repeated until the fork queue is empty. When the queue is empty, the fork

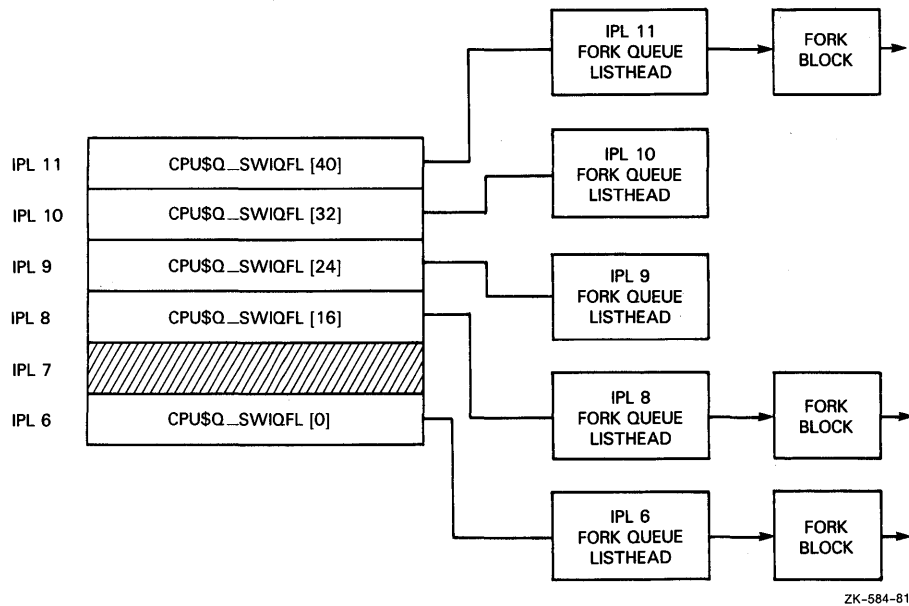
Synchronization of I/O Request Processing

3.3 Device Driver Synchronization

dispatcher restores R0 through R5 from the stack and dismisses the interrupt with an REI instruction.

Figure 3-3 illustrates the fork queue structure.

Figure 3-3 Processor-Specific Fork Queue Structure



3.3.3.2 Restrictions on Fork Processes

A driver fork process executes under the following constraints:

- It should not attempt to 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 VMS routine.
- It must execute at IPLs between the driver's fork IPL and IPL\$_POWER. It must not lower IPL below the driver's fork IPL except by creating a fork process to execute at a lower IPL.
- If executing in a VMS multiprocessing environment, it cannot attempt to obtain system spin locks with lower ranks than that of its fork lock.
- When it returns control to the fork dispatcher, the fork process must be at the same fork IPL and, if executing on a VMS multiprocessing system, own the appropriate fork lock.

Synchronization of I/O Request Processing

3.4 Resource Wait Queues

3.4 Resource Wait Queues

The processing of an I/O request often requires shared system resources such as memory and I/O adapter map registers. Drivers that depend on such resources synchronize access to these resources and their respective resource wait queues by executing at fork IPL and, in a VMS multiprocessing environment, obtaining ownership of the associated fork lock.

The \$QIO system service and fork processes call VMS routines to allocate and deallocate shared system resources. Because the resources are limited, I/O processing might be delayed until any such needed resources are released. Thus, synchronization of access to these resources can have a substantial impact on the processing of I/O requests.

For example, the \$QIO system service calls a VMS routine to allocate nonpaged system space for an IRP. If there is insufficient nonpaged pool, the routine calls another VMS routine to save the process context and change the process state to resource-wait mode (also called miscellaneous wait, or MWAIT). As a result of waiting, the process is a candidate to be swapped out of memory. When nonpaged pool becomes available, the scheduler reschedules the process.

During fork process execution at elevated IPLs, driver context is very small. At any point, the driver can obtain all details about an I/O request by referring to the I/O database (see Appendix A). The driver needs only the address of the device's UCB, which is the key to the rest of the database. Therefore, VMS routines that control driver resources, such as map registers, use fork blocks and resource-wait queues to save minimal driver context. Each entry in a queue is a fork block (or UCB) that contains R3, R4, and the continuation PC of 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 VMS routine that controls a particular resource stalls any driver that requests an unavailable resource, drivers are unaware of execution being suspended and subsequently reactivated. *Drivers must not leave anything on the stack, or in general purpose registers, other than R3, R4, and R5, when calling a routine that might suspend the driver's execution.*

3.4.1 Competing for a Controller's Data Channel

A controller's data channel is a VMS synchronization mechanism that guarantees that only one unit of a multiunit controller uses the controller at one time.

Devices that share a controller, such as disk units, own the controller's data channel only when a VMS routine assigns the channel to the unit's fork process. The device driver's start-I/O routine issues the REQCHAN macro to obtain the channel.

Synchronization of I/O Request Processing

3.4 Resource Wait Queues

In contrast, a device unit on a single-unit controller always owns the controller's data channel. The device driver's controller (or unit) initialization routine affirms this fact by moving the address of the device's UCB into `IDB$L_OWNER`. Generally, the driver's start-I/O routine does not request a single-unit controller.

In each case, the driver's start-I/O routine must take steps to synchronize its access to device registers with any access of these registers by the driver's interrupt service routine. The routine does so by issuing the `DEVICELock` macro (as described in Section 3.1.4). The `DEVICELock` macro raises IPL to device IPL and, in a VMS multiprocessing system, obtains the device lock associated with the controller.

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

- 1 The start-I/O routine requests the controller's data channel by invoking a VMS channel arbitration macro (`REQPCCHAN`).
- 2 The VMS routine tests the CRB mask field to determine whether the controller's data channel is available.
- 3 If the channel is available, the VMS routine allocates the channel to the fork process and returns the address of the device's CSR to the fork process.

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

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

The VMS channel-arbitration routines keep track of controller availability using a flag field in the CRB. The fork process must always request and release the controller's data channel by invoking these routines.

4 Overview of I/O Processing

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

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

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

When the transfer is completed, the device requests an interrupt that results in execution of the driver's interrupt service routine. This routine handles the interrupt and requests resumption of the driver fork process to perform device-dependent I/O postprocessing. The driver fork process finally transfers control to the system to perform device-independent I/O postprocessing. Figure 4-1 illustrates the sequence of events.

The \$QIO system service is dispatched by means of a corresponding system service vector in process P1 space. This vector contains a CHMK instruction that causes an exception that alters the process's access mode to kernel and dispatches to the service-specific procedure, EXE\$QIO. For the purposes of the discussion in this section, as well as the rest of the book, Figure 4-2 portrays the flow of an I/O request from its system service entry point to its servicing by VMS executive routines and driver code. Discussion of other entry points appears in Chapters 8, 9, and 10.

4.1 Preprocessing an I/O Request

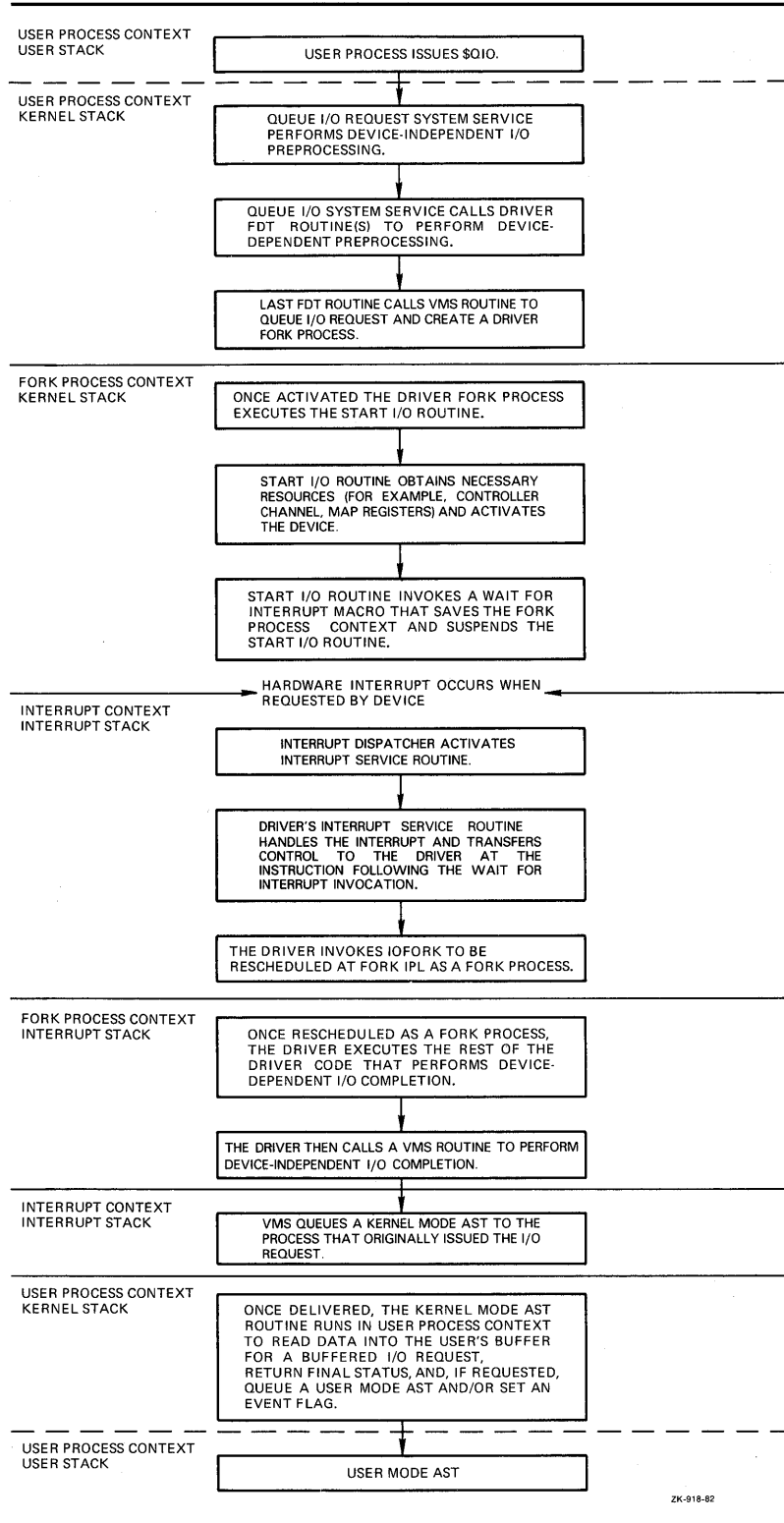
EXE\$QIO performs device-independent preprocessing of an I/O request and calls driver FDT routines to perform device-dependent preprocessing. To preprocess an I/O request, EXE\$QIO takes the following steps:

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

Overview of I/O Processing

4.1 Preprocessing an I/O Request

Figure 4-1 Sequence of Driver Execution

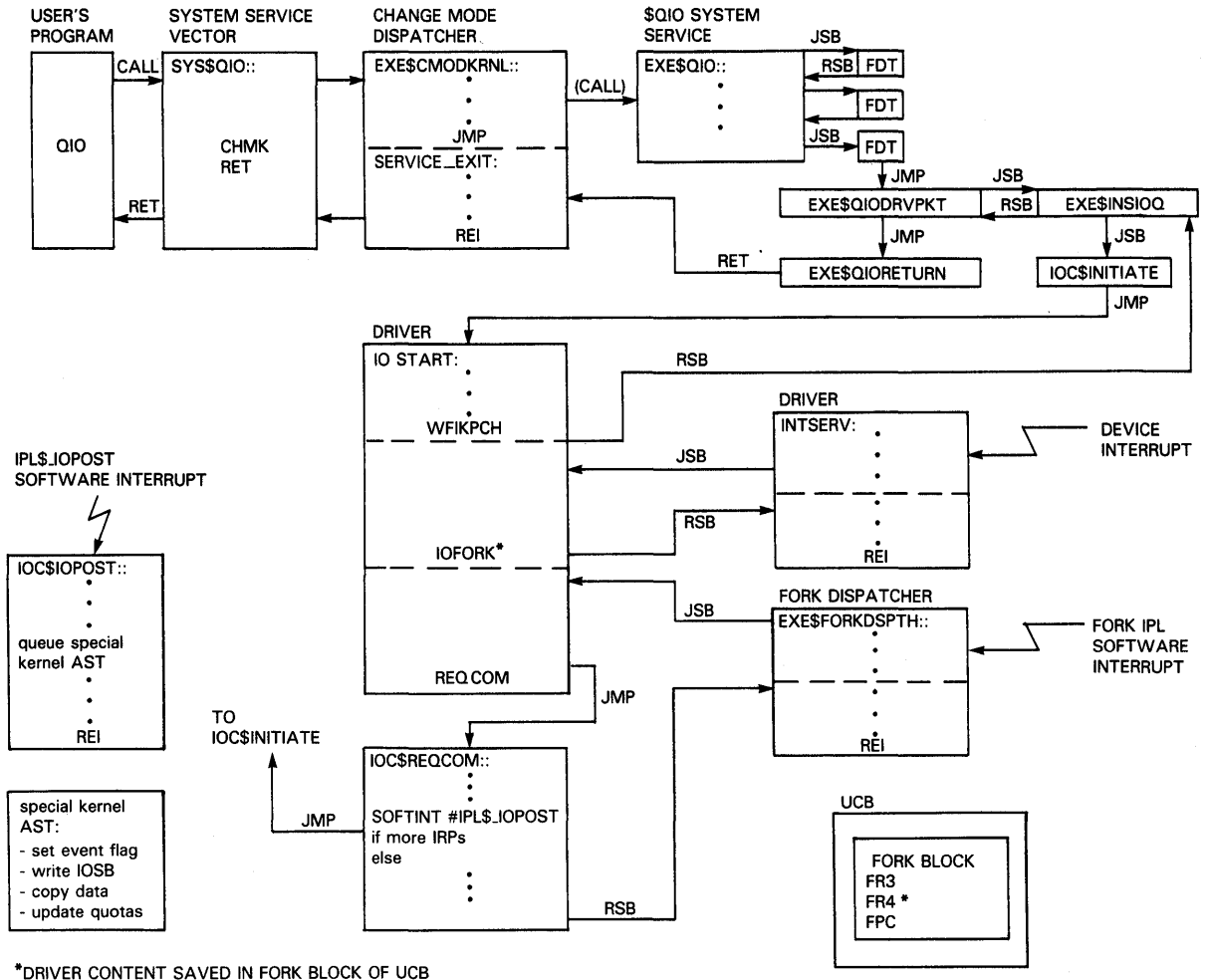


ZK-918-82

Overview of I/O Processing

4.1 Preprocessing an I/O Request

Figure 4-2 Detailed Sequence of VMS I/O Processing



ZK-4844-85

4.1.1 Process I/O Channel Assignment

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

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

Overview of I/O Processing

4.1 Preprocessing an I/O Request

When the program issues an I/O request, EXE\$QIO verifies that the channel number specified is associated with a device and locates the unit control block associated with the specified channel using the field CCB\$L_UCB.

Refer to Figure A-4 and Table A-3 for an illustration of the channel control block and a description of its contents.

4.1.2 Locating a Device Driver in the I/O Database

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

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

Because drivers run as fork processes and cannot use process address space to store additional context, drivers use the UCB for temporary data storage during I/O processing. (Section 6.1 describes how you can allocate additional UCB space for storing data or device-dependent driver context.)

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

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

Figure A-17 represents a UCB and Table A-16 describes its fields.

4.1.2.1 Channel Request Block

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

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

Controllers can be either multiunit or dedicated.

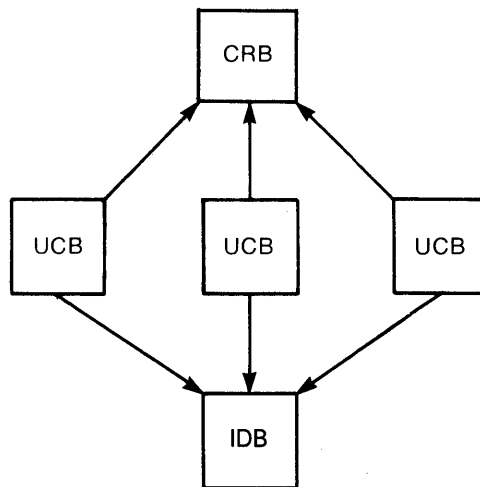
All UCBs describing device units attached to a single *multiunit controller* contain a pointer to a single CRB (UCB\$L_CRB). For these controllers, a VMS routine uses fields in the CRB (CRB\$L_WQFL, CRB\$B_MASK) and IDB (IDB\$L_OWNER) to arbitrate pending driver requests for the controller. When the system grants ownership of a multiunit controller data channel to a driver fork process, the fork process can initiate an I/O operation on a device

Overview of I/O Processing

4.1 Preprocessing an I/O Request

attached to that controller. Figure 4-3 illustrates the data structures required to describe three devices on a multiunit controller.

Figure 4-3 Data Structures for Three Devices on One Controller



ZK-920-82

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

See Figure A-6 and Table A-5 for an illustration of the CRB and a description of its contents.

4.1.2.2 Interrupt Dispatch Block

The CRB contains a pointer to an interrupt dispatch block (IDB) (CRB\$L_INTD+VEC\$L_IDB). In turn, the IDB (at IDB\$L_UCBLST) points to all UCBs that share the controller (see Figure 4-3).

The IDB contains the addresses of these three critical data structures:

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

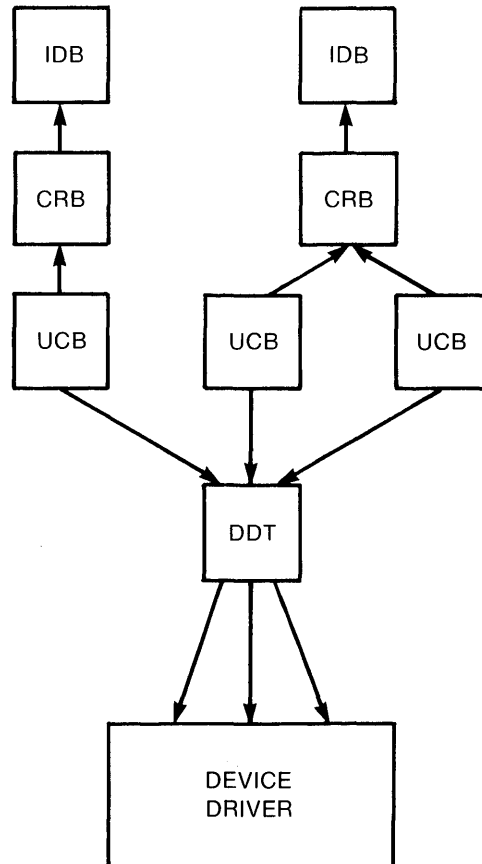
A detailed description of the fields in the IDB appears in Table A-10; Figure A-11 shows its structure.

Overview of I/O Processing

4.1 Preprocessing an I/O Request

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

Figure 4-4 I/O Database for Two Controllers



ZK-1765-84

4.1.2.3 Device Data Block

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

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

Table A-7 further describes the fields of the DDB. For a representation of its structure, see Figure A-8.

Overview of I/O Processing

4.1 Preprocessing an I/O Request

4.1.3 Validating the I/O Function

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

UCB → DDT → FDT

EXE\$QIO then uses data in the function decision table to analyze the I/O function. The procedure confirms that the function specified in the I/O request is a valid function for the device.

4.1.4 Checking Process I/O Request Quotas

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

If the flag is clear, EXE\$QIO aborts the I/O request. However, if the flag is set, it places the process in a wait state until previously issued I/O requests complete and the number of requests drops below quota. When this occurs, process execution resumes, at which time EXE\$QIO charges process quotas as appropriate for the requested operation.

4.1.5 Validating the I/O Status Block

If the I/O request specifies a quadword I/O status block to receive final I/O status information, EXE\$QIO determines whether the process issuing the request has write access to the status block locations specified. If the process has write access, EXE\$QIO fills the quadword with zeros. If the process does not have write access, the procedure terminates the request with an error status.

4.1.6 Allocating and Setting Up an I/O Request Packet

If validation of the I/O request succeeds to this point, EXE\$QIO allocates a block of nonpaged pool to contain an IRP.

Before EXE\$QIO allocates an IRP, it raises the IPL of the processor to IPL\$_ASTDEL to block any other asynchronous activity in the process. The new IPL prevents possible deletion of the process; process deletion would result in the operating system's losing track of the pool allocated for the IRP.

EXE\$QIO attempts to allocate an IRP from a *lookaside list* containing preallocated IRPs. If no preallocated packets exist, the procedure calls a VMS routine that allocates an IRP from general nonpaged pool. This allocating routine synchronizes with the rest of the system so that it can allocate the memory needed.

Overview of I/O Processing

4.1 Preprocessing an I/O Request

EXE\$QIO resumes I/O preprocessing by writing a description of the I/O request into the fields of the IRP as follows. Note that this data encompasses the *device-independent* information associated with the request. It is up to the device driver's FDT routines or VMS common FDT routines to fill in the *device-dependent* portions of the IRP, as described in Section 4.1.7 and Chapter 7.

| Data | Field(s) |
|---|--|
| Size in bytes of the IRP | IRP\$W_SIZE |
| Identification of the block as an IRP | IRP\$B_TYPE |
| Access mode of the process at the time of the request | IRP\$B_RMOD |
| Process ID of the requesting process | IRP\$L_PID |
| Address of an AST routine (if specified in the request) and its parameter ¹ | IRP\$L_AST, IRP\$L_ASTPRM |
| For file-structured devices, address of a window control block (WCB) that describes the physical location of part of the file | IRP\$L_WIND |
| Address of the target device's UCB | IRP\$L_UCB |
| I/O function code ² | IRP\$W_FUNC |
| Number of event flag to set when processing of the I/O request is complete | IRP\$B_EFN |
| Base software priority of the requesting process | IRP\$B_PRI |
| Address of an I/O status block (if specified in the request) | IRP\$L_IOSB |
| Process I/O channel index number | IRP\$W_CHAN |
| A flag indicating whether the I/O function is for buffered or direct I/O | IRP\$V_BUFIO in IRP\$W_STS |
| A flag indicating whether the I/O request is an input request | IRP\$V_FUNC in IRP\$W_STS |
| A flag indicating whether the I/O function is a physical-I/O function | IRP\$V_PHYSIO in IRP\$W_STS |
| Address of a diagnostic buffer (if specified in the request) ³ and a flag indicating that the buffer is present | IRP\$L_DIAGBUF, IRP\$V_DIAGBUF in IRP\$W_STS |
| Address of process's access rights block | IRP\$L_ARB |
| I/O transaction sequence number | IRP\$L_SEQNUM |

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

²For nonfile devices (DEV\$V_FOD clear in UCB\$L_DEVCHAR), EXE\$QIO reduces read- and write-virtual-block functions to their equivalent read- and write-logical-block functions before storing a code.

³The size of the diagnostic buffer is specified in the driver dispatch table of the driver servicing the device unit to which the request is made. See Section 6.2 for more information.

Overview of I/O Processing

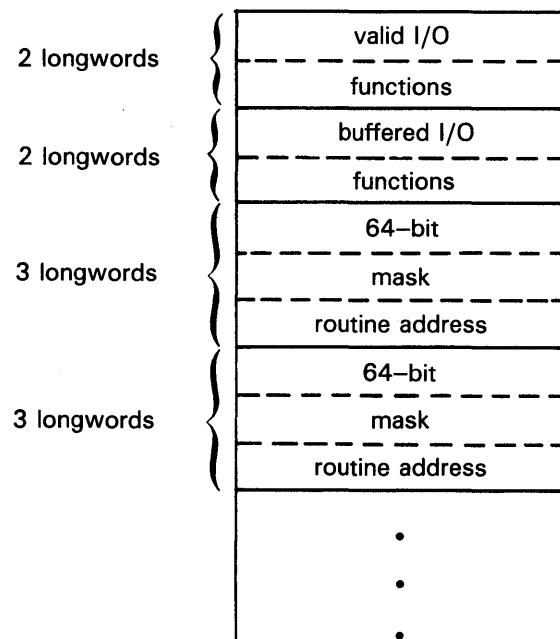
4.1 Preprocessing an I/O Request

Figure A-12 illustrates the format of an IRP; Table A-11 describes each of its fields.

4.1.7 FDT Processing

The driver's function decision table controls the device-dependent preprocessing of an I/O request. Figure 4-5 illustrates the layout of a function decision table.

Figure 4-5 Layout of a Function Decision Table



ZK-921-82

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

- A 6-bit I/O function code (bits 0 through 5) that permits you to define 64 unique I/O function codes for every device type. Table 6-1 lists the function codes defined by VMS. Section 6.3.2 describes how you can define device-specific function codes.
- A 10-bit I/O function modifier (bits 6 through 15). In subsequent processing of the I/O request, the driver's start-I/O routine uses both I/O function code and I/O function modifier, as stored in IRP\$W_FUNC, to create a device-specific function code to use in device activation.

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

Overview of I/O Processing

4.1 Preprocessing an I/O Request

EXE\$QIO uses the value of the low-order six bits of the I/O function code to determine which bit to check in each of these bit masks. For example, if the function code has a value of 22, the procedure checks the twenty-third bit (bit 22) of each bit mask. Thus, EXE\$QIO determines whether the I/O function code is valid for the device and is able to charge against the appropriate quota of the requesting process for a direct- or buffered-I/O operation.¹

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

Some FDT routines are present in the operating system because they provide common services for many devices. Section 7.5 describes these routines. Other routines are included in the device driver because they perform device-dependent services.

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

- 1 If the bit corresponding to the function code is set in the action routine bit mask, EXE\$QIO calls the FDT routine whose address appears in the following longword.
 - If this I/O function requires additional preprocessing after this particular FDT routine completes its activity, the FDT routine returns control to EXE\$QIO with an RSB instruction. When EXE\$QIO regains control, it advances to the next action routine bit mask and repeats step 1.
 - If this FDT routine completes all necessary preprocessing for this particular I/O function, then it transfers control to a VMS routine that queues the IRP or completes the request.
- 2 If the bit corresponding to the function code is not set, EXE\$QIO advances to the next action routine bit mask in the table and repeats step 1.

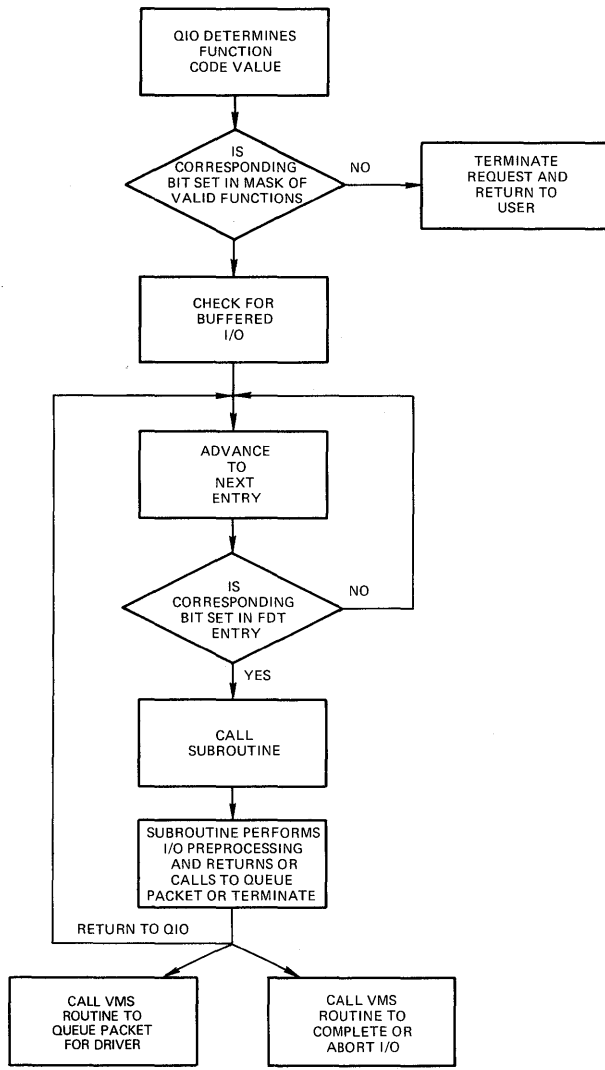
Note: A single function decision table can specify that EXE\$QIO call more than one FDT routine to perform the many and varied steps in the preprocessing of a single I/O function. However, it is the responsibility of the FDT routine that ultimately completes the preprocessing to end the scan (by EXE\$QIO) of the function decision table. An FDT routine accomplishes this by transferring control to either a VMS routine that queues the I/O request for the driver's start-I/O routine or one that completes or aborts the request (see Figure 4-2). In other words, for each valid I/O function code for a device, an FDT entry must contain the address of a routine that ends I/O preprocessing.

¹ For physical- and logical-I/O operations, EXE\$QIO also verifies that the process making the I/O request has suitable privileges.

Overview of I/O Processing

4.1 Preprocessing an I/O Request

Figure 4-6 FDT Routines and I/O Preprocessing



ZK-922-82

Overview of I/O Processing

4.1 Preprocessing an I/O Request

FDT routines execute in the context of the process that requested the I/O operation. Thus, FDT routines can access 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.

4.2 Handling Device Activity

When I/O preprocessing is complete, the last-called FDT routine generally jumps (with a JMP instruction) to a routine called EXE\$QIODRVPKT.² EXE\$QIODRVPKT, in turn, transfers control (using a JSB instruction) to EXE\$INSIOQ, the VMS routine that queues IRPs and arbitrates device activity. (See Figure 4-2 for a representation of the flow of I/O request processing at this juncture.)

4.2.1 Creating a Driver Fork Process to Start I/O

EXE\$INSIOQ creates only one driver fork process at a time for each device unit on the system. As a result, only one IRP for each device unit is serviced at one time. EXE\$INSIOQ determines whether a driver fork process exists for the target device, as follows:

- If the device is idle, no driver fork process exists for the device; in this case, EXE\$INSIOQ immediately calls IOC\$INITIATE to create and transfer control to a driver fork process to execute the driver's start-I/O routine.
- If the device is busy, a driver fork process already exists for the device, servicing some other I/O request. In this case, EXE\$INSIOQ calls EXE\$INSERTIRP to insert the IRP into a queue of IRPs waiting for the device unit. The routine queues the IRP according to the base priority of the caller. Within each priority, IRPs are in first-in/first-out order. The completion of the current I/O request triggers the servicing of the I/O request that is first in the queue, according to the procedure described in Section 10.1.2.3.

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

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

- 1 Raising IPL to driver fork IPL (and obtaining the associated fork lock in a VMS multiprocessing environment)
- 2 Loading the address of the IRP into R3
- 3 Loading the address of the device's UCB into R5

² The rules for exiting from FDT preprocessing, including descriptions of EXE\$QIODRVPKT and other FDT exit routines, appear in Sections 7.2.1 and 7.2.

Overview of I/O Processing

4.2 Handling Device Activity

- 4 Transferring control (with a JMP instruction) to the entry point of the device driver's start-I/O routine

The newly activated driver fork process executes under the constraints listed in Section 3.3.3.2. It 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 3.4.
- Device activity requires the fork process to wait for a device interrupt.

4.2.2 Activating a Device and Waiting for an Interrupt

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

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

This routine (IOC\$WFIKPCH or IOC\$WFIRLCH) expects to find, among the items it inherits on the stack, the driver's fork IPL, as placed there by the start-I/O routine in step 7. As it suspends the driver, IOC\$WFIKPCH or IOC\$WFIRLCH saves the driver's context in the UCB's fork block. This context consists of the following information:

- The contents of R3 and R4 (UCB\$_FR3, UCB\$_FR4)
- The implicit contents of R5 as the address of the UCB
- A driver return address (UCB\$_FPC)

Overview of I/O Processing

4.2 Handling Device Activity

- The relative offset to a device timeout handler (calculated from UCB\$_FPC and the value specified in the invocation of the WFIKPCH or WFIRLCH macro)
- The time at which the device will time out (UCB\$_DUETIM)

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

Having removed the driver's start-I/O routine's return address from the stack and stored it in UCB\$_FPC, IOC\$WFIKPCH (or IOC\$WFIRLCH) issues a DEVICEUNLOCK macro that restores IPL to fork IPL from the stack. It then exits with an RSB instruction. Thus, IOC\$WFIKPCH (or IOC\$WFIRLCH) effectively passes control to the caller of its caller. In this case, the caller of the driver start-I/O routine is EXE\$INSIOQ. The flow back from EXE\$INSIOQ to a user process that asynchronously requested the I/O operation is shown in Figure 4-2.

You can find additional information on the context of a start-I/O routine in Chapter 8.

4.2.3 Handling a Device Interrupt

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

- 1 Retrieves the address of the UCB that owns the controller from IDB\$_OWNER
- 2 Issues the DEVICELOCK macro to obtain the device lock associated with operations at device IPL in a VMS multiprocessing environment
- 3 For multiunit device controllers, determines which device unit generated the interrupt
- 4 Examines the UCB for the device to confirm that the driver fork process expects the interrupt
- 5 Saves device registers
- 6 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.

4.2.4 Switching from Interrupt to Fork Process Context

To lower its priority, the driver calls a VMS fork process queuing routine (by means of the IOFORK macro) that performs the following actions:

- 1 Disables the timeout that was specified in the wait-for-interrupt routine

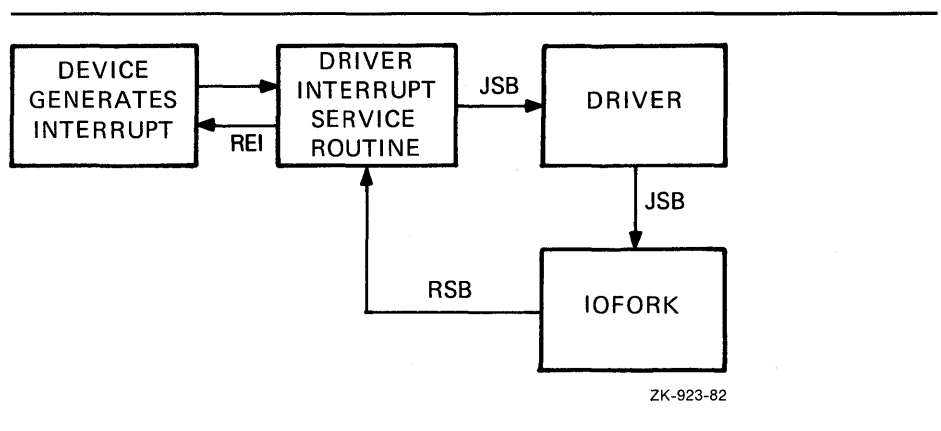
Overview of I/O Processing

4.2 Handling Device Activity

- 2 Saves R3 and R4 (UCB\$L_FR3, UCB\$L_FR4)
- 3 Saves the address of the instruction following the IOFORK request in the UCB fork block (UCB\$L_FPC)
- 4 Places the address of the UCB fork block from R5 in a processor-specific fork queue for the driver's fork level
- 5 Returns to the driver's interrupt service routine

The interrupt service routine then cleans up the stack, issues the DEVICEUNLOCK macro to release the device lock, restores registers, and dismisses the interrupt. Figure 4-7 illustrates the flow of control in a driver that creates a fork process after a device interrupt.

Figure 4-7 Creating a Fork Process After an Interrupt



4.2.5 Activating a Fork Process from a Fork Queue

When no higher priority interrupts are pending, the local processor transfers control to the fork dispatcher. When the processor grants an interrupt at a fork IPL, the fork dispatcher processes the local fork queue that corresponds to the IPL of the interrupt. To do so, the dispatcher performs these actions:

- 1 Removes a fork block from the fork queue
- 2 Restores fork context
- 3 Obtains the fork lock specified in the fork block
- 4 Transfers control back to the fork process

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

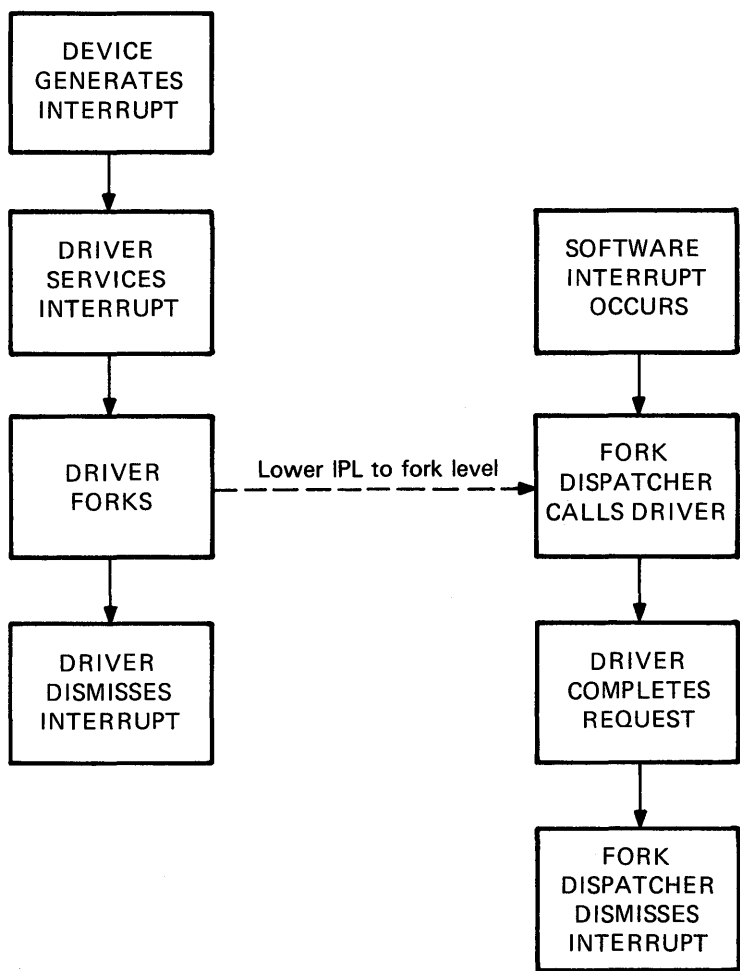
Overview of I/O Processing

4.2 Handling Device Activity

When a given fork process completes execution, the fork dispatcher releases the fork lock and removes the next entry, if any, from the local fork queue. This fork dispatcher repeats the sequence described previously until the fork queue is empty. After servicing the last entry in the queue, the fork dispatcher releases the fork lock, restores R0 through R5 from the stack, and dismisses the interrupt with an REI instruction.

Figure 4-8 illustrates the reactivation of a driver fork process.

Figure 4-8 Reactivation of a Driver Fork Process



ZK-924-82

Overview of I/O Processing

4.3 Completing an I/O Request

4.3 Completing an I/O Request

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

- 1 Releases shared driver resources, such as map registers, UNIBUS adapter buffered data path, and controller ownership
- 2 Returns status to the VMS I/O completion routine

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

- 1 Writes return status from the driver into the IRP
- 2 Inserts the finished IRP in the local processor's I/O-postprocessing queue and requests an interrupt from the processor at IPL\$_IOPOST
- 3 Creates a new fork process for the next IRP in the device's pending-I/O queue
- 4 Activates the new driver fork process

4.3.1 I/O Postprocessing

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

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

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

Overview of I/O Processing

4.3 Completing an I/O Request

The queuing of a special kernel-mode AST routine allows I/O postprocessing to execute in the context of the user process but in a privileged access mode. Process context is needed to return the results of the I/O operation to the process's address space. The special kernel-mode AST routine sets any common event flag associated with the I/O request and 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 an I/O completion AST routine, the special kernel-mode AST routine queues the I/O completion AST for the process. When VMS delivers the I/O completion AST, the system AST delivery routine deallocates the IRP. The first part of an IRP is the AST control block for user requested ASTs.

Part II Writing a Device Driver

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

- Static tables
- Routines that use the device driver's function decision table (FDT)
- Routines that start an I/O operation on the device and complete the I/O operation
- Routines that handle interrupts
- Routines that initialize devices and controllers
- Routines that cancel an I/O operation
- Routines that log errors

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

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

5

Template for a Device Driver

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

Figure 5-1 illustrates the organization of a device driver. The first item in a device driver is the driver prologue table and the second is usually the driver dispatch table. The order of the remaining driver components varies from driver to driver.

The last statement in every driver, except for the .END assembly directive, must be a label marking the end of the driver. The address of this label is stored in the driver prologue table. The driver-loading procedure uses this address to calculate the size of the driver. Chapter 15 describes the driver-loading procedure.

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

5.1 Coding Conventions

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

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

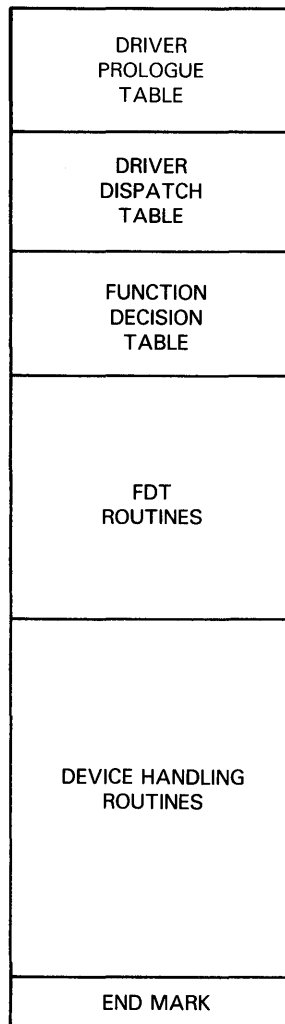
The rules of position-independent and reentrant code are as follows:

- Instructions can branch only to relative addresses within the driver and to global addresses listed in the VMS symbol table (SYS\$SYSTEM:SYS.STB).
- Static tables can list only global addresses and relative addresses within the driver.
- The driver cannot store temporary data in local driver tables for dynamic driver context. All dynamic temporary storage must be contained within the unit control block corresponding to an I/O request or the current I/O request block.
- The driver must refer to the I/O database by loading the address of a data structure into a general register and using displacement addressing to the fields of the data structure.

Template for a Device Driver

5.1 Coding Conventions

Figure 5-1 Driver Organization



ZK-925-82

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

- FDT routines can use R0 through R2 and R9 through R11 as available registers. The routines can use other registers by saving the registers before use and restoring them before exiting from the FDT routine.
- All other driver routines can use R0 through R5 as available registers. The routines can use other registers, if necessary, by saving and restoring them; but using other registers in this way is discouraged.
- All driver routines can use the stack for temporary storage only if the routines restore the stack to its previous state before calling any VMS routines, forking, or executing RSB instructions.

Template for a Device Driver

5.1 Coding Conventions

Because certain VAX processors and VMS cooperate to support the emulation of specific sets of VAX instructions, a device driver writer should exercise some caution. Because the software emulation for floating-point instructions may at some time be placed in pageable code, drivers should *never* use floating-point instructions. VMS only guarantees the emulation for character string instructions to be nonpaged.

5.2 Restrictions on the Use of Device-Register I/O Space

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

- Drivers should always store the address of a device control register in a general register and then gain access to the device register indirectly through the general register. The following example 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 UCBS$L_CRE(R5),R4 ; Get address of CRB  
MOVL @CRB$L_INTD+VEC$L_IDB(R4),R4 ; Get the address of  
; the device's CSR  
  
.  
.  
TSTW LP_CSR(R4) ; Is printer on line?
```

- Floating-point, field, queue, quadword, and octaword operands are not allowed in I/O address space, nor can an instruction obtain the position, size, length, or base of an operand from I/O space. For example, a driver cannot use a bit field instruction to test a bit in a device register.
- Drivers cannot use string-handling instructions when referring to I/O space.
- Drivers can use only those instructions that modify or write to a maximum of one destination. The destination must be the last operand.
- Registers of devices connected to the backplane interconnect (for example, UNIBUS adapter device registers and MASSBUS device registers) are longwords. Registers of devices connected to the UNIBUS or Q22 bus are words. Instructions that refer to UNIBUS adapter registers must use longword context. All driver instructions that affect UNIBUS or Q22 bus device registers must use word context (for example, BISW, MOVW, and ADDW3) unless the register is byte addressable.

Template for a Device Driver

5.2 Restrictions on the Use of Device-Register I/O Space

- An instruction that refers to I/O space must not generate an exception or be interruptable. If the instruction is allowed to restart, it will reread the device register, which can cause undesirable device side effects or data loss.
- On any given VAX processor, a device driver cannot anticipate the completion of an instruction that writes to I/O space before subsequent instructions execute. The processor can continue to execute without waiting for the data to reach its intended destination.

Among the consequences of this behavior are the following:

- If a driver initiates device actions that result in an interrupt from the device, the amount of time before that interrupt actually occurs is unpredictable.
- If a driver disables interrupts from a device, the time before that device can no longer generate an interrupt is unpredictable.
- An I/O bus error will not be reported synchronously with the instruction causing the error.

As a result, a driver's interrupt service routine always should be prepared to service unexpected or spurious interrupts. See Section 9.3 for additional discussion of the servicing of unexpected interrupts.

- To access I/O space, use only the following instructions. These instructions cannot be interrupted unless they use autoincrement-deferred addressing mode or any of the displacement-deferred modes when specifying an operand.

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

Template for a Device Driver

5.3 Implementing Conditional Code in a Driver

5.3 Implementing Conditional Code in a Driver

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

The ADPDISP macro (defined in SYS\$LIBRARY:LIB.MLB) provides a means by which a device driver can be designed to drive a similar device in a variety of VAX configurations. The ADPDISP macro allows the driver to determine at run time the existence of a certain I/O bus or adapter characteristic, and transfer control to code designed to execute given this hardware trait.

A driver can use ADPDISP to transfer control to specific code given any of the following characteristics:

- Adapter type
- Number of adapter address bits (18 or 22)
- Map registers supported
- Autopurging data paths supported
- Buffered data paths supported
- Direct-vector interrupt dispatching supported
- Odd-aligned transfers on buffered data path supported
- Odd-aligned transfers on direct data path supported
- Alternate set of map registers (496 to 8191) available
- Q22 bus device

Use ADPDISP when it is necessary to conditionally execute pieces of code, for instance, the allocation and loading of map registers for devices for which map registers are available or the allocation of a physically contiguous buffer for a DMA transfer on the MicroVAX I (which cannot map such a transfer). VMS supplies a similar macro, CPUDISP, which causes a run-time transfer of control to a specified destination depending on the CPU type of the executing processor. For those processors not uniquely identified by CPU type, CPUDISP also provides the means to dispatch on a particular CPU subtype.

Because a device driver cannot make assumptions about the I/O architecture of any given VAX system, DIGITAL recommends that most instances of the CPUDISP macro be replaced by an appropriate usage of the ADPDISP macro.

Appendixes E and F contain examples of drivers that use the ADPDISP macro to provide conditional code in a driver. See also the description of the ADPDISP macro in Appendix B.

Template for a Device Driver

5.4 Driver Template

```

;          Remove BLBC instruction from CANCEL routine.
;
;          VO2-001 ROW0067 R. Programmer  11-Feb-1981   13:10
;          Add description of reason argument to CANCEL routine.
;          Correct references to channel index number.
;
;--
.SBTTL  External and local symbol definitions
;
; External symbols
;
;          $CANDEF          ; Cancel reason codes
;          $CRBDEF          ; Channel request block
;          $DCDEF           ; Device classes and types
;          $DDBDEF          ; Device data block
;          $DEVDEF          ; Device characteristics
;          $IDBDEF          ; Interrupt 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

```

Template for a Device Driver

5.4 Driver Template

```
;  
; Bit positions for device-dependent status field in UCB  
;  
$VFIELD 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  
;  
_VFIELD TD_STS,0,<- ; Control/status register  
        <GO,,M>,- ; Start device  
        <BIT1,,M>,- ; Bit one  
        <BIT2,,M>,- ; Bit two  
        <BIT3,,M>,- ; Bit three  
        <XBA,2,M>,- ; Extended address bits  
        <INTEN,,M>,- ; Enable interrupts  
        <READY,,M>,- ; Device ready for command  
        <BIT8,,M>,- ; Bit eight  
        <BIT9,,M>,- ; Bit nine  
        <BIT10,,M>,- ; Bit ten  
        <BIT11,,M>,- ; Bit eleven  
        <,1>,- ; Disregarded bit  
        <ATTN,,M>,- ; Attention bit  
        <NEX,,M>,- ; Nonexistent memory flag  
        <ERROR,,M>,- ; Error or external interrupt  
        >  
$DEF TD_WRDCNT ; Word count  
        .BLKW 1  
$DEF TD_BUFADR ; Buffer address  
        .BLKW 1  
$DEF TD_DATBUF ; Data buffer  
        .BLKW 1  
$DEFEND TD ; End of device register  
           ; definitions  
$SBTTL Standard tables
```


Template for a Device Driver

5.4 Driver Template

```

;
; 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_FLCK,B,SPL$C_IOLOCK8  ; Device FORK LOCK
DPT_STORE  UCB,UCB$B_DIPL,B,22             ; Device interrupt IPL
DPT_STORE  UCB,UCB$L_DEVCHAR,L,<-         ; Device characteristics
          DEV$I_IDV!-                       ; input device
          DEV$I_ODV>                         ; output device
DPT_STORE  UCB,UCB$B_DEVCLASS,B,DC$SCOM    ; Sample device class
DPT_STORE  UCB,UCB$W_DEVBUSIZ,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$l_INTD+VEC$l_ISR,D,-    ; Address of interrupt
          TD_INTERRUPT                       ; service routine
DPT_STORE  CRB,-                            ; Address of controller
          CRB$l_INTD+VEC$l_INITIAL,-        ; initialization routine
          D,TD_CONTROL_INIT

DPT_STORE  CRB,-                            ; Address of device
          CRB$l_INTD+VEC$l_UNITINIT,-      ; unit initialization
          D,TD_UNIT_INIT                    ; routine

DPT_STORE  END                             ; End of initialization
          ; tables

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

```

Template for a Device Driver

5.4 Driver Template

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

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

Template for a Device Driver

5.4 Driver Template

```

; ++
; TD_UNIT_INIT, Readies unit for I/O operations
;
; Functional description:
;
;     The operating system calls this routine after calling the
;     controller initialization routine:
;
;         at system startup
;         during driver loading
;         during recovery from a power failure
;
; Inputs:
;
;     R4     - address of the CSR (controller status register)
;     R5     - address of the UCB (unit control block)
;
; Outputs:
;
;     The routine must preserve all registers except R0-R3.
;
; --
TD_UNIT_INIT:                                ; Initialize unit
    BISW    #UCB$M_ONLINE, -                ; Set unit online
    RSB     ; Return
    .SBTTL  TD_FDT_ROUTINE, Sample FDT routine
; ++
; TD_FDT_ROUTINE, Sample FDT routine
;
; Functional description:
;
;     T.B.S.
;
; Inputs:
;
;     R0-R2  - scratch registers
;     R3     - address of the IRP (I/O request packet)
;     R4     - address of the PCB (process control block)
;     R5     - address of the UCB (unit control block)
;     R6     - address of the CCB (channel control block)
;     R7     - bit number of the I/O function code
;     R8     - address of the FDT table entry for this routine
;     R9-R11 - scratch registers
;     AP     - address of the 1st function dependent QIO parameter
;
; Outputs:
;
;     The routine must preserve all registers except R0-R2, and
;     R9-R11.
;
; --
TD_FDT_ROUTINE:                              ; Sample FDT routine
    RSB     ; Return
    .SBTTL  TD_START, Start I/O routine

```

Template for a Device Driver

5.4 Driver Template

```
;++
; TD_START - Start a transmit, receive, or set mode operation
;
; Functional description:
;
;     T.B.S.
;
; 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
DEVICELOCK LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access.
        SAVIPL=-(SP)                ; Save current IPL

        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
DEVICEUNLOCK LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
        NEWIPL=#3,-                ; Lower IPL
        PRESERVE=NO                ; Don't preserve R0
        MOVZWL #SS$_TIMEOUT,R0     ; Return error status
        BSBB  COMPLETE_IO         ; Call I/O postprocessing
        DEVICELOCK LOCKADDR=UCB$L_DLCK(R5),- ; Acquire device lock for exit
        PRESERVE=NO

        RSB

        SBTTL TD_INTERRUPT, Interrupt service routine
```

Template for a Device Driver

5.4 Driver Template

```

; ++
; 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 REQPCAN macro call, and
;         then invoked the WFIKPC 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 PC
;     32(SP) - saved PSL (processor status longword)
;
;     The IDB contains the CSR address and the UCB address.
;
; Outputs:
;
;     The routine must preserve all registers except R0-R5.
;
; --
TD_INTERRUPT:
; Service device interrupt
    MOVL    @(SP)+,R4          ; Get address of IDB and remove
                                ; pointer from stack
    ASSUME  IDB$L_CSR EQ 0
    ASSUME  IDB$L_OWNER EQ 4
    MOVQ   IDB$L_CSR(R4),R4    ; Get address of device's CSR
                                ; Get address of device owner's UCB
    DEVICELOCK LOCKADDR=UCB$L_DLCK(R5), - ; Lock device access
    PRESERVE=NO,-            ; Don't preserve R0
    CONDITION=NOSETIPL      ; Don't bother setting our IPL
    BECC   #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$W_TD_STATUS(R5)      ; registers. First the CSR
    MOVW   TD_WRCNT(R4),-    ; Save the word count register
    UCB$W_TD_WRCNT(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 a Device Driver

5.4 Driver Template

```

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

;
; Dismiss the interrupt
;
UNSOL_INTERRUPT:
    DEVICEUNLOCK LOCKADDR=UCB$L_DLCK(R5),- ; Dismiss unsolicited interrupt
                                           ; Unlock device access
    PRESERVE=NO                      ; Don't bother preserving R0
    POPR   #^M<R0,R1,R2,R3,R4,R5>    ; Restore R0-R5
    REI                                     ; Return from interrupt

    .SBTTL TD_CANCEL, Cancel I/O routine

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

Template for a Device Driver

5.4 Driver Template

```

;
; Device-dependent cancel operations go next.
;
;
; Finally, the return.
;
10$:
    RSB                                ; Return
    .SBTTL TD_REG_DUMP, Device register dump routine

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

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

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

TD_END:                                ; Last location in driver
    .END

```


6

Writing Device-Driver Tables

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

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

The VMS operating system provides macros that drivers can invoke to create these tables.

6.1 Driver Prologue Table

The driver prologue table (DPT) is the first part of every device driver. This table, along with parameters to the SYSGEN command that request driver loading, describes the driver to the driver-loading procedure. In turn, the driver-loading procedure computes the size of the driver, loads it into nonpaged system memory, and creates data structures for the new device(s) in the I/O database. The loading procedure also links the new DPT into a list of all DPTs known to the system. Chapter 15 describes how the driver-loading procedure decides which data structures to build for a given device.

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

Figure A-10 illustrates the DPT data structure, and Table A-9 describes its contents. Drivers must treat many of the fields initialized by the driver-loading procedure as read-only fields. These fields are marked with an asterisk in Figure A-10.

To create a DPT, the driver invokes the DPTAB macro, as described in Appendix B. The DPTAB macro generates a driver prologue table (DPT) in a program section called \$\$\$105_PROLOGUE.

The DPTAB macro requires the following information:

- Address of the end of the driver in its **end** argument.
- Code identifying the device by its adapter type in the **adapter** argument. Accepted adapter types include UBA (for devices attached to either a UNIBUS or Q22 bus), MBA, and GENBI.
- Name of the driver in the **name** argument.

Writing Device-Driver Tables

6.1 Driver Prologue Table

- Size of the unit control block (UCB) in the **ucbsize** argument. (The template in Section 5.4 and the macro descriptions in Appendix B demonstrate how you can specify an extended UCB defined by VMS or create an extended UCB within a driver.)

The DPTAB also allows you to specify the following information, if applicable to the device driver:

- Whether the driver needs a permanently allocated system page
- Whether the driver has been written to run in a VMS symmetric multiprocessing system
- Name of a driver unloading routine, if any, to be called subject to a SYSGEN RELOAD command
- Maximum number of units supported by the driver (default is 8)
- Number of UCBs to be created when the driver is loaded by means of the SYSGEN autoconfiguration facility and the address of a unit delivery routine to be called by that facility

A driver follows the DPTAB macro invocation with several instances of the DPT_STORE macro. The DPT_STORE macro provides the driver with a means of communicating its initialization needs to the driver-loading procedure. When invoked, the DPT_STORE macro places information in the DPT that the driver-loading procedure uses to load specified values into specified fields. The DPT_STORE macro accepts two lists of fields:

- Fields to be initialized only when the driver is first loaded
- Fields to be initialized when a driver is first loaded and reinitialized if the driver is reloaded

The DPTAB macro stores the relative addresses of these two lists, called initialization and reinitialization tables, in the DPT.

Drivers use the DPT_STORE macro with the INIT table marker label to begin a list of DPT_STORE invocations that supply initialization data for the following fields:

| | |
|-------------|--|
| UCB\$B_FLCK | Index of the fork lock under which the driver performs fork processing. The DPTAB macro, in invoking the \$SPLCODDEF macro, defines the symbols for these indexes. |
| UCB\$B_DIPL | Device interrupt priority level. |

Other commonly initialized fields are

| | |
|------------------|-----------------------------|
| UCB\$_DEVCHAR | Device characteristics |
| UCB\$B_DEVCLASS | Device class |
| UCB\$B_DEVTYPE | Device type |
| UCB\$W_DEVBUFSIZ | Default buffer size |
| UCB\$Q_DEVDEPEND | Device-dependent parameters |

Writing Device-Driver Tables

6.1 Driver Prologue Table

Drivers use the `DPT_STORE` macro with the `REINIT` table marker label to begin a list of `DPT_STORE` invocations that supply initialization and reinitialization data for certain fields. Every driver must specify the following field in such an invocation:

`DDB$_DDT` Driver dispatch table

Other commonly initialized fields are

`CRB$_INTD+VEC$_ISR` Interrupt service routine.

`CRB$_INTD2+VEC$_ISR` Interrupt service routine for second interrupt vector.

`CRB$_INTD+VEC$_INITIAL` Controller initialization routine.

`CRB$_INTD+VEC$_UNITINIT` Unit initialization routine (for UNIBUS, Q22 bus, and generic VAXBI device drivers). Note that MASSBUS drivers must specify the address of the unit initialization routine in an invocation of the `DDTAB` macro.

For an example of the use of the `DPT` and `DPT_STORE` macros, see the description of the `DPTAB` macro in Appendix B.

6.2 Driver Dispatch Table

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

The routines listed in the DDT can reside in the driver module or in a VMS module. Appendix C describes the VMS device-independent routines that can be specified.

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

The driver creates a DDT by invoking the macro `DDTAB`. The `DDTAB` macro labels the DDT `devnam$DDT`, according to the value you supply in its **devnam** argument. The driver-loading procedure writes the address of the DDT table, as specified in a `DPT_STORE` macro, into the DDB. Figure A-9 illustrates the structure of a DDT and Table A-8 describes its contents.

The `DDTAB` macro also generates the program section (`$$$115_DRIVER`) in which the DDT itself and all driver code reside.

The `DDTAB` macro has a single required argument, **functb**, for which the driver must specify the address of its function decision table. Several optional arguments allow the driver to specify the names of the following routines, if applicable:

- Start-I/O routine
- Unsolicited interrupt service routine (for MASSBUS device drivers)
- Cancel-I/O routine
- Register dumping routine

Writing Device-Driver Tables

6.2 Driver Dispatch Table

- Unit initialization routine
- Alternate start-I/O routine
- Cloned UCB routine

In addition, you specify the length of any diagnostic buffer or error message buffer using the DDTAB macro.

See the description of the DDTAB macro in Appendix B for additional information.

6.3 Function Decision Table

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

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

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

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

Refer to Chapter 7 for information on the writing of FDT routines.

Table 6-1 lists the physical, logical, and virtual I/O function codes defined by VMS. Note that certain of the function codes listed have the same values in VMS Version 5.0. A complete list of function codes and values is contained in the macro \$IODEF in SYS\$LIBRARY:STARLET.MLB.

Writing Device-Driver Tables

6.3 Function Decision Table

Table 6–1 I/O Function Codes

| Function | Description | Equivalent Symbol(s) |
|---------------------|---|---|
| Physical I/O | | |
| IO\$_NOP | No operation | – |
| IO\$_UNLOAD | Unload drive (required by all disk drivers) | IO\$_LOADMCODE |
| IO\$_SEEK | Seek cylinder | IO\$_SPACEFILE (space files), IO\$_STARTMPROC (start microprocessor) |
| IO\$_RECAL | Recalibrate drive | IO\$_STOP (stop) |
| IO\$_DRVCLR | Drive clear | IO\$_INITIALIZE (initialize) |
| IO\$_RELEASE | Release port | IO\$_SETCLOCKP (set clock—physical) |
| IO\$_OFFSET | Offset read heads | IO\$_ERASETAPE (erase tape), IO\$_STARTDATAP (start data transfer—physical) |
| IO\$_RETCENTER | Return to center line | IO\$_QSTOP (queue stop request) |
| IO\$_PACKACK | Pack acknowledgment (required by all disk drivers) | – |
| IO\$_SEARCH | Search for sector | IO\$_SPACERECORD (space records), IO\$_READRCT (read replacement and caching table) |
| IO\$_WRITECHECK | Write check data | – |
| IO\$_WRITEPBLK | Write physical block | – |
| IO\$_READPBLK | Read physical block | – |
| IO\$_WRITEHEAD | Write header and data | IO\$_RDSTATS (read statistics), IO\$_CRESHAD (create a shadow set) |
| IO\$_READHEAD | Read header and data | IO\$_ADDSHAD (add member to shadow set) |
| IO\$_WRITETRACKD | Write track data | IO\$_COPYSHAD (perform shadow set copy operations) |
| IO\$_READTRACKD | Read track data | IO\$_REMSHAD (remove member from shadow set) |
| IO\$_AVAILABLE | Set device available (required by all disk drivers) | – |
| IO\$_SETPRFPATH | Set preferred path | – |
| IO\$_DSE | Data security erase (and rewind) | – |
| IO\$_REREADN | Reread next | – |
| IO\$_REREADP | Reread previous | – |
| IO\$_WRITERET | Write retry | IO\$_WRITECHECKH (write check header and data) |
| IO\$_READPRESET | Read in preset | IO\$_STARTSPNDL (start spindle) |
| IO\$_SETCHAR | Set device characteristics | – |
| IO\$_SENSECHAR | Sense device characteristics | – |
| IO\$_WRITEMARK | Write tape mark | – |
| IO\$_WRITTMKR | Write tape mark retry | IO\$_DIAGNOSE (diagnose), IO\$_SHADMV (perform mount verification on shadow set) |
| IO\$_FORMAT | Format | IO\$_CLEAN (clean tape) |

Writing Device-Driver Tables

6.3 Function Decision Table

Table 6–1 (Cont.) I/O Function Codes

| Function | Description | Equivalent Symbol(s) |
|--------------------|-----------------------------------|-------------------------------------|
| Logical I/O | | |
| IO\$_WRITEBLK | Write logical block | – |
| IO\$_READBLK | Read logical block | – |
| IO\$_REWINDOFF | Rewind and set offline | – |
| IO\$_SETMODE | Set mode | – |
| IO\$_REWIND | Rewind tape | – |
| IO\$_SKIPFILE | Skip files | – |
| IO\$_SKIPRECORD | Skip records | – |
| IO\$_SENSEMODE | Sense mode | – |
| IO\$_WRITEOF | Write end of file | – |
| IO\$_TTY_PORT | Terminal port FDT routine | IO\$_FREECAP (return free capacity) |
| IO\$_FLUSH | Flush controller cache | – |
| Virtual I/O | | |
| IO\$_WRITEVBLK | Write virtual block | – |
| IO\$_READVBLK | Read virtual block | – |
| IO\$_ACCESS | Access file | – |
| IO\$_CREATE | Create file | – |
| IO\$_DEACCESS | Deaccess file | – |
| IO\$_DELETE | Delete file | – |
| IO\$_MODIFY | Modify file | – |
| IO\$_NETCONTROL | X25 network control function | – |
| IO\$_READPROMPT | Read terminal with prompt | IO\$_SETCLOCK (set clock) |
| IO\$_ACPCONTROL | Miscellaneous ACP control | IO\$_STARTDATA (start data) |
| IO\$_MOUNT | Mount volume | – |
| IO\$_TTYREADALL | Terminal read passall | – |
| IO\$_TTYREADPALL | Terminal read with prompt passall | – |
| IO\$_CONINTREAD | Connect to interrupt read-only | – |
| IO\$_CONINTWRITE | Connect to interrupt with write | – |

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

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

Writing Device-Driver Tables

6.3 Function Decision Table

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

See the description of the FUNCTAB macro, and the example of its use, in Appendix B for additional information on creating an FDT.

6.3.1 Defining Buffered-I/O Functions

The second entry in an FDT is a *buffered function bit mask* that indicates which legal functions the driver handles as buffered-I/O operations. In selecting the functions that are to be buffered, you should take the following information into consideration:

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

6.3.2 Defining Device-Specific Function Codes

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

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


7

Writing FDT Routines

The \$QIO system service uses the driver's function decision table (FDT) to determine which FDT routines to call to preprocess an I/O request. These FDT routines validate process-specified arguments to the \$QIO request. VMS supplies many device-independent FDT routines. Device drivers contain device-dependent FDT routines.

A driver should call the VMS device-independent FDT routines, described in Section 7.5, whenever possible. This practice encourages the use of well debugged routines and minimizes driver size.

7.1 Context of FDT Routine Execution

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

- It cannot call VMS system services or VMS RMS services.
- It does not lower IPL below IPL\$_ASTDEL. If a routine raises IPL, it must obtain any appropriate spin lock, and it must lower IPL to IPL\$_ASTDEL before exiting, releasing any acquired spin lock.
- It does not alter the stack without restoring its original state before exiting.
- If it issues a subroutine call, it must preserve the contents of R3 through R8 across the call. It can, however, use R0 through R2 and R9 through R11 without saving their previous contents. If an FDT routine needs to use R3 through R8, it can use the PUSHR and POPR instructions to save registers on the stack and later restore them.
- It exits either by an RSB instruction to return control to the system service, or it issues a JMP instruction to one of the VMS routines described in Section 7.2.1.

Before calling an FDT routine, the \$QIO system service sets up the contents of certain registers, as described in Table 7-1.

Table 7-1 Registers Loaded by the \$QIO System Service

| Register | Content |
|----------|---|
| R0 | Address of FDT routine being called |
| R3 | Address of IRP for current I/O request |
| R4 | Address of process control block (PCB) of current process |

Writing FDT Routines

7.1 Context of FDT Routine Execution

Table 7–1 (Cont.) Registers Loaded by the \$QIO System Service

| Register | Content |
|----------|---|
| R5 | Address of UCB of device assigned to user-specified process-I/O channel |
| R6 | Address of CCB that describes user-specified process-I/O channel |
| R7 | Bit number of user-specified I/O function code |
| R8 | Address of current entry in FDT |
| AP | Address of first function-dependent argument (p1) specified in I/O request |

While FDT routines can perform extensive preprocessing, such as determining whether user buffers are accessible and reformatting data into buffers in the system address space, they should not access device registers because the device might be active. Furthermore, FDT routines should exercise restraint when modifying the UCB. Routines usually access the UCB while holding the associated fork lock at driver fork IPL to synchronize modifications, and FDT routines do not execute with such synchronization. Drivers containing FDT routines that access device registers or carelessly modify the UCB risk unpredictable operation or a system failure.

7.2 FDT Routines and Their Exit Paths

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

```
JSB    (R0)
```

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

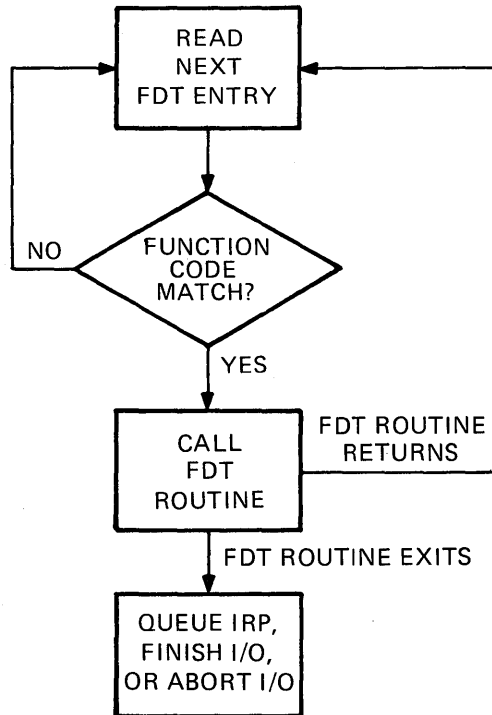
- Whether another FDT routine needs to be called to perform additional function-specific processing
- Whether an error is found in the I/O request
- Whether the operation is complete
- Whether the I/O operation requires and is ready for device activity

The FDT routines, as illustrated in Figure 7–1, must transfer control out of the FDT processing loop and into a VMS routine that queues an IRP, completes an I/O request, or aborts an I/O request. The \$QIO system service does not stop scanning the FDT. Therefore, you must ensure that for each valid function code in a driver's FDT, there is an FDT routine that does not return control to the \$QIO system service.

Writing FDT Routines

7.2 FDT Routines and Their Exit Paths

Figure 7-1 \$QIO Scan of a Function Decision Table



ZK-926-82

7.2.1 FDT Exit Paths

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

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

These methods are described in the following sections, and you can find additional details on the routines they involve in Appendix C.

7.2.1.1 RSB

An FDT routine issues an RSB instruction to return to the \$QIO system service. The FDT routine returns to the system service because the routine knows that the FDT contains a subsequent entry with the same function code bit set. As a result, the system service searches for another FDT routine.

Writing FDT Routines

7.2 FDT Routines and Their Exit Paths

7.2.1.2 **JMP G^EXE\$QIODRVPKT**

EXE\$QIODRVPKT transfers control to a VMS routine that queues an IRP to a driver. The FDT routine uses this exit method if all preprocessing is complete, if no fatal errors are found in the specification of an I/O request, and if device activity, synchronized access to the device's UCB, or synchronized access to device registers is required to complete the I/O request. Common examples of such a request are read and write functions.

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

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

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

7.2.1.3 **JMP G^EXE\$FINISHIO or JMP G^EXE\$FINISHIOC**

EXE\$FINISHIO and EXE\$FINISHIOC transfer control to a VMS routine that writes a quadword of final I/O status from R0 and R1 into the I/O status field of the IRP (IRP\$L_MEDIA and IRP\$L_MEDIA+4). (Note that EXE\$FINISHIOC clears the second longword of the final I/O status.) The routine then inserts the IRP in the I/O postprocessing queue. These routines return to the \$QIO system service the two longwords of status contained in the I/O status block (if any) specified in the I/O request.

An FDT routine that discovers a device-dependent error should always return status using EXE\$FINISHIO or EXE\$FINISHIOC. 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.

7.2.1.4 **JMP G^EXE\$ABORTIO**

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

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. EXE\$ABORTIO stores a longword of status in R0 and returns this to the system service.

Writing FDT Routines

7.2 FDT Routines and Their Exit Paths

7.2.1.5 JSB G^EXE\$ALTQUEPKT

EXE\$ALTQUEPKT transfers control to a VMS routine that calls an alternate start-I/O routine in the driver (specified in the driver dispatch table at offset DDT\$L_ALTSTART) that synchronizes requests for activity on a device unit and initiates the processing of I/O requests.

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

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

Unlike the other FDT exit routines, EXE\$ALTQUEPKT is called with a JSB instruction rather than a JMP instruction. When the alternate start-I/O routine finishes, it returns control to EXE\$ALTQUEPKT by executing an RSB instruction. The FDT routine performs any postprocessing and transfers control to the routine EXE\$QIORETURN. When EXE\$QIORETURN gains control, it performs the following steps:

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

7.3 FDT Routines for VMS Direct I/O

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

EXE\$READ and EXE\$WRITE are described in Section 7.5.

Writing FDT Routines

7.4 FDT Routines for VMS Buffered I/O

7.4 FDT Routines for VMS Buffered I/O

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

An FDT routine for a buffered I/O data transfer operation should confirm either read or write access to the user's buffer and allocate a buffer in system space. Sections 7.4.1 and 7.4.2 describe these tasks.

An FDT routine for a buffered I/O operation that does not involve data transfer should copy the function-dependent parameters of the \$QIO request (p1 to p6) to the IRP, perform any necessary preprocessing, and use one of the exit methods listed in Section 7.2.1.

7.4.1 Checking Accessibility of the User's Buffer

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

7.4.2 Allocating the System Buffer

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

- 1 It adds 12 bytes to the byte count passed in the p2 argument of the user's I/O request, thus accommodating the standard size of a VMS buffer header. This is the total system buffer size.
- 2 It calls EXE\$DEBIT_BYTCNT_ALO to ensure that the process's job has sufficient remaining byte count quota to allow its use of the requested buffer. If the job has sufficient quota, EXE\$DEBIT_BYTCNT_ALO allocates the requested buffer from nonpaged pool, writes the buffer's size and type into its third longword, and subtracts the system buffer size from JIB\$L_BYTCNT.

VMS also supplies the routines EXE\$DEBIT_BYTCNT_BYTLM_ALO, EXE\$DEBIT_BYTCNT(_NW), EXE\$DEBIT_BYTCNT_BYTLM(_NW), and EXE\$ALLOCBUF which perform the same type of work as EXE\$DEBIT_BYTCNT_ALO. These routines are fully described in Appendix C.

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

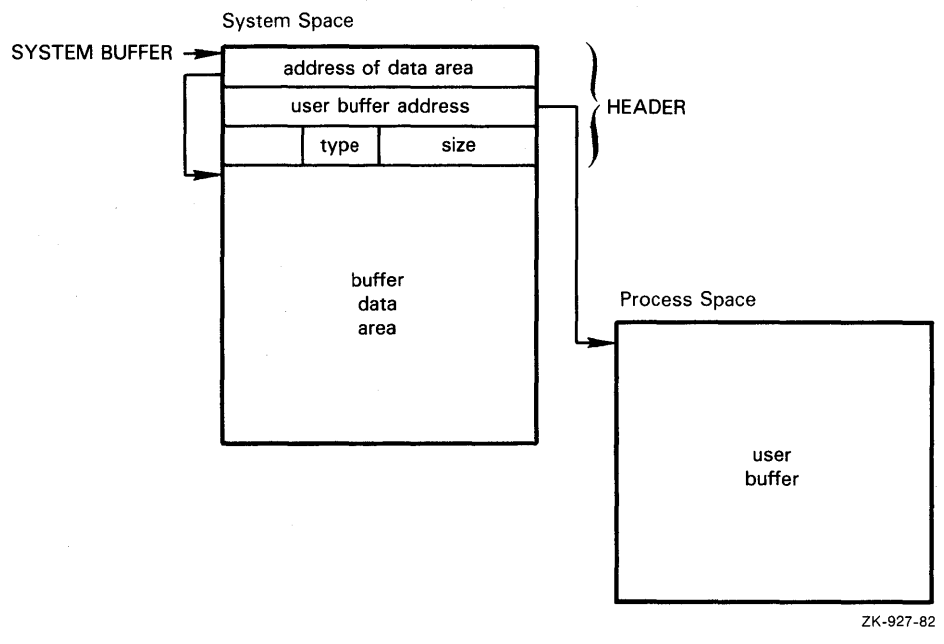
- 1 Loads the address of the system buffer into IRP\$L_SVAPTE.
- 2 Loads the total size of the system buffer into IRP\$W_BOFF.
- 3 Stores the starting address of the system buffer data area in the first longword of the buffer header.
- 4 Stores the user's buffer address in the second longword of the header.
- 5 Copies data from the user buffer to the system buffer if the I/O request is a write operation.

Writing FDT Routines

7.4 FDT Routines for VMS Buffered I/O

At this point, the buffers are ready for the transfer. Figure 7-2 illustrates the format of the system buffer.

Figure 7-2 Format of System Buffer for a Buffered-I/O Read Function



7.4.3 Buffered-I/O Postprocessing

When the transfer finishes, the driver returns control to VMS for completion of the I/O request. The driver writes the final request status in the low-order word of R0. Use of the high-order word of R0 and the longword of R1 is driver specific. Certain drivers use these fields to report a transfer byte count, for example.

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

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

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

- 1 Obtains the address of the system buffer from IRP\$L_SVAPTE.

Writing FDT Routines

7.4 FDT Routines for VMS Buffered I/O

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

7.5 FDT Routines Provided by VMS

The 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 VMS FDT routine is called, no additional FDT processing is required. All of the VMS FDT routines listed in Table 7-2 exit by transferring control to EXE\$QIODRVPKT, EXE\$FINISHIO, EXE\$FINISHIOC, or EXE\$ABORTIO. Once a VMS FDT routine is called, no subsequent FDT processing occurs.

For additional information about VMS FDT routines, see the pertinent routine descriptions in Appendix C.

Table 7-2 FDT Routines Provided by VMS

| FDT Routine | Function | Exit Method |
|---------------------------|--|--|
| EXE\$ONEPARG | Processes a nontransfer I/O function code that has one parameter associated with it | Transfers control to EXE\$QIODRVPKT |
| EXE\$READ | Processes a logical-read or physical-read function for a direct I/O operation | 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, transfers control to EXE\$QIODRVPKT |
| EXE\$SENSEMODE | Processes the sense-device-mode and sense-device-characteristics functions by reading fields of the UCB | Transfers control to EXE\$FINISHIO |
| EXE\$SETCHAR ¹ | Processes the set-device-mode and set-device-characteristics functions | Transfers control to EXE\$FINISHIO |
| EXE\$SETMODE ¹ | Processes the set-device-mode and set-device-characteristics functions by creating a driver fork process | Aborts the I/O request if an error occurs; otherwise, transfers control to EXE\$QIODRVPKT |

¹If 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.

Writing FDT Routines

7.5 FDT Routines Provided by VMS

Table 7-2 (Cont.) FDT Routines Provided by VMS

| FDT Routine | Function | Exit Method |
|--------------------|---|--|
| EXE\$WRITE | Processes a logical-write or physical-write function for a direct I/O operation | 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, transfers control to EXE\$QIODRVPKT |
| EXE\$ZEROPARM | Processes a nontransfer I/O function code that has no associated parameters | Transfers control to EXE\$QIODRVPKT |

8

Writing a Start-I/O Routine

A driver start-I/O routine activates a device and then waits for a device interrupt or timeout. This chapter describes the start-I/O routine. Chapter 10 describes the reactivation of the driver routine that performs device-dependent I/O postprocessing. With a few exceptions, the start-I/O routine discussed in the following sections describes a DMA transfer using a single-unit controller.

8.1 Transferring Control to the Start-I/O Routine

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

When FDT processing is complete for an I/O request, the FDT routine transfers control to EXE\$QIODRVPKT. If the designated device is idle, IOC\$INITIATE is called to create a driver fork process. (This procedure is detailed in Section 7.2.1.2.) The driver fork process then gains control in the start-I/O routine of the appropriate driver. If the device is busy, EXE\$QIODRVPKT calls EXE\$INSIOQ, which queues the packet to the device unit's pending-I/O queue.

After a device completes an I/O operation, the driver fork process exits by transferring control to IOC\$REQCOM. IOC\$REQCOM inserts the IRP for the finished transfer into the postprocessing queue. It then dequeues the next IRP from the device unit's pending-I/O queue and calls IOC\$INITIATE to initiate the processing of this I/O request in the driver's fork process at the entry point of the driver's start-I/O routine.

8.2 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 context | Driver code can only refer to system virtual addresses. |
| Kernel mode | Execution occurs in the most privileged access mode and can, therefore, change IPL and obtain spin locks. |
| High IPL | The VMS routine that creates a driver fork process obtains the driver's fork lock, raising IPL to driver fork level before activating the driver. |
| Kernel or interrupt stack | Execution occurs on the kernel or interrupt stack. The driver must not alter the state of the stack without restoring the stack to its previous state before relinquishing control. The stack used depends on whether the I/O startup is the result of a new I/O request or because a previously requested I/O operation has been completed. The choice of stacks must not affect the operation of the start-I/O routine. |

Writing a Start-I/O Routine

8.2 Context of a Driver Fork Process

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

- R3 contains the address of the IRP.
- R5 contains the address of the UCB for the device.

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

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

- IRP\$L_BCNT (low-order word) → UCB\$W_BCNT
- IRP\$W_BOFF → UCB\$W_BOFF
- IRP\$L_SVAPTE → UCB\$L_SVAPTE

8.3 Functions of a Start-I/O Routine

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

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

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

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

The following sections describe the general activities of a start-I/O routine for a typical device. The details of DMA processing are specific to the particular device. Section 12.2 describes the UNIBUS- and Q22 bus-related details of DMA transfers. Section 13.5.3 relates those tasks that MASSBUS DMA device drivers must perform. Section 14.5 discusses similar functions that drivers for generic VAXBI devices may need to perform.

Writing a Start-I/O Routine

8.3 Functions of a Start-I/O Routine

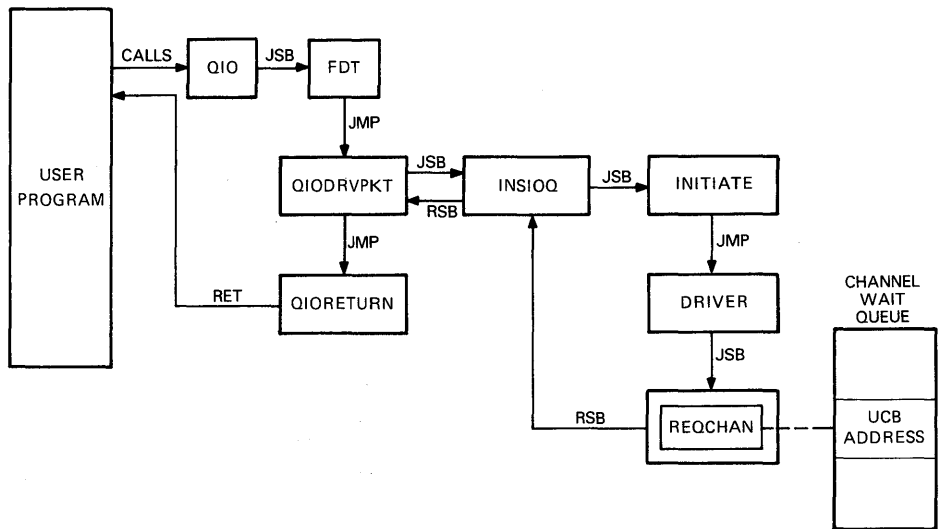
8.3.1 Obtaining Controller Access

If the device is one of several attached to a controller, the start-I/O routine invokes the VMS macro REQCHAN to assign the controller's data channel to the device unit. Controllers that control only one device do not require arbitration for the controller's data channel. REQCHAN calls the VMS routine IOC\$REQCHANL that acquires ownership of the controller data channel.

The transfer being controlled by the start-I/O routine discussed here requires no seek preceding the transfer. Disk I/O is an example of a transfer that requires a seek first. To permit seeks to be overlapped with transfers, invoke 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 in the channel wait queue. IOC\$REQCHANL then returns control to the caller of the driver, that is, to EXE\$INSIOQ, as illustrated in Figure 8-1. This procedure is further discussed in Section 3.4.1.

Figure 8-1 Inserting a UCB into the Channel Wait Queue



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

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

Note that, because IOC\$RELCHAN moves the address of the device's CSR into R4 before resuming a suspended driver, IOC\$REQCHANL does not save R4 in the UCB fork block.

Writing a Start-I/O Routine

8.3 Functions of a Start-I/O Routine

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

UCB → CRB → IDB

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

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

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:

- 00(SP)—Address of the next instruction to be executed in the driver fork process. The transfer of control to IOC\$REQPCHANL places this address on the stack.
- 04(SP)—Address of the next instruction to be executed in the routine that called the driver start-I/O routine.

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

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

At this point, a UNIBUS/Q22 bus DMA driver follows procedures to obtain I/O bus resources and compute the size and starting address of a transfer. These procedures are discussed in Section 12.2. MASSBUS DMA device drivers perform the steps indicated in Section 13.5.3.

8.3.3 Preparing the Device Activation Bit Mask

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

When the start-I/O routine copies the contents of the register into the device's CSR, the device starts the transfer. Before activating the device, however, the start-I/O routine should perform the steps described in Sections 8.3.4 and 8.3.5.

Writing a Start-I/O Routine

8.3 Functions of a Start-I/O Routine

8.3.4 Synchronizing Access to the Device Database

The start-I/O routine invokes the VMS macro `DEVICELock` to synchronize its access to device registers with the interrupt service routine. This macro invocation is doubly important, for it establishes the context wherein the driver can later issue the wait-for-interrupt macro (`WFIKPC` or `WFIRLCH`). The wait-for-interrupt macros expect the driver's fork IPL to be on the stack, as placed there by the `DEVICELock` macro. In addition, the wait-for-interrupt macros issue the `DEVICEUNLOCK` macro to release ownership of the device lock and restore the previous IPL.

8.3.5 Checking for a Local Processor Power Failure

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

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

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

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

- Clear `UCB$_POWER`
- Issue the `DEVICEUNLOCK` macro to release the device lock and restore IPL to fork IPL

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

8.3.6 Activating the Device

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

Writing a Start-I/O Routine

8.4 Waiting for an Interrupt or Timeout

8.4 Waiting for an Interrupt or Timeout

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

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

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

| | |
|---------|--|
| WFIKPCH | Wait for an interrupt or timeout and keep the controller data channel |
| WFIRLCH | Wait for an interrupt or timeout and release the controller data channel |

The WFIKPCH and WFIRLCH macros require the address of a timeout handling routine in the **excpt** argument. Optionally, but almost always, the driver can also indicate the number of seconds the system must wait before signaling a timeout in the **time** argument. A full description of these macros appears in Appendix C.

Both macros invoke routines that release ownership of the device lock, relinquish synchronization, and return IPL to the previous level when exiting. These routines expect to find the return IPL on the stack. This IPL is saved on the stack by the DEVICELOCK macro as described in Section 8.3.4.

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

8.4.1 Expansion of WFIKPCH Macro

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

If the driver specifies the **time** argument in the macro call, the macro pushes the value of the argument into the stack. If the **time** argument is not specified, the macro pushes the value 65,536 onto the stack. IOC\$WFIKPCH uses the time value to calculate the length of time VMS waits before transferring control to a device timeout handler.

WFIKPCH completes its expansion with two lines of code:

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

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

Writing a Start-I/O Routine

8.4 Waiting for an Interrupt or Timeout

8.4.2 IOC\$WFIKPCH Routine

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

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

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

The VMS routine that detects a device timeout calculates the address of the driver's timeout routine by subtracting 2 from the saved PC in the UCB's fork block and calling indirectly through the result. For example:

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

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

Finally, IOC\$WFIKPCH reenables interrupts by releasing the device lock and lowering IPL to fork level, the IPL at which the driver was executing previously. It then returns control to the caller of the driver.

9

Writing an Interrupt Service Routine

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

The mechanism of interrupt dispatching has no direct bearing on the contents of a driver's interrupt service routine. Its implementation varies slightly according to the VAX processing system and I/O subsystem in use. To obtain background information on the dispatcher, refer to the overview provided in Section 12.3, which also details the method of dispatching UNIBUS/Q22 bus device interrupts. MASSBUS device driver writers should refer also to Section 13.4; generic VAXBI device driver writers should read the discussion in Section 14.3.1.

For most device drivers, the driver prologue table contains, in the reinitialization section established by the `DPT_STORE` macro, the address of one or more interrupt service routines. Each interrupt service routine corresponds to an interrupt vector on the I/O bus. You specify the address of an I/O bus vector using the `SYSGEN` command `CONNECT`, as described in Section 15.2.2.

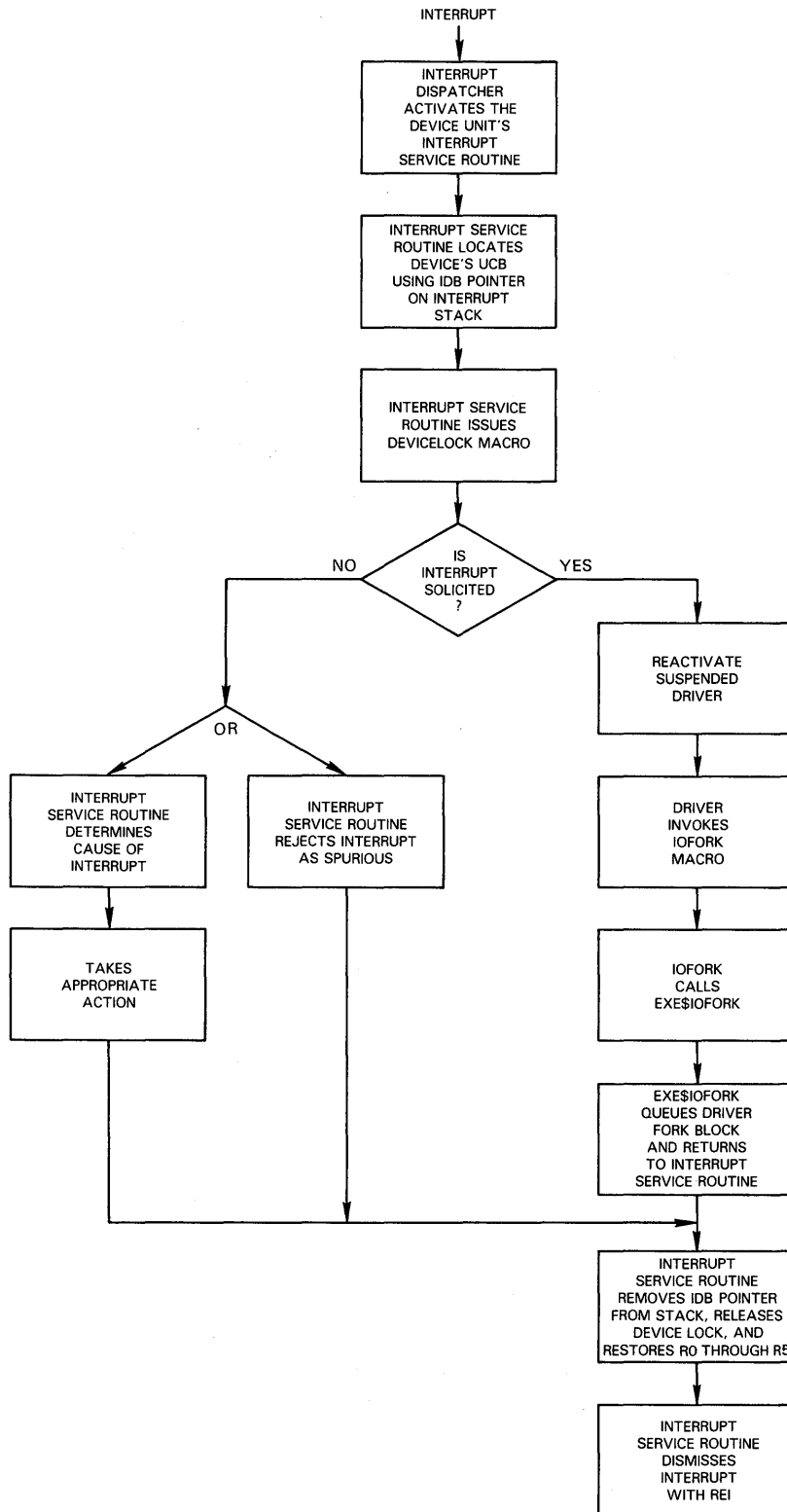
Most device interrupt service routines perform the following functions:

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

Figure 9-1 illustrates the general flow of interrupt handling. The remaining sections of this chapter describe the handling of solicited and unsolicited interrupts in further detail.

Writing an Interrupt Service Routine

Figure 9-1 Flow of Interrupt Servicing



ZK-929-82

Writing an Interrupt Service Routine

9.1 Interrupt Context

9.1 Interrupt Context

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

- R0 through R5 are saved on the stack.
- Only system address space may be accessed.
- IPL is at hardware device interrupt level.
- The processor is running in kernel mode.
- The processor is running on the interrupt stack.

The stack contains the following information:

| Stack Location | Content |
|-----------------------|-----------------------------------|
| 00(SP) | Pointer to the address of the IDB |
| 04(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 |

In the course of its processing, an interrupt service routine must remove the IDB pointer and the saved registers from the stack before dismissing the interrupt with an REI instruction.

9.2 Servicing a Solicited Interrupt

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

- A description of the I/O request and the state of the device
- The contents of R3 and R4
- The implicit contents of R5 (the address of the UCB fork block)
- The address at which to return control to the driver
- The implicit address of a timeout handling routine

When the interrupt service routine returns control to the main line of driver processing, it has only restored the contents of R3, R4, R5, and the PC.

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

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

00(SP) → CRB → IDB → IDB\$L_OWNER → UCB

The interrupt service routine restores the UCB address to R5.

- 2 Issues the DEVICELOCK macro to obtain synchronized access to device registers.

Writing an Interrupt Service Routine

9.2 Servicing a Solicited Interrupt

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

Note: Because device timeout processing mostly occurs at fork IPL (see Section 10.2), a driver's interrupt service routine, executing at device IPL, could interrupt the processing of a timeout on the same device unit. For this reason, the driver's interrupt service routine should check the interrupt-expected bit (UCB\$V_INT) before handling the interrupt. VMS clears this bit before it calls the driver's timeout handler.

- 4 Obtains device-status or controller-status information from the device registers, if necessary, and stores the status information in the UCB.
- 5 Places the contents of UCB\$L_FR3 and UCB\$L_FR4 in R3 and R4, respectively.
- 6 Issues a JSB instruction to the waiting driver's PC address, which is saved in the UCB fork block at UCB\$L_FPC.

The restored driver should execute as briefly as possible in interrupt context. As soon as possible, the driver should invoke the IOFORK macro to request the creation of a fork process at the driver's fork IPL. It must do this in order to complete the I/O operation. Forking lowers the IPL of driver execution below device IPL, allowing the processor to service additional device interrupts. IOFORK calls the routine EXE\$IOFORK. EXE\$IOFORK inserts into the appropriate fork queue the UCB fork block that describes the driver process. It then returns control to the driver's interrupt service routine. (See Section 10.1.1 for additional information on driver forking.)

The interrupt service routine then performs the following steps:

- 1 Removes the IDB pointer from the stack
- 2 Issues the DEVICEUNLOCK macro to release ownership of the device lock
- 3 Restores R0 through R5
- 4 Dismisses the interrupt with an REI instruction

9.3 Servicing an Unsolicited Interrupt

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

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

Writing an Interrupt Service Routine

9.3 Servicing an Unsolicited Interrupt

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

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

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

As mentioned in Section 9.2, an interrupt service routine first obtains the address of the device's UCB from the IDB. It then issues the `DEVICELock` macro to obtain synchronized access to device registers.

The routine determines whether an interrupt is solicited or not by examining the interrupt-expected bit in the UCB status longword (`UCB$V_INT` in `UCB$L_STS`). All UNIBUS, Q22 bus, and generic VAXBI device drivers must use this method to determine whether or not an interrupt is solicited; the address of the unsolicited interrupt service routine, specified in the driver dispatch table, is used only by MASSBUS drivers (see Sections 13.4 and 13.6.)

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

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

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

If `UCB$V_BSY` is set in `UCB$L_STS`, the UCB fork block is currently in use by the driver's start-I/O routine. An attempt by the interrupt service routine to concurrently use the fork block can destroy the fork context already stored in that UCB. Moreover, if `UCB$V_BSY` is not set, the interrupt service routine cannot safely assume that the fork block is not in use, for it may be currently employed to service a previous unsolicited interrupt.

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

Writing an Interrupt Service Routine

9.3 Servicing an Unsolicited Interrupt

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

The VMS routine that creates the fork process (once these conditions are satisfied) returns control to the interrupt service routine. The interrupt service routine then releases the device lock, restores the saved registers, and issues an REI instruction to dismiss the interrupt.

9.3.1 Examples of Unsolicited Interrupts

A card reader requests an unsolicited interrupt when a user puts the reader online. Once the card-reader driver's interrupt service routine determines that the interrupt is unsolicited, the routine analyzes the interrupt, as in the following code example.

Because only one sequence of instructions can use the UCB as a fork block, the interrupt service routine performs the following steps before it creates the fork process:

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

```
CR$INT::
  MOVL   @(SP)+,R3                ;Get address of IDB1
  MOVQ   IDB$L_CSR(R3),R4         ;Get controller CSR and owner UCB address2
  DEVICELOCK LOCKADDR=UCB$L_DLCK(R5),-
        PRESERVE=NO,-
        CONDITION=NOSETIPL      ;Obtain device lock3
  BBCC   #UCB$V_INT,UCB$L_STS(R5),10$ ;If clear, interrupt not expected4
  .
  .
;
; UNSOLICITED INTERRUPT
;
10$:   MOVZWL CR_CSR(R4),R0        ;Get reader status
       MOVZBW #CR_CSR_M_IE,CR_CSR(R4) ;Clear status, enable interrupts5
       BITW  #CR_CSR_M_ONLINE,R0    ;Reader transition to online?6
       BEQL  20$                    ;If equal no
       TSTW  UCB$W_REFC(R5)         ;Device assigned or allocated?7
       BNEQ  20$                    ;If not equal yes
       BSS   #UCB$V_JOB,UCB$W_DEVSTS(R5),-
           20$                      ;If set, message already sent8
       BSBB  30$                    ;Send message to job controller
20$:   DEVICEUNLOCK LOCKADDR=UCB$L_DLCK(R5),-
       PRESERVE=NO                 ;Release device lock
       MOVQ  (SP)+,R0               ;Restore registers
       MOVQ  (SP)+,R2
       MOVQ  (SP)+,R4
       REI
```


Writing an Interrupt Service Routine

9.3 Servicing an Unsolicited Interrupt

```

30$:  FORK                                ;Create fork process⑨
      MOVZBL #MSG$_CRUNSOLIC,R4          ;Set message type⑩
      MOVL  G^SYS$AR_JOBCTLMB,R3        ;Set address of job controller mailbox
      JSB   G^EXE$SNDEVMSG              ;Sent message to job controller
      BLBS  R0,40$                      ;If LBS successful notification⑪
      BICW  #UCB$_JOB,UCB$_DEVSTS(R5)   ;Clear message sent bit⑫
40$:  RSB

```

- ① The interrupt service routine obtains the address of the IDB from the top of the stack.
- ② By means of this action, it obtains the address of the control and status register (CSR) in R4 and restores the address of the UCB in R5.¹
- ③ It issues a DEVICELOCK macro to secure synchronized access to device registers and UCB fields.
- ④ It checks for an unsolicited interrupt by testing the interrupt expected bit in the UCB status longword.
- ⑤ Because the interrupt is unsolicited, the routine clears all CSR bits except for the interrupt-expected bit.
- ⑥ It confirms that the reader was just placed on line by examining a saved copy of the CSR.
- ⑦ It examines the reference count field of the device's UCB (UCB\$_REFC) to determine whether a process has allocated the device or assigned a channel to it.
- ⑧ If the reference count is zero, the interrupt service routine tests the job-attached bit in the device-dependent status field (UCB\$_JOB in UCB\$_DEVSTS) to make sure it has not already sent the job controller a message about the card reader being placed on line.
- ⑨ If the job-attached bit is not set, the routine sets the bit and creates a fork process to send the message to the job controller, using the system routine EXE\$SNDEVMSG (described in Appendix C). It is necessary to lower IPL from device IPL by forking at this point because EXE\$SNDEVMSG expects its caller's IPL to be no greater than IPL\$_MAILBOX.

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

- ⑩ When the fork process created at step 8 eventually executes, it writes a message to the job controller's mailbox, indicating that the card reader is on line.
- ⑪ If the fork process successfully sends the message, it leaves the job-attached bit set to prevent the job controller from receiving any further messages about the card reader's state. (The driver's cancel-I/O routine later clears the bit.)

¹ Because the card reader has a dedicated controller, the IDB\$_OWNER field always points to the UCB for the single unit:

00(SP) → CRB → IDB → IDB\$_OWNER → UCB

Writing an Interrupt Service Routine

9.3 Servicing an Unsolicited Interrupt

- 12 If the send-message request fails, the fork process clears the job-attached bit so that if the card reader makes a subsequent state change to on line, the interrupt service routine can attempt again to send a message to the job controller.

Another example of unsolicited interrupt processing occurs in a device driver for a multiunit controller. When a disk is placed off line, the disk drive hardware requests an interrupt. The driver interrupt service routine must determine what device unit requested the interrupt, obtain status information from the disk device's CSR, and then decide whether the interrupt was solicited.

Because it must access device UCB fields and device registers, the interrupt service routine first obtains the appropriate device lock. If the interrupt is unexpected, it calls code that services the unsolicited interrupt. This code checks the status of the volume, as described in the following steps:

- 1 It sets a bit in the UCB to indicate that the unit is on line (UCB\$V_ONLINE in UCB\$L_STS).
- 2 If the UCB's volume-valid bit is set (UCB\$V_VALID in UCB\$L_STS), the routine tests the volume valid status bit in a device register to determine whether the volume status has changed. If the volume is no longer valid, the routine clears the UCB volume valid bit.
- 3 The routine returns control to the driver's interrupt service routine.

The driver's 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, releases the device lock, restores registers R0 through R5, and dismisses the interrupt with an REI instruction.

10 Completing an I/O Request and Handling Timeouts

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

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

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

10.1 I/O Postprocessing

Once the driver interrupt service routine has processed 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 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.

10.1.1 EXE\$IOFORK

IOFORK generates a call to the routine EXE\$IOFORK. EXE\$IOFORK converts the driver context from that of an interrupt service routine to that of a fork process by performing the following steps:

- 1 It disables software timeouts by clearing the timeout enable bit in the UCB status longword (UCB\$V_TIM in UCB\$L_STS).
- 2 It saves R3 and R4 of the current driver context in the UCB fork block (UCB\$L_FR3 and UCB\$L_FR4).
- 3 It saves the current driver PC in the UCB fork block (UCB\$L_FPC). (The driver PC is the top longword on the stack, as a result of the JSB to EXE\$IOFORK.)
- 4 It obtains the fork lock index of the driver from the UCB (UCB\$B_FLCK) and uses it to determine in which fork queue it should place the fork block.
- 5 It inserts the address of the UCB fork block (R5) into the processor-specific fork queue corresponding to the driver's fork IPL.

Completing an I/O Request and Handling Timeouts

10.1 I/O Postprocessing

- 6 Finally, if the fork block is the first entry in the fork queue, EXE\$IOFORK requests a software interrupt from the local processor at the driver's fork IPL.

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

10.1.2 Completing an I/O Request

When VMS reactivates a driver's fork process by dequeuing the fork block, the driver resumes processing of the I/O operation holding the appropriate fork lock at fork IPL. Generic VAXBI devices perform whatever device-dependent operations are needed to prepare an I/O request for completion. If the device has completed the I/O operation without errors, a UNIBUS/Q22 bus driver for a DMA device proceeds as follows:

- 1 Purges the data path
- 2 Releases the buffered data path (applies only to UNIBUS DMA device drivers)
- 3 Releases map registers (does not apply to MicroVAX I DMA device drivers)
- 4 Releases the controller (applies only to drivers of devices on multiunit controllers)
- 5 Checks device register images saved in the UCB to determine the status of the I/O operation
- 6 Saves in the IRP the status code, transfer count, and device-dependent status that is to be returned to the user process in an I/O status block
- 7 Returns control to the operating system

The first three steps listed previously apply to UNIBUS/Q22 bus DMA transfers only and are discussed in Section 12.2. The following sections describe the last three steps.

10.1.2.1 Releasing the Controller

To release the controller channel, the driver code invokes the VMS macro RELCHAN. RELCHAN calls the VMS routine IOC\$RELCHAN. If another driver is waiting for the controller channel, IOC\$RELCHAN grants that driver's fork process the channel, restores its context from the UCB fork block, and transfers control to the saved PC. When no more drivers are awaiting the channel, IOC\$RELCHAN returns control to the fork process that released the channel.

Drivers for devices with dedicated controllers need not release the controller's data channel (as discussed in Sections 8.3.1 and 11.1). By means of code in the unit initialization routine, these drivers set up the device's UCB so that the device owns the controller permanently.

Drivers must be executing at driver's fork IPL when they invoke RELCHAN or call IOC\$RELCHAN.

Completing an I/O Request and Handling Timeouts

10.1 I/O Postprocessing

10.1.2.2 Saving Status, Count, and Device-Dependent Status

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

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

10.1.2.3 Returning Control to the Operating System

Finally, the driver fork process returns control to the system by invoking the REQCOM macro to complete the I/O request. REQCOM issues a JMP instruction to the VMS routine IOC\$REQCOM. IOC\$REQCOM locates the address of the I/O request packet (IRP) corresponding to the I/O operation in the device's UCB (UCB\$_IRP). It then writes the two longwords of completion status contained in R0 and R1 into the media field of the IRP (IRP\$_MEDIA and IRP\$_MEDIA+4).

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

If the error-logging bit is set in the device's UCB (UCB\$_ERLOGIP in UCB\$_STS), IOC\$REQCOM obtains the address of the error message buffer from the UCB (UCB\$_EMB). It then writes the following information into the error buffer:

- Final device status (UCB\$_DEVSTS)
- Final error count (UCB\$_ERTCNT)
- Maximum error retry count for the driver
- Two longwords of completion status (R0 and R1)

To release the error message buffer, IOC\$REQCOM calls ERL\$RELEASEMB. Section 11.3 describes error logging in more detail.

If any IRPs are waiting for driver processing, IOC\$REQCOM dequeues an IRP from the head of the queue of packets waiting for the device unit (UCB\$_IOQFL), and transfers control to IOC\$INITIATE. IOC\$INITIATE initiates execution of this I/O request in the driver's fork process, by activating the driver's start-I/O routine, as described in Section 4.2.1.

Otherwise, IOC\$REQCOM clears the unit-busy bit in the device's UCB status longword (UCB\$_BSY in UCB\$_STS) and transfers control to IOC\$RELCHAN to release the controller channel in case the driver failed to do so. IOC\$RELCHAN, in turn, returns control to the caller of the driver fork process (if the fork process issued the REQCOM macro). This is generally the VMS fork dispatcher. The fork dispatcher releases the fork lock, restores saved registers, and dismisses the fork IPL software interrupt with an REI instruction.

¹ R0 and R1 are the status values that VMS returns to the user process in the I/O status block specified in the original \$QIO system service.

Completing an I/O Request and Handling Timeouts

10.1 I/O Postprocessing

The remaining steps in processing the I/O request are performed by VMS I/O postprocessing. (See Section 4.3.1 for additional information.)

10.2 Timeout Handling Routines

VMS transfers control to the driver's timeout handling routine if a device unit does not request an interrupt within the time limit specified in the invocation of the wait-for-interrupt macro. Among its other activities, the VMS software timer interrupt service routine, having raised IPL from IPL\$_TIMERFORK to IPL\$_SYNCH, scans UCBs once every second to determine whether a device has timed out.

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

- 1 It obtains both the fork lock and the device lock associated with the device unit to synchronize access to its fork database and device database. It raises IPL to device IPL as a result of obtaining the device lock.
- 2 It raises IPL on the local processor to IPL\$_POWER to block local power failure servicing.
- 3 It disables expected interrupts and timeouts on the device by clearing bits in the status field of the device's UCB (UCB\$V_INT and UCB\$V_TIM in UCB\$L_STS).
- 4 It sets the device-timeout bit in the UCB status field (UCB\$V_TIMEOUT in UCB\$L_STS).
- 5 It lowers IPL to hardware device interrupt IPL (UCB\$B_DIPL).
- 6 It restores the saved R3 and R4 of the driver's fork process from the UCB fork block (UCB\$L_FR3 and UCB\$L_FR4).
- 7 It restores R5 (address of the UCB fork block).
- 8 It computes the address of the driver's timeout handling routine from the saved PC in the UCB fork block (UCB\$L_FPC).
- 9 It transfers control to the driver's timeout handling routine.

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

- R0 through R5 are saved on the stack.
- R5 contains the address of the UCB for the device that timed out.
- Only system address space may be accessed.
- The processor is running in kernel mode.
- The processor is running on the interrupt stack.
- The processor holds both fork lock and device lock.
- IPL is at hardware device interrupt level.

A timeout handling routine returns control to the software timer interrupt service routine by issuing an RSB instruction. The driver's fork process eventually regains control, with R3 and R4 restored from UCB\$L_FR3 and UCB\$L_FR4.

Completing an I/O Request and Handling Timeouts

10.2 Timeout Handling Routines

Certain timeout handling routines may find it useful to fork to execute low priority code or to accomplish certain tasks, such as the restarting of an I/O request (see Section 10.2.1). If a driver uses this method, its interrupt service routine should check the interrupt-expected bit (UCB\$V_INT) before handling the interrupt. The operating system clears this bit before it calls the driver's timeout handling routine. This allows the routine to determine whether device-timeout processing is in progress at fork IPL.

During recovery from a power failure, VMS forces a device timeout by altering the timeout field (UCB\$L_DUETIM) of a UCB if that device's UCB records that the unit is waiting for an interrupt or timeout (UCB\$V_INT and UCB\$V_TIM set in UCB\$L_STS). The timeout handling routine can perceive that recovery from a power failure is occurring by examining the power bit (UCB\$V_POWER in UCB\$L_STS) in the UCB.

A timeout handling routine usually performs one of three functions:

- It retries the I/O operation unless a retry count is exhausted.
- It aborts the I/O request, returning status (for instance, SS\$_TIMEOUT) in R0.
- It sends a message to an operator mailbox and waits for a subsequent interrupt or timeout.

10.2.1 Retrying an I/O Operation

Some devices might retry an I/O operation after a timeout. For example, a disk driver's timeout handling routine might take the following steps after a transfer timeout:

- 1 Invokes the FORK macro to lower IPL to fork level.
- 2 Releases any owned map registers, data path, and controller data channel.
- 3 Determines whether it is possible to retry the I/O operation.
- 4 Examines the error retry count (UCB\$B_ERTCNT) to determine whether it is possible to retry the I/O operation.

If the retry count is exhausted, the timeout handling routine sets the error code, performs a normal abort I/O cleanup operation, and issues the REQCOM macro to complete the I/O request.

If the retry count is not exhausted, the routine proceeds to the next step.

- 5 Examines the power bit (UCB\$V_POWER in UCB\$L_STS) to determine if it must take special steps before retrying the operation. For instance, the timeout handling routine should load the address of the IRP into R3 and reload the following fields of the IRP into the corresponding UCB fields, if they have been altered by partial processing of the I/O request:

```
IRP$L_BCNT  
IRP$W_BOFF  
IRP$L_SVAPTE
```

These actions set up an environment in which the transfer can be retried from the beginning.

- 6 Calls ERL\$DEVICTMO to log the device timeout if the driver supports error logging (see Section 6.2).

Completing an I/O Request and Handling Timeouts

10.2 Timeout Handling Routines

- 7 Decreases the error retry count (UCB\$B_ERTCNT).
- 8 Clears the UCB timeout bit (UCB\$V_TIMEOUT) in UCB\$L_STS.
- 9 Branches to the start-I/O routine to retry the operation.

10.2.2 Aborting an I/O Request

A driver's timeout handling routine aborts the I/O request when it exhausts its retry count or when, having read device registers, the driver determines that some fatal error condition has occurred such that there is no point in retrying the request. Similarly, the routine aborts a request if the device's cancel-I/O bit (UCB\$V_CANCEL in UCB\$L_STS) is set, signifying that a cancel-I/O request was made.

To abort an I/O request, a timeout handling routine performs the following sequence of steps:

- 1 Clears the device control and status register (CSR), if appropriate to the device and controller
- 2 Invokes the FORK macro to lower IPL to fork level
- 3 Releases any owned map registers, data path, and controller data channel
- 4 Loads the abort status code (SS\$_ABORT) into the low word of R0
- 5 Clears bits 16 through 31 in R0 to indicate that no data was transferred
- 6 Issues the REQCOM macro to complete the request

10.2.3 Sending a Message to the Operator

The following sequence describes a timeout handling routine that sends a message to the operator's mailbox and then goes back into a wait-for-interrupt or timeout state on the presumption that subsequent human intervention will make the device operational:

- 1 The timeout handling routine invokes the FORK macro to lower IPL to driver fork level.
- 2 It checks the cancel-I/O bit in the UCB status longword (UCB\$V_CANCEL in UCB\$L_STS).

If UCB\$V_CANCEL is set, the timeout handling routine can abort the request. However, if UCB\$V_CANCEL is clear, the timeout handling routine performs the following actions:

- a. Saves R3 and R4 on the stack.
- b. Loads an OPCOM message code, such as MSG\$_DEVOFFLIN, into R4. Note that the driver must invoke the message definition macro \$MSGDEF (located in SYS\$LIBRARY:STARLET.MLB) to use these message codes.
- c. Loads the address of the operator's mailbox (a pointer to which is located at SYS\$AR_OPRMBX) into R3.

Completing an I/O Request and Handling Timeouts

10.2 Timeout Handling Routines

- d. Calls a VMS routine to place the message in the operator's mailbox, as follows:

```
JSB G^EXE$SNDEVMSG
```

- e. Restores R3 and R4.
- f. Invokes the DEVICELock macro to raise IPL to device IPL and obtain the associated device lock.
- g. Issues a SETIPL macro to raise IPL\$_POWER and prevent power failure interrupts on the local processor.
- h. Invokes the WFIKpch macro to wait for another interrupt or timeout.

When the OPCOM process reads the message in its mailbox, it sends the requested message, in this case "device-offline," to all operator terminals enabled for that device class.

11 Other Driver Routines

Drivers normally contain initialization, cancel-I/O, error logging, and register dumping routines. The driver prologue table specifies the addresses of the unit and controller initialization routines.¹ The driver dispatch table (DDT) contains the addresses of the cancel-I/O, error logging, and register dumping routines. The type of device determines which of these routines are required in a driver.

Drivers more rarely require a driver unloading routine, cloned UCB routine, or unit delivery routine. VMS, however, provides a method for specifying these routines in the DPT or DDT. A brief discussion of the driver unloading routine appears in Section 15.2.3. Section 11.4 describes the functions of a cloned UCB routine. A description of the unit delivery routine appears in Section 15.4.2.

11.1 Initialization Routines

Most device controllers and device units require initialization both when the corresponding device driver is loaded and when the operating system is recovering from a power failure. At these times, the duty of initialization routines is to prepare controllers and device units for operation, according to their characteristics.

The 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 local processor while initialization is occurring; for this reason, initialization routines should only contain code that is absolutely needed at initialization time. Initialization routines should *not* explicitly lower IPL. The system calls initialization routines with a JSB instruction; the routines return by executing an RSB instruction.

11.1.1 Controller Initialization Routine

The duties of a controller initialization routines depend on the characteristics of the device. For example, a controller initialization routine for a card reader might enable interrupts from the device by setting the interrupt-enable bit in the device's control and status register (CSR). A disk's controller initialization routine, on the other hand, might enable interrupts and initialize all unit-status registers. A controller initialization routine can typically perform any of the following tasks:

- Determines if it is being called as a result of a power failure by examining the power bit (UCB\$_V_POWER in UCB\$_L_STS) in the UCB. A controller initialization routine may want to perform or avoid specific tasks when servicing a power failure (see Section 11.1.4).
- Clears error-status bits in device registers.

¹ A MASSBUS device driver must specify the address of its unit initialization routine in the driver dispatch table (using the **unitinit** argument to the DDTAB macro as discussed in Section 6.2). UNIBUS, Q22 bus, and generic VAXBI device drivers can specify the address in either the DPT or DDT.

Other Driver Routines

11.1 Initialization Routines

- Initiates a device operation, such as clearing a drive or acknowledging a disk pack.
- Enables controller interrupts.
- If the controller is dedicated to a single-unit device, such as a printer, fills in IDB\$L_OWNER and set the online bit (UCB\$V_ONLINE in UCB\$L_STS).
- Permanently allocates driver resources, such as
 - UNIBUS/Q22 bus map registers (see Section 12.2.2.2)
 - UNIBUS buffered data path (see Section 12.2.1.2)
- Allocates a buffer from nonpaged system dynamic memory.

Note that the permanent allocation of driver resources and the allocation of nonpaged pool require that the controller initialization routine fork to the driver's fork IPL. This action warrants careful coordination of the activities of the controller and unit initialization routines, both with each other and with the System Generation Utility (SYSGEN). See Section 11.1.5 for a discussion of forking in an initialization routine.

The controller initialization routine for a generic VAXBI device driver must initialize the device-specific aspects of the VAXBI device. Hardware initialization might include such activities as writing values to BIIC and device-specific registers, examining the results of the BIIC self test, mapping a node's window space, building data structures to control the device, and linking these structures into chains of similar data structures. (Section 14.4 extensively discusses the means by which a driver's controller initialization routine performs these tasks.)

At the time of a call to a controller initialization routine, the following registers contain the listed values:

| Register | Value |
|----------|---|
| R4 | Address of CSR |
| R5 | Address of IDB that describes the controller |
| R6 | Address of DDB associated with the controller |
| R8 | Address of CRB for the controller |

A controller initialization routine must preserve the contents of all registers except R0, R1, and R2.

11.1.2 Unit Initialization Routine

A unit initialization routine is useful for initializing device-dependent fields in the UCB. For example, a unit initialization routine for a disk can also specify disk-drive geometry (such as number of cylinders) in the UCB and wait for online units to spin up to speed. Unit initialization routines must set the online bit in the UCB (UCB\$V_ONLINE) to declare the unit to be on line.

A unit initialization routine can perform the same types of tasks as a controller initialization routine (see Section 11.1.1). Generally, the driver for a single-unit controller does not need a unit initialization routine.

Other Driver Routines

11.1 Initialization Routines

At the time of a call to a unit initialization routine, the registers contain the following values:

| Register | Value |
|----------|--|
| R3 | Address of primary CSR |
| R4 | Address of secondary CSR; R4 is equal to R3 if there is no secondary CSR |
| R5 | Address of the device's UCB |

A unit initialization routine must preserve the contents of all registers except R0, R1, and R2.

11.1.3 Initialization During Driver Loading

Prior to calling the initialization routines within a driver, VMS takes steps to initialize the appropriate I/O database structures and establish the appropriate links between these data structures and the driver. First, during system initialization, VMS creates an ADP for the device adapter. For generic VAXBI devices and MASSBUS devices, VMS creates an ADP, CRB, and IDB for the device at this time. Secondly, during driver loading, VMS performs some additional initialization. Finally, the driver's initialization routines are given an opportunity to initialize the device in a device-specific manner.

The extent of the initialization VMS performs during driver loading depends upon whether the I/O database is being created, and whether the driver is being loaded for the first time or is replacing a driver that was previously loaded.

The SYSGEN commands LOAD, AUTOCONFIGURE, and CONNECT add new drivers to the system configuration. The RELOAD command unloads an existing version of a driver and replaces it with a new one.

The LOAD command loads the driver into nonpaged system memory but does not call any driver-specific routines or execute any initialization requests specified in DPT_STORE macro invocations.

The AUTOCONFIGURE and CONNECT commands create and initialize I/O database structures associated with the device driver, call driver-specific initialization routines, and perform requests specified in DPT_STORE macro invocations. For each new device they add to the system, AUTOCONFIGURE and CONNECT perform the following steps:

- Create a UCB for the device. If this is the first occurrence of device and controller name, the commands create a DDB, CRB, and an IDB. (Because the CRB and IDB for a generic VAXBI device driver or MASSBUS device driver have already been created by the VMS adapter initialization routine, a CONNECT or AUTOCONFIGURE command for such a device never creates these structures.)
- Perform the initialization operations specified by the DPT_STORE macros within the initialization and reinitialization portions of the DPT.
- Relocate all addresses in the DDT and FDT to system virtual addresses.
- Call the controller initialization routine specified in the CRB, if it has created a CRB (or if CRB\$V_UNINIT is set in CRB\$B_MASK for a generic VAXBI device).

Other Driver Routines

11.1 Initialization Routines

- Call the unit initialization routine (if any) specified in the DDT. If no routine exists in the DDT, call the unit initialization routine (if any) specified in the CRB.

The AUTOCONFIGURE and CONNECT command operations raise IPL to IPL\$_POWER before calling the driver's initialization routines.

The RELOAD command replaces an existing driver with a new driver. The command loads the new driver's code into nonpaged system memory. Unlike the other SYSGEN commands for driver loading, RELOAD assumes that the data structures associated with the driver already exist, and thus updates the I/O database to reflect the modified code and its different location in system virtual address space. It performs the following functions:

- Calls the driver unloading routine in the old version of the driver, if one exists (as indicated in the **unload** argument of the DPTAB macro) and if bit DPT\$_V_NOUNLOAD in DPT\$_B_FLAGS is clear.

The driver unloading routine must return success status in R0 for SYSGEN to proceed with the following steps.

- Deallocates the memory occupied by the old version of the driver.
- Loads the new version of the driver.
- Executes requests specified by DPT_STORE macro invocations in only the reinitialization section of the DPT in the new driver.
- Relocates all addresses in the FDT and DDT to system virtual addresses.
- Calls the controller initialization routine.

Chapter 15 contains detailed descriptions of all SYSGEN commands related to device drivers.

11.1.4 Initialization During Recovery from a Power Failure

During recovery from a power failure, the operating system locates every UCB in the I/O database, by following the chain of pointers to all DDBs in the system (starting at IOC\$_GL_DEVLIST and chained by DDB\$_LINK) and the chain of pointers to all UCBs of the same device and controller type (starting at DDB\$_UCB and chained by UCB\$_LINK). For each UCB it finds, VMS performs the following procedure:

- 1 It locates the CRB associated with the UCB (UCB\$_CRB) and determines whether a controller initialization routine exists for the device's controller by examining CRB\$_INTD+VEC\$_INITIAL. If an invocation of the DPT_STORE macro loaded the address of a controller initialization routine into this field, VMS calls that routine.
- 2 It determines whether a unit initialization routine exists for the particular device unit by examining the unit initialization field of the DDT (DDT\$_UNITINIT). If the field does not contain an address, the system checks the CRB (CRB\$_INTD+VEC\$_UNITINIT).²

² MASSBUS drivers store unit initialization routines addresses only in the DDT.

Other Driver Routines

11.1 Initialization Routines

If either the CRB or the DDT contains a nonzero address for such a routine, the system calls the routine to initialize the device unit. The system calls only one routine; if the DDT contains an address, the address in the CRB is ignored.

When called to service a power failure, driver initialization routines must adhere to the following rules:

- They cannot acquire any spin locks. Controller and unit initialization routines are called at IPL 31 during power failure recovery to reinitialize I/O devices before the processors are allowed to proceed with execution at lower IPLs. Because processors may have been holding spin locks at the time of the power failure, they will not be able to release them until after they resume execution. As a result, spin locks are not available to controller and unit initialization routines.
- They cannot perform any operation that requires the intervention of other processors in a VMS multiprocessing system.

A driver initialization routine can determine if it is being called as a result of a power failure by examining the power bit (UCB\$V_POWER in UCB\$L_STS) in the UCB.

11.1.5 Forking from a Driver Initialization Routine

If a driver initialization routine must fork to perform a thread of code that must synchronize with code or a structure synchronized at a lower IPL, it must take special care to avoid breaking that synchronization.

First of all, because SYSGEN, under normal circumstances, immediately calls a driver's unit initialization routine at IPL\$_POWER after its controller initialization completes, the unit initialization routine must be prepared for the instance of a controller initialization routine that forks. Such a unit initialization routine would complete before the fork thread of the controller initialization routine resumed.

A fork thread in a unit initialization routine (or a controller initialization routine in a driver without a unit initialization routine) must otherwise take the following precautions to avoid breaking synchronization:

- Use either the CRB fork block, or a fork block defined in a device-specific extension to the UCB. The separate fork block prevents a conflict with the use of the normal UCB fork block by the IOFORK routine. If you are using a separate UCB fork block, you must not attempt to allocate the fork block from paged pool.
- You should use a semaphore bit to protect against multiple forking. Remember that the unit initialization routine may be called repeatedly in the case of power failures. If the semaphore shows that a fork is in progress, then exit without attempting to fork. Access the semaphore bit using interlocked instructions (for example, BBSSI or BBCCI).
- Invoke EXE\$FORK with R5 pointing to the alternate fork block. Restore the original value of R5 once the fork process is active.

Other Driver Routines

11.1 Initialization Routines

- Restore all registers on exit. Because EXE\$FORK removes the caller's address from the stack and returns to the caller's caller, the unit initialization routine must set up a dummy caller's caller routine to restore registers destroyed by EXE\$FORK.

11.2 Cancel-I/O Routine

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

- When a process issues a Cancel-I/O-on-Channel system service (\$CANCEL)
- When a process deallocates a device, causing the device reference count (UCB\$W_REFC) to become zero (that is, no process I/O channels are assigned to the device)
- When a process deassigns a channel from a device, using the \$DASSGN system service³
- When VMS 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 VMS routine EXE\$CANCEL locates the UCB for the device associated with a process I/O channel from a pointer in the CCB, as follows:

channel index number → CCB → UCB

EXE\$CANCEL performs the following steps:

- 1 Obtains the fork lock associated with the driver, thus raising IPL to fork IPL.
- 2 Removes from the device's pending-I/O queue all IRPs associated with the process and that channel.
- 3 For a buffered-I/O read operation, clears the buffered-read function bit (IRP\$V_FUNC) in IRP\$W_STS.
- 4 Sets the status code SS\$_CANCEL in IRP\$L_MEDIA.
- 5 Inserts the IRPs removed from the pending-I/O queue into the local processor's I/O postprocessing queue.
- 6 Requests a software interrupt from the local processor at IPL\$_IOPOST.
- 7 Calls the cancel-I/O routine specified in the DDT of the associated device driver (argument **cancel** to the DDTAB macro). EXE\$CANCEL locates the routine using the following chain of pointers:

UCB → DDT → cancel-I/O routine

³ Note that if the call to \$DASSGN deassigns the last channel to the device, the device driver's cancel-I/O routine is called a second time. Channel deassignment and last channel deassignment are both potentially significant events for certain devices. The former means, in effect, that a user has finished with a device and the latter means that all users are finished with a device. The reason code for both events is CAN\$_DASSGN. However, a driver's cancel-I/O routine can distinguish between the two cases by examining UCB\$W_REFC.

Other Driver Routines

11.2 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.

11.2.1 Context of a Cancel-I/O Routine

When EXE\$CANCEL calls the cancel-I/O routine, the local processor is at driver fork IPL holding the associated fork lock. As a result, the cancel-I/O routine can read and modify the device's UCB. Registers at the time of the call contain the following values:

| Register | Value |
|----------|--|
| R2 | Channel index number. |
| R3 | Address of current IRP. |
| R4 | Address of process control block (PCB) of process for which the \$CANCEL system service is being performed. |
| R5 | Address of device's UCB. |
| R8 | Reason for call to cancel the I/O request. Codes that signify the reasons for cancellation are defined by the \$CANDEF macro. Possible values for R8 include CAN\$_CANCEL Called by \$CANCEL system service CAN\$_DASSGN Called by \$DASSGN or \$DALLOC system service |

If a cancel-I/O routine uses registers other than R0 through R3, it must save the registers and restore them before exiting.

Device drivers might want to base their cancel-I/O operation on whether the cancel-I/O request is the result of a channel deassignment (CAN\$_DASSGN). For example, the terminal driver cancels out-of-band AST requests only if the call to its cancel-I/O routine results from a Deassign-I/O-Channel (\$DASSGN) system service call.

11.2.2 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 VMS routine IOC\$RETURN into the appropriate position in the DDT. The routine IOC\$RETURN executes a single RSB instruction.

Other Driver Routines

11.2 Cancel-I/O Routine

11.2.3 Device-Independent Cancel-I/O Routine

Drivers can specify the VMS routine IOC\$CANCELIO as the value of the **cancel** argument in the DDTAB macro invocation. IOC\$CANCELIO cancels I/O to a device in the following device-independent manner:

- 1 It confirms that the device is busy by examining the device-busy bit in the UCB status longword (UCB\$V_BSY in UCB\$L_STS).
- 2 It locates the process-identification field in the IRP currently being processed on the device by using the following chain of pointers:

UCB → IRP → process identification field

IOC\$CANCELIO confirms that the field (IRP\$L_PID) contains the same value as the corresponding field in the PCB (PCB\$L_PID).

- 3 It confirms that the specified channel-index number is the same as the value stored in the IRP's channel-index field (IRP\$W_CHAN).
- 4 It sets the cancel-I/O bit in the UCB status longword (UCB\$V_CANCEL in UCB\$L_STS). Other driver routines, such as the timeout handling routine, check the cancel-I/O bit to determine whether to retry the I/O operation or abort it. (See Section 10.2.2 for additional information.)

11.2.4 Device-Dependent Cancel-I/O Routine

Drivers that include their own cancel-I/O routines must perform the first three steps of IOC\$CANCELIO listed in Section 11.2.3 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-I/O routine can proceed in a device-specific manner. For instance, a cancel-I/O routine may perform the following tasks:

- Clear UCB\$V_INT and UCB\$V_TIM in the UCB status longword (UCB\$L_STS)
- Release any owned map registers, data path, and controller data channel
- Load a status code (SS\$_CANCEL, for instance) into the low word of R0
- Load other status information into the high word of R0 and the longword of R1
- Issue the REQCOM macro to complete the request

11.3 Error Logging Routines

A driver that supports error logging must satisfy the following prerequisites:

- It must invoke the data structure definition macro \$EMBDEF (located in SYS\$LIBRARY:LIB.MLB).
- It must use the local disk extension or local tape extension of the UCB. These extensions include error-log extension. (See Section A.14 for additional information.)

Other Driver Routines

11.3 Error Logging Routines

- It must provide a means whereby error logging can be enabled for the device. For instance, it can use the DPT_STORE macro to set the device characteristic DEV\$V_ELG in UCB\$L_DEVCHAR or it can support an IO\$_SETCHAR function that sets this bit.
- It must ensure that the size of the error log buffer, as specified in DDT\$W_ERRORBUF, is large enough to accommodate EMB\$L_DV_REGSAV+4, plus one longword for each register to be dumped. It must specify this value in the **erlgbf** argument to the DDTAB macro.
- It should include a register dumping routine, specifying its address in the **regdmp** argument of the DDTAB macro.
- It must complete the servicing of the I/O request by invoking the REQCOM macro. (Routines, like ERL\$DEVICEATTN, that log errors that are not associated with the current I/O request skip this step.) IOC\$REQCOM takes steps to complete the error logging initiated by a call to an error logging routine.

11.3.1 Error Logging Routines Supplied by VMS

The VMS operating system provides the following routines that drivers can call to allocate and fill an error message buffer after a device error or timeout occurs:

| Routine | Function |
|-----------------|--|
| ERL\$DEVICERR | Logs an error associated with the I/O request in progress |
| ERL\$DEVICTMO | Logs a timeout associated with the I/O request in progress |
| ERL\$DEVICEATTN | Logs an error not associated with an I/O request |

These routines are described in full in Appendix C, but they all perform similar functions, as follows:

- Increment UCB\$W_ERRCNT to record a device error. If the error-log-in-progress bit (UCB\$V_ERLOGIP in UCB\$L_STS) is set, the routine returns control to its caller (ERL\$DEVICERR and ERL\$DEVICTMO only).
- Allocate from the current error log allocation buffer an error message buffer of the length specified in the device's DDT (in argument **erlgbf** to the DDTAB macro).
- Initialize the buffer with the current system time, error log sequence number, and error type code. These routines use the following error type codes:

| Routine | Error Code |
|-----------------|-----------------------------|
| ERL\$DEVICERR | Device error (EMB\$_DE) |
| ERL\$DEVICTMO | Device timeout (EMB\$_DT) |
| ERL\$DEVICEATTN | Device attention (EMB\$_DA) |

- Place the address of the error message buffer in UCB\$L_EMB.
- Set UCB\$V_ERLOGIP in UCB\$L_STS.

Other Driver Routines

11.3 Error Logging Routines

- Load into R0 the address of the location in the buffer in which the contents of the device registers are to be stored.
- Call the driver's register dumping routine.

11.3.2 Register Dumping Routine

A driver that supports error logging or diagnostics specifies the address of a register dumping routine in the **regdmp** argument to the DDTAB macro.

When an error logging routine passes control to the driver's register dumping routine, the following registers contain the listed values:

| Register | Value |
|----------|--|
| R0 | Address of buffer into which a register dumping routine copies the contents of device registers |
| R4 | Address of device's CSR (if the driver invoked the WFIKPC macro to wait for an interrupt or timeout) |
| R5 | Address of UCB |

The register dumping routine preserves the contents of all registers except R0 through R2. If it uses the stack, the register dumping routine must restore the stack before passing control to another routine, waiting for an interrupt, or returning control to its caller.

A register dumping routine uses the following procedure to fill the indicated buffer:

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

The source of these register values depends upon the nature of the driver. If the driver has established a UCB extension, its interrupt service routine can copy to it the values of critical device registers. In this case, the register dumping routine may contain instructions similar to the following:

```
MOVL  UCB$L_TD_STATUS(R5), (R0)+
```

Alternatively, the register dumping routine can obtain device register values directly from I/O address space, offsetting from the address of the CSR as follows:

```
MOVZWL TD_STATUS(R4), (R0)+
```

Note that this latter method is not truly accurate in that, at the time a register dumping routine runs, all or some device registers may have been modified during the servicing of device interrupts unrelated to the error.

When a driver fork process invokes the system routine IOC\$DIAGBUFILL, as described in Appendix C, the routine transfers control to the register dumping routine with the address of the diagnostic buffer in R0, the address of the device's CSR in R4, and the address of the UCB in R5.

Other Driver Routines

11.3 Error Logging Routines

11.3.3 Interpreting Error Log Entries

See the *Guide to Maintaining a VMS System* and the *VMS Error Log Utility Manual* for help with producing and reading error log files.

11.4 Cloned UCB Routine

EXE\$ASSIGN calls the driver's cloned UCB routine when an Assign I/O Channel system service request (\$ASSIGN) specifies a template device (that is, bit UCB\$V_TEMPLATE in UCB\$_STS is set). EXE\$ASSIGN does not assign the channel to the template device itself. Rather, it creates a copy of the template device's UCB and ORB, initializing and clearing certain fields as appropriate.

A cloned UCB routine receives control at IPL\$_ASTDEL in kernel mode with process context available, holding the I/O database mutex (IOC\$_GL_MUTEX).

Only drivers for network devices or template devices, such as mailboxes, include a cloned UCB routine. A driver specifies the address of a cloned UCB routine in the **cloneducb** argument of the DDTAB macro.

The driver's cloned UCB routine verifies the contents of fields in the UCB and ORB and completes their initialization. When a cloned UCB routine is called, the following locations contain the listed values:

| Location | Contents |
|------------------|--|
| R0 | SS\$_NORMAL |
| R2 | Address of cloned UCB |
| R3 | Address of DDT |
| R4 | Address of current PCB |
| R5 | Address of template UCB |
| UCB\$_FQFL(R2) | Address of UCB\$_FQFL(R2) |
| UCB\$_FQBL(R2) | Address of UCB\$_FQFL(R2) |
| UCB\$_FPC(R2) | 0 |
| UCB\$_FR3(R2) | 0 |
| UCB\$_FR4(R2) | 0 |
| UCB\$_BUFQUO(R2) | 0 |
| UCB\$_ORB(R2) | Address of cloned ORB |
| UCB\$_LINK(R2) | Address of next UCB in DDB chain |
| UCB\$_IOQFL(R2) | Address of UCB\$_IOQFL(R2) |
| UCB\$_IOQBL(R2) | Address of UCB\$_IOQFL(R2) |
| UCB\$_UNIT(R2) | Device unit number (minimum UCB\$_UNIT_SEED(R5)+1) |
| UCB\$_CHARGE(R2) | Mailbox byte quota charge (UCB\$_SIZE) |
| UCB\$_REFC(R2) | 0 |
| UCB\$_STS(R2) | UCB\$_DELETEUCB set, UCB\$_ONLINE set |

Other Driver Routines

11.4 Cloned UCB Routine

| Location | Contents |
|-------------------------------------|--|
| UCB\$W_DEVSTS(R2) | UCB\$V_DELMBX set if DEV\$V_MBX is set in UCB\$L_DEVCHAR(R2) |
| UCB\$L_OPCNT(R2) | 0 |
| UCB\$L_SVAPTE(R2) | 0 |
| UCB\$W_BOFF(R2) | 0 |
| UCB\$W_BCNT(R2) | 0 |
| UCB\$L_ORB(R2) | Address of cloned ORB |
| ORB\$L_OWNER of template ORB | UIC of current process |
| ORB\$L_ACL_MUTEX of template ORB | FFFF ₁₆ |
| ORB\$B_FLAGS of template ORB | ORB\$V_PROT_16 set |
| ORB\$W_PROT of template ORB | 0 |
| ORB\$L_ACL_COUNT of template ORB | 0 |
| ORB\$L_ACL_DESC of template ORB | 0 |
| ORB\$R_MIN_CLASS of template ORB | 0 in first longword |

A cloned UCB routine must preserve the contents of R2. It issues an RSB instruction to return control to EXE\$ASSIGN. If the routine returns error status in R0, EXE\$ASSIGN undoes the process of UCB cloning and completes with the failure status in R0.

Part III Bus Specific Considerations and Advanced Topics

12 UNIBUS and Q22 Bus Device Support

This chapter provides information specific to the creation of drivers for devices attached to the UNIBUS or Q22 bus. VMS provides extensive support for UNIBUS/Q22 bus drivers, including many system routines and macros that drivers can use to accomplish a multiblock transfer to a DMA device by means of UNIBUS adapter/Q22 bus interface resources. Section 12.1 explains the functions of the UNIBUS adapter and Q22 bus interface, describing these resources in detail. Section 12.2 provides a step-by-step account of how UNIBUS/Q22 bus device drivers can use the facilities of VMS to accomplish DMA transfers.

Although the general mechanism of device interrupt dispatching, as defined by the VAX architecture and briefly described on Chapter 9, is the same for all VAX processing systems and I/O subsystems, certain implementation details differ. In that regard, Section 12.3 describes the means by which VAX hardware and the VMS operating system deliver a UNIBUS or Q22 bus device's interrupt to its driver's interrupt service routine.

12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

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

- Arbitrates interrupts from UNIBUS devices according to their priority
- Delivers interrupts from UNIBUS devices to the processor
- Allows drivers to gain access to UNIBUS device registers using system virtual addresses
- Translates 18-bit UNIBUS addresses to physical addresses in main memory
- Provides a data-transfer path to randomly ordered physical addresses in main memory
- Provides buffered data transfer paths to consecutively increasing UNIBUS addresses, thus optimizing CPU-to-UNIBUS data transfers
- Permits byte-aligned buffers for UNIBUS devices requiring word-aligned buffer addresses

The Q22 bus closely resembles the UNIBUS. For MicroVAX 3600-series, MicroVAX II, or MicroVAX I devices attached to the Q22 bus, special processor logic implements a Q22 bus interface that similarly allows drivers access to device registers and manages device interrupts. Additional logic in the MicroVAX 3600-series processor or MicroVAX II processor establishes a scatter-gather map that translates 22-bit Q22 bus addresses to physical addresses. However, MicroVAX 3600-series, MicroVAX II, and MicroVAX I systems do not implement buffered data paths. (Table 12-1 compares the UNIBUS and Q22 bus I/O subsystems of the various VAX and MicroVAX processing systems.)

UNIBUS and Q22 Bus Device Support

12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

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

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

On the other hand, the differences between the functions of the UNIBUS adapter and the Q22 bus interface of the MicroVAX 3600-series, MicroVAX II, and MicroVAX I systems are significant to those drivers that manage DMA device operations.

Section 12.2 describes the means by which device drivers set up DMA transfers, according to any of these interfaces. If a DMA driver that drives similar devices on various VAX systems must secure some measure of machine independence, it can include some run-time conditional code that branches to appropriate routines in the driver that accomplish the machine-dependent work. See the description of the ADPDISP macro in Appendix B and the sample drivers that appear in Appendixes E and F for guidance.

This section discusses the functions of the UNIBUS adapter and the Q22 bus, as follows:

- The discussion of reading and writing device registers in Section 12.1.1 applies to UNIBUS, MicroVAX 3600-series, MicroVAX II, and MicroVAX I drivers.
- The description of mapping I/O bus addresses in Section 12.1.2 pertains only to UNIBUS, MicroVAX 3600-series, and MicroVAX II DMA drivers.
- The description of buffering data transfers in Section 12.1.3 relates mainly to UNIBUS drivers, although Section 12.1.3.1 contains information relevant to MicroVAX 3600-series, MicroVAX II, and MicroVAX I drivers as well.

Table 12-1 Features of the UNIBUS Adapters/Q22 Bus Interfaces of VAX Systems

| System | Adapter | Memory References (Physical Address) | Direct Data Path | Buffered Data Paths | Map Registers | Interrupt Dispatcher |
|--|---------|--------------------------------------|------------------------------|--|---------------|----------------------|
| VAX-11/780 VAX-11/785 VAX 8600 VAX 8650 VAX 8670 | UBA | 30-bit (via SBI) | 1, no byte-aligned transfers | 15, 8-byte buffer, byte-aligned transfers, LWAE, ³ prefetch | 496 | Non-direct-vector |

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

UNIBUS and Q22 Bus Device Support

12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

Table 12–1 (Cont.) Features of the UNIBUS Adapters/Q22 Bus Interfaces of VAX Systems

| System | Adapter | Memory References (Physical Address) | Direct Data Path | Buffered Data Paths | Map Registers | Interrupt Dispatcher |
|---|---------|--------------------------------------|---|---|------------------|----------------------|
| VAX–11/750 | UBI | 24-bit (via CMI) | 1, byte-aligned transfers | 3, 4-byte buffer, ² byte-aligned transfers, LWAE, ³ no prefetch | 512 ⁴ | Direct-vector |
| VAX–11/730 VAX–11/725 | UBA | 24-bit | 1, byte-aligned transfers | None | 512 ⁴ | Direct-vector |
| VAX 8200 VAX 8250 VAX 8300 VAX 8350 VAX 8530 VAX 8550 VAX 8700 VAX 8800 VAX 8830 VAX 8850 VAX 6200 series | DWBUA | 30-bit (via VAXBI) | 1, byte-aligned transfers | 5, 8-byte buffer, byte-aligned transfers, LWAE, ³ no prefetch | 512 ⁴ | Direct-vector |
| MicroVAX 3600 series | — | 29-bit | 1, no restrictions on data alignment ¹ | None | 8192 | Direct-vector |
| MicroVAX II | — | 24-bit | 1, no restrictions on data alignment ¹ | None | 8192 | Direct-vector |
| MicroVAX I | — | 22-bit | 1, no restrictions on data alignment ¹ | None | None | Direct-vector |

¹The MicroVAX 3600-series, MicroVAX II, and MicroVAX I implementations of the Q22 bus provide no byte-offset register; so, on Q22 bus devices that are only capable of word-aligned transfers, only word-aligned transfers are possible.

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

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

⁴The VMS operating system makes available only 496 of these map registers.

UNIBUS and Q22 Bus Device Support

12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

12.1.1 Reading and Writing Device Registers

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

Because the VMS operating system maps this I/O space into virtual address space, a device driver can treat the addresses of device registers as identical to all other virtual addresses. The driver can read and write data to the device's register as though the device's register were a location in memory. The driver must use instructions within the restrictions described in Section 5.2.

Before a driver for a device that shares a controller can gain access to a device's registers, it must first obtain a controller channel, as described in Sections 3.4.1 and 8.3.1.

12.1.2 Map Registers

DMA devices read and write data from and to memory locations using 18-bit UNIBUS addresses or, for the MicroVAX 3600 series, MicroVAX II, and MicroVAX I, 22-bit Q22 bus addresses.

A driver that performs multiblock DMA transfers for a UNIBUS device or Q22 bus device must set up any mapping or buffering mechanisms required by the system's I/O interface. For UNIBUS DMA drivers, this involves setting up sufficient map registers and, perhaps, a buffered data path prior to the transfer. MicroVAX 3600-series and MicroVAX II DMA drivers, likewise, must allocate and fill a set of map registers. By contrast, MicroVAX I DMA drivers—because the MicroVAX I has no scatter-gather map—cannot map the many and scattered pages of a multiblock DMA transfer to a contiguous set of addresses in the I/O adapter's address space. As a result, when it is loaded into the system, a MicroVAX I DMA driver must reserve enough physically contiguous memory to accommodate its largest possible DMA transfer (see Section 12.2.8).

For UNIBUS devices and MicroVAX II/MicroVAX 3600-series devices, the UNIBUS adapter and the Q22 bus interface translate the bus addresses into main memory addresses, thus allowing the operating system, I/O drivers, and UNIBUS devices to access the same physical address space. DMA devices connected to either a UNIBUS, MicroVAX 3600-series Q22 bus, or MicroVAX II Q22 bus can access a block of memory indirectly by means of the scatter-gather map supplied by the UNIBUS adapter or MicroVAX processor, respectively. The map registers provided allow the device to access scattered, physical memory addresses as contiguous, physical addresses in I/O space.

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

UNIBUS and Q22 Bus Device Support

12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

Consider a buffer, for example, that consists of virtual pages 400, 401, 402, and 403, which are physical pages 1003, 204, 1190, and 240, respectively. For a UNIBUS or Q22 bus device to access this buffer, the driver requests four map registers, then places the physical addresses of these pages in the map registers. A field in each map register identifies the page-frame number corresponding to the UNIBUS space or Q22 bus space address that the map register represents (see Figure 12-1).

Assume the driver has allocated four map registers, 127 through 130. The driver loads them as follows:

| Map Register | Contents |
|--------------|----------|
| 127 | 1003 |
| 128 | 204 |
| 129 | 1190 |
| 130 | 240 |

Note that the VMS routine the driver calls to allocate map registers automatically allocates an additional map register (register 131 in this case). The map register loading routine clears this register in order to prevent a runaway DMA transfer.

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

Generally, a map register exists for each page of I/O space. Because the UNIBUS address space consists of 256K of memory, minus the 8KB reserved for device control registers, 496 map registers are available for UNIBUS DMA transfers. MicroVAX 3600-series and MicroVAX II DMA devices can use up to 8192 of the map registers that correspond to the 4MB of Q22 bus I/O space.

Drivers call VMS routines to fill as many map registers with valid page-frame numbers as needed for a DMA transfer. The DMA device puts an address on the I/O bus. The UNIBUS adapter or Q22 bus interface receives the address and translates it using the following information (see Figures 12-2 and 12-3):

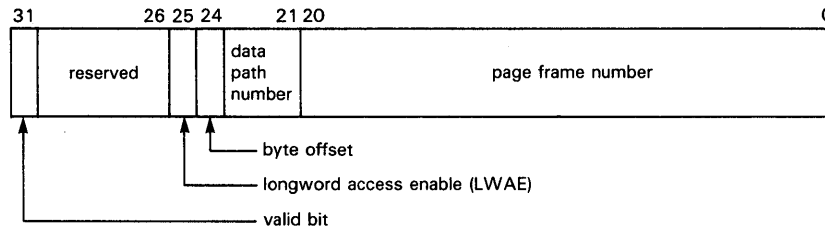
- In *UNIBUS addresses*, the 9-bit UNIBUS page address field (bits 9 through 17 of the UNIBUS address) identifies the UNIBUS adapter map register.
In *Q22 bus addresses*, the 13-bit Q22 bus page address field (bits 9 through 22 of the Q22 bus address) identifies the MicroVAX II map register.
- The page-frame-number (PFN) field in the map register specifies the high-order bits of the physical address. (The PFN field is 15 bits long for the MicroVAX II, VAX-11/750, and VAX-11/730; 20 bits long for the MicroVAX 3600-series systems, and 21 bits long for other VAX systems.)

UNIBUS and Q22 Bus Device Support

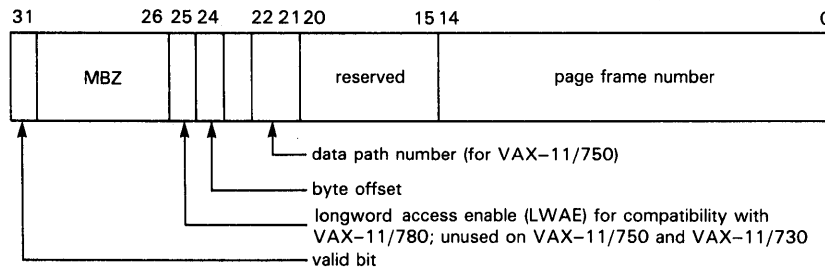
12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

Figure 12–1 UNIBUS and Q22 Bus Map Registers

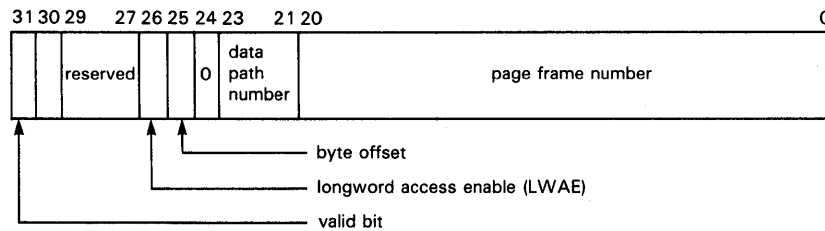
VAX–11/780, VAX–11/785, and VAX 8600/8650/8670



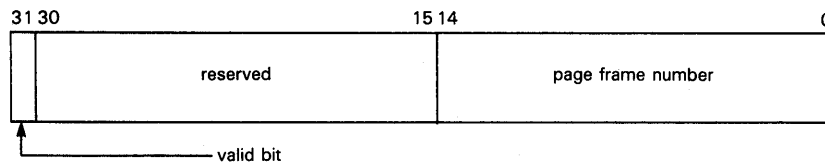
VAX–11/750, VAX–11/730, and VAX–11/725



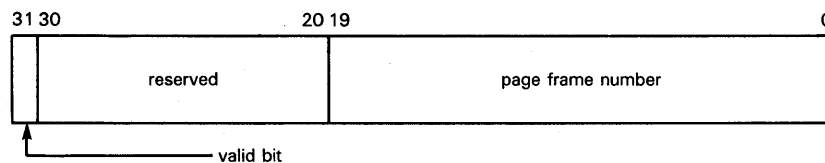
VAX 8200/8250/8300/8350, VAX 8530/8550/8700/8800/8830/8850, and VAX 6200 Series



MicroVAX II



MicroVAX 3600 Series



ZK-4842-85

UNIBUS and Q22 Bus Device Support

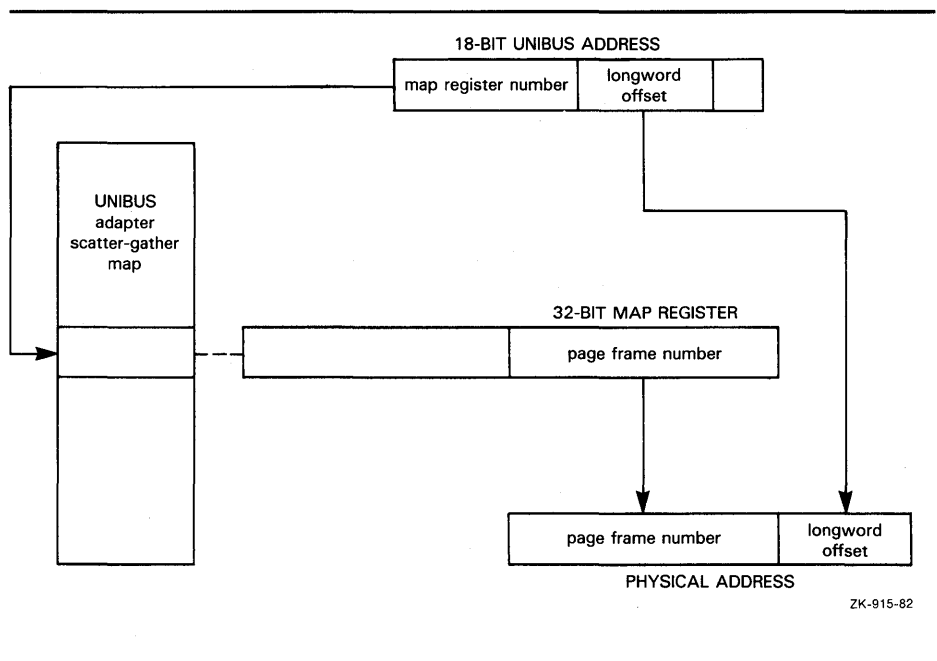
12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

- From *UNIBUS addresses*, bits 2 through 8 map to bits 0 through 6 of the physical address.¹ The resulting physical address locates the longword that is the target of the transfer.

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

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

Figure 12–2 Mapping a UNIBUS Address to a Physical Address

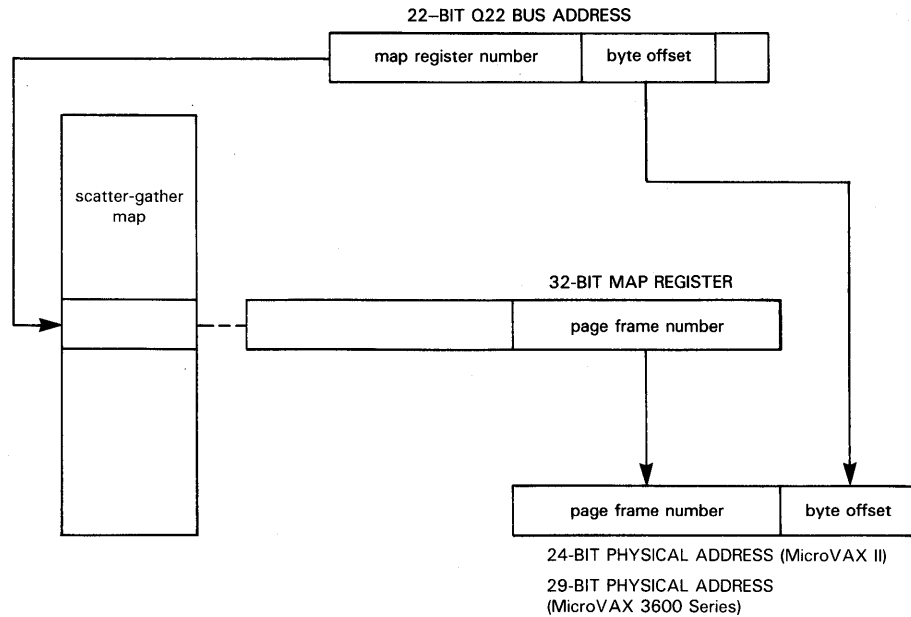


¹ The disposition of the lowest two bits of the UNIBUS address depends on the VAX system. For instance, the VAX-11/780 uses them to construct a byte-selection mask and function to be transmitted across UNIBUS lines that modify the I/O transaction.

UNIBUS and Q22 Bus Device Support

12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

Figure 12-3 Mapping a Q22 Bus Address to a Physical Address



ZK-4841-85

12.1.3 UNIBUS Adapter Data Transfer Paths

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

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

When a UNIBUS device begins a DMA transfer by placing an address on the UNIBUS, the UNIBUS adapter not only performs address mapping but also provides the number of the data path to be used for the transfer (see Figure 12-1). Each UNIBUS adapter map register contains a field that describes the data path. Data path 0 is the direct data path; the other data paths are the buffered data paths. (The UNIBUS data path registers of the various VAX systems are pictured in Figure 12-4.)

The following sequence describes a UNIBUS-device DMA transfer.

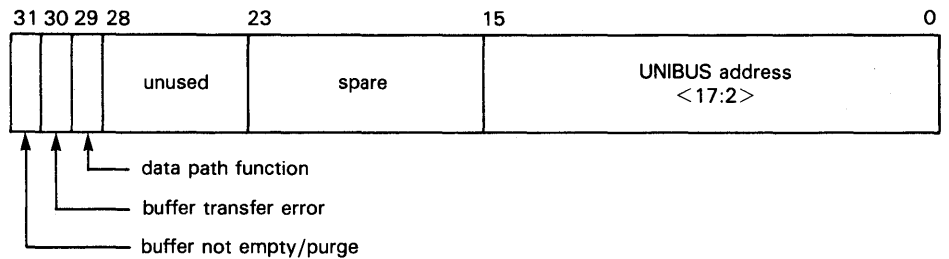
- 1 The UNIBUS device puts an address on the UNIBUS.
- 2 The UNIBUS adapter locates the UNIBUS adapter map register that corresponds to the UNIBUS address.

UNIBUS and Q22 Bus Device Support

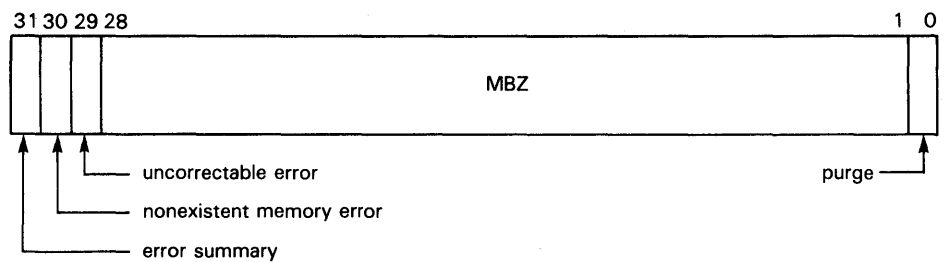
12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

Figure 12-4 UNIBUS Data Path Registers

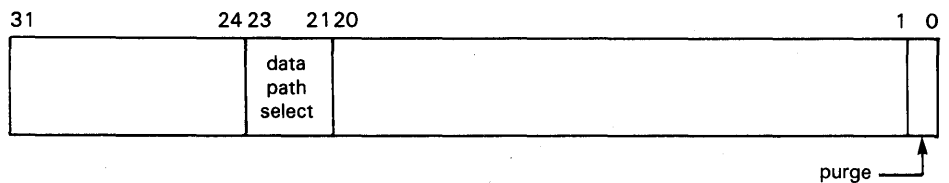
VAX-11/780, VAX 8600, VAX 8650, and VAX 8670



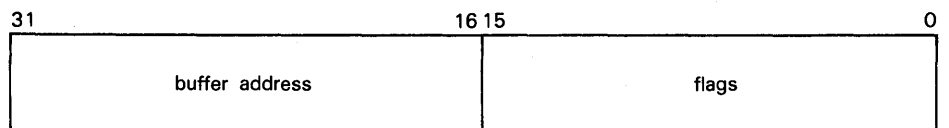
VAX-11/750



VAX 8200/8250/8300/8350, VAX 8530/8550/8700/8800/8830/8850, and VAX 6200 Series
DATA PATH CONTROL/STATUS REGISTER



ADDRESS/STATUS REGISTER



ZK-4843-85

- 3 The UNIBUS adapter verifies that the map register has the map-register valid bit set.
- 4 The UNIBUS adapter maps the UNIBUS address to a physical address.
- 5 The UNIBUS adapter extracts the number of the data path to be used for the transfer from the map register.
- 6 The UNIBUS adapter translates the UNIBUS function to a backplane interconnect function by reading the UNIBUS control lines.

UNIBUS and Q22 Bus Device Support

12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

- 7 Based on the UNIBUS function indicated by the UNIBUS control lines, (DATI, DATIP, DATO, or DATOB), the UNIBUS adapter starts appropriate UNIBUS and backplane interconnect operations to transfer data between the UNIBUS device and memory.

12.1.3.1 Direct Data Path

Since the direct data path performs a backplane interconnect transfer for every I/O bus transfer, it can be used by more than one UNIBUS or Q22 bus device at a time. The UNIBUS adapter or Q22 bus interface arbitrates among devices that wish to use the direct data path simultaneously. The device driver is unaffected by this arbitration.

The direct data path is less efficient than a buffered data path because each I/O bus transfer cycle corresponds to a backplane interconnect cycle. One word or byte is transferred for each backplane interconnect cycle. On some hardware configurations, the direct data path is unable to transfer a word of data to an odd-numbered physical address. Therefore, an FDT routine for a DMA device that uses the direct data path should check that the specified buffer is on a word boundary.²

The Q22 bus systems only employ a direct data path. A UNIBUS device driver may choose to use a direct data path rather than a buffered data path to perform the following functions:

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

The direct data path is the simplest data path to program. Since the direct data path can be shared simultaneously by any number of I/O transfers, the device driver does not need to call a VMS routine to allocate the data path. Instead, the driver performs the following actions:

- 1 Uses the REQMPR macro to allocate a set of map registers (or the REQALT macro to allocate a set of Q22 bus alternate map registers (registers 496 to 8191)).
- 2 Uses the LOADUBA macro (or LOADALT macro) to load the map registers with physical address map data and, for UNIBUS devices, the number of the direct data path (0). The VMS routines called in the expansion of these macros (IOC\$LOADUBAMAP and IOC\$LOADALTMAP respectively) also set the valid bit in every map register except the last, which remains invalid to prevent a runaway transfer.
- 3 Loads the starting address of the transfer in a device register.
- 4 Loads the transfer byte or word count in a device register.
- 5 Sets bits in the device CSR to initiate the transfer.

² The MicroVAX 3600-series, MicroVAX II, and MicroVAX I implementations of the Q22 bus provide no byte-offset register. As a result, for Q22 bus devices that are only capable of word-aligned transfers, only word-aligned transfers are possible.

UNIBUS and Q22 Bus Device Support

12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

12.1.3.2 Buffered Data Paths

When a buffered data path is used, the UNIBUS adapter transfers data much more efficiently between the UNIBUS and the backplane interconnect than when a direct data path is used. It accomplishes this by decoupling the UNIBUS transfer from the backplane interconnect transfer. The buffered data path allows the UNIBUS adapter to read or write multiple words of data in a transfer, and buffer the unrequested portions of the data in a UNIBUS adapter buffer. Thus, several UNIBUS read functions can be accommodated with a single backplane interconnect transfer.

Q22 bus systems do not employ buffered data paths. The writer of a UNIBUS device driver may choose to use a buffered data path rather than a direct data path to perform the following functions:

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

A single buffered data path cannot be assigned to more than one active transfer at a time. When a driver fork process is preparing to transfer data to or from a UNIBUS device on a buffered data path, it performs a sequence of steps similar to those performed by a driver that uses the direct data path, with the exception that it uses a macro that calls a VMS routine that allocates a free buffered data path. The following are among the actions of the driver fork process:

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

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

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

UNIBUS and Q22 Bus Device Support

12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

- To reference 32-bit data at random, longword-aligned locations in physical memory, the longword-access-enable bit (LWAE) must be set.

A buffered data path stores data from the UNIBUS in a buffer until multiple words of data have been transferred (except in longword-aligned, 32-bit, random-access mode as discussed in Section 12.1.3.5). Then, the UNIBUS adapter transfers the contents of the buffer to the appropriate physical address in a single backplane interconnect operation. The procedure for a UNIBUS write operation that transfers data from a device to memory is broken into individual steps.

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

The procedure for a UNIBUS read operation that transfers data from main memory to a device varies according to the type of UNIBUS adapter. Those adapters that can perform a prefetch function complete UNIBUS reads from memory more quickly than those that cannot. The prefetch feature accomplishes this improved performance by automatically filling the data path buffer after the buffer's contents are transferred to the UNIBUS.

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

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

UNIBUS and Q22 Bus Device Support

12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

- 7 The buffered data path then initiates a read function to prefetch data from memory.
- 8 When the prefetch is complete, the buffered data path sets the buffer-not-empty flag to indicate that the buffers now contain valid data.

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

A UNIBUS read function without prefetch includes the following steps:

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

12.1.3.3 Byte-Offset Data Transfers

The UNIBUS adapter has a byte-offset register; thus, words that are not word-aligned can be transferred to and from any device on the UNIBUS regardless of whether the device supports non-word-aligned transfers.

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

12.1.3.4 Purging a Buffered Data Path

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

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

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

UNIBUS and Q22 Bus Device Support

12.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

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

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

To transfer data in the longword-aligned, 32-bit, random-access mode, the driver 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 operation that transfers data from main memory to a device, the function occurs as follows:

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

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

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

To ensure that random-access mode works correctly regardless of the VAX system involved, the writer of a device driver should ensure that a device assigned to a buffered data path does not repeatedly address the same longword. On certain systems, a UNIBUS device that polls a single longword, waiting for data, will constantly be returned the same data.

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

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

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

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

In general, a driver fork process does not directly access the ADP. Instead, a driver calls VMS routines that perform adapter-related services, such as the following:

- Allocating a buffered data path
- Allocating map registers or alternate map registers
- Loading map registers or alternate map registers
- Deallocating map registers or alternate map registers
- Purging a buffered data path
- Deallocating a buffered data path

The critical responsibility of device drivers that actively compete for such shared I/O adapter resources as map registers and data paths is to ensure the synchronized access of adapter resources. Drivers that share these resources must execute at the same fork IPL. In a VMS multiprocessing system, they must additionally contend for the same fork lock. A given driver code thread that must attempt to access its fork database can only do so if suitably synchronized.

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 of the following steps to initiate an I/O transfer to or from a device on a UNIBUS, MicroVAX 3600-series Q22 bus, MicroVAX II Q22 bus, or MicroVAX I Q22 bus.

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

| Operation | Applicable to |
|----------------------------------|---|
| Requests buffered data path | UNIBUS |
| Requests map registers | UNIBUS, MicroVAX 3600 series, MicroVAX II |
| Requests alternate map registers | MicroVAX 3600 series, MicroVAX II |
| Loads map registers | UNIBUS, MicroVAX 3600 series, MicroVAX II |
| Loads alternate map registers | MicroVAX 3600 series, MicroVAX II |
| Calculates starting bus address | UNIBUS, MicroVAX 3600 series, MicroVAX II, MicroVAX I |
| Activates device | UNIBUS, MicroVAX 3600 series, MicroVAX II, MicroVAX I |
| Waits for interrupt | UNIBUS, MicroVAX 3600 series, MicroVAX II, MicroVAX I |

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:

| Operation | Applicable to |
|----------------------------------|--|
| Purges data path | UNIBUS, MicroVAX 3600 series, MicroVAX II, MicroVAX I ¹ |
| Releases buffered data path | UNIBUS |
| Releases map registers | UNIBUS, MicroVAX 3600 series, MicroVAX II |
| Releases alternate map registers | MicroVAX 3600 series, MicroVAX II |

¹Regardless of whether the associated VAX system provides buffered data paths, drivers of all devices should initiate a purge of the data path after a transfer. The purge operation enables the detection of memory parity errors that may have occurred during the transfer, as described in the sections on the PURDPR macro and IOC\$PURGDATAP in Appendixes B and C, respectively.

Because of the different requirements of DMA transfers on different VAX and MicroVAX systems, a driver must contain some run-time conditional code in order to function for equivalent UNIBUS, MicroVAX 3600-series, MicroVAX II, and MicroVAX I devices. Appendix E contains an example of one driver that supports the RL11 on the UNIBUS and the RLV11 on the MicroVAX I and MicroVAX II Q22 bus.

Regarding the material presented in this section, UNIBUS driver writers should read Sections 12.2.1 through 12.2.7.3. MicroVAX 3600-series and MicroVAX II driver writers should read Section 12.2.1.3 and Sections 12.2.2 through 12.2.7.3. MicroVAX I driver writers should turn directly to Section 12.2.8. Because the MicroVAX I provides no scatter-gather map, MicroVAX I device drivers must perform transfers according to the method described therein.

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

12.2.1 Selecting and Requesting a Data Path

DMA device drivers for certain VAX systems can elect to request the use of a UNIBUS adapter buffered data path to accelerate data transfers (as described in Section 12.1.3). Other VAX processing systems, such as the MicroVAX 3600 series, MicroVAX II and VAX-11/730, provide no buffered data paths for data transfers. The descriptions of the direct data path in the following sections apply to drivers written for devices in those systems.

12.2.1.1 Requesting a Buffered Data Path

Some VAX systems allow UNIBUS drivers to request temporary or permanent allocation of a buffered data path (see Table 12-1). After the driver fork process gains access to the controller (see Section 8.3.1), it requests a buffered data path by invoking the VMS macro REQDPR. REQDPR calls a VMS routine named IOC\$REQDATAP that locates the ADP. To do this, IOC\$REQDATAP uses a series of pointers that begins in the current unit control block (UCB), as follows:

UCB → CRB → ADP

IOC\$REQDATAP performs the following services:

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

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

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

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

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

12.2.1.2 Requesting a Permanent Buffered Data Path

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

- 1 Issue the FORK macro to drop IPL to fork IPL. The VMS fork dispatcher causes the following steps to be performed at fork IPL under the ownership of the required fork lock in a VMS multiprocessing system. (The consequences of forking in a unit initialization routine are discussed at length in Section 11.1.5.)
- 2 Test the path-lock bit (VEC\$V_PATHLOCK) in the data-path-number field of the CRB (CRB\$L_INTD+VEC\$B_DATAPATH) to ensure that a data path is not already allocated for this device.
- 3 Call the subroutine IOC\$REQDATAPNW to allocate the data path as follows:

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

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

- 4 If the data path has been successfully obtained, set the path-lock bit (VEC\$V_PATHLOCK) in the CRB at CRB\$L_INTD+VEC\$B_DATAPATH.

The driver-loading procedure calls the unit initialization routine for each unit that the driver serves. A unit initialization routine that contains the code described previously will permanently allocate one buffered data path for each CRB associated with the driver, which is one path for each controller that the driver serves.

Because some VAX systems have few buffered data paths (refer to Table 12-1), device drivers running in these systems must limit their allocation of permanent buffered data paths. For example, if the drivers loaded on a VAX-11/750 permanently allocated all three of the system's buffered data paths, none would remain for normal system operations. As a result, I/O transfers requiring a buffered data path would wait forever.

12.2.1.3 Requesting the Direct Data Path

Because the UNIBUS adapter or Q22 bus interface arbitrates among devices that wish to use the direct data path and initializes the data path field in the CRB (CRB\$L_INTD+VEC\$B_DATAPATH) to 0 (0 = direct data path), drivers are not required to invoke the REQDPR macro to request the direct data path.

Some VAX systems, such as the VAX-11/780 or VAX 8600, do not permit byte-offset transfers on the direct data path (see Table 12-1). Because the UNIBUS itself is word-oriented, such a system must ensure that the data buffer is aligned on a word boundary for word-aligned devices.

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

12.2.1.4 Mixed Use of Direct and Buffered Data Paths

A UNIBUS device driver can use the buffered data path for certain operations, then use the direct data path for other operations. To accomplish this task, the driver should allocate a buffered data path for buffered I/O. When the operation is completed, the driver should then purge and release the buffered data path. The release automatically resets the data path number to zero, which signifies a direct data path. When the driver has finished using the direct data path, it should purge it (but not release it). (A purge of the direct data path is a NOP and always yields success.)

12.2.2 Requesting Map Registers

The UNIBUS adapter and Q22 bus interface allow UNIBUS and Q22 bus drivers, respectively, to allocate map registers as needed or to allocate them permanently.

12.2.2.1 Allocating Map Registers

After the driver fork process gains access to the controller (see Section 8.3.1), it can request a set of adapter map registers (registers 0 through 495) by invoking the VMS macro REQMPR. This macro calls the routine IOC\$REQMAPREG. IOC\$REQMAPREG calculates the number of map registers needed for a transfer, allocating one map register for each full or partial page of the buffer (based on the values of UCB\$W_BCNT and UCB\$W_BOFF). In addition, it reserves an additional map register to be marked invalid to stop a potential runaway transfer and inhibit prefetches from the page past that in which the end of the buffer resides. Finally, IOC\$REQMAPREG may allocate one more extra map register to ensure that an even number of map registers is allocated.

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

UCB → CRB → ADP

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

When the map registers are available, IOC\$REQMAPREG allocates them and adjusts the appropriate information about the allocation of map registers in the ADP. IOC\$REQMAPREG then writes the number of the first map register and the number of map registers allocated into the CRB and returns control to the driver fork process.

VMS supplies a similar macro (REQALT) and routine (IOC\$REQALTMAP) that MicroVAX 3600-series or MicroVAX II drivers can use to allocate a set of alternate map registers (registers 496 through 8191). REQALT and IOC\$REQALTMAP perform the allocation in the same manner as REQMPR and IOC\$REQMAPREG. Note that, although VMS records the allocation of standard and alternate map registers in separate areas of the ADP and CRB, IOC\$REQMAPREG and IOC\$REQALTMAP use the same UCB fields to calculate the number of map registers required for a transfer.

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

12.2.2.2 Permanently Allocating Map Registers

A device driver can allocate a permanent set of map registers with code in its unit initialization routine. The number of map registers permanently allocated must be sufficient for the largest possible transfer and must include an extra map register to be marked invalid to prevent a runaway transfer.

A unit initialization routine performs the following steps to permanently allocate a set of map registers:

- 1 Issue the FORK macro to drop IPL to fork IPL. The VMS fork dispatcher causes the following steps to be performed at fork IPL under the ownership of the required fork lock in a VMS multiprocessing system. (The consequences of forking in a unit initialization routine are discussed at length in Section 11.1.5.)
- 2 Test the map-lock bit (VEC\$V_MAPLOCK) in the CRB (CRB\$L_INTD+VEC\$W_MAPREG) to ensure that map registers are not already allocated for this device.
- 3 Load the number of map registers required into R3.
- 4 Call the 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 number of the first of the allocated map registers at CRB\$L_INTD+VEC\$B_NUMREG and CRB\$L_INTD+VEC\$W_MAPREG, respectively. It then returns with the low-order bit set in R0. Otherwise, it returns with the low-order bit of R0 clear.

- 5 If map registers have been successfully allocated, set the map-lock bit in the CRB (VEC\$V_MAPLOCK in CRB\$L_INTD+VEC\$W_MAPREG).

MicroVAX 3600-series drivers and MicroVAX II drivers perform the following steps to allocate a permanent set of alternate map registers (registers 496 through 8191):

- 1 Issue the FORK macro to drop IPL to fork IPL. The VMS fork dispatcher causes the following steps to be performed at fork IPL under the ownership of the required fork lock in a VMS multiprocessing system. (The consequences of forking in a unit initialization routine are discussed at length in Section 11.1.5.)
- 2 Test the alternate-map-lock bit (VEC\$V_ALTLOCK) in the CRB (CRB\$L_INTD+VEC\$W_MAPALT) to ensure that alternate map registers are not already allocated for this device.
- 3 Load the number of alternate map registers required into R3.
- 4 Call the VMS routine IOC\$ALOALTMAPN with a JSB instruction:

```
JSB G^IOC$ALOALTMAPN
```

If IOC\$ALOALTMAPN successfully allocates the alternate map registers, it stores the number of map registers allocated and the number of the first of the allocated map registers in CRB\$L_INTD+VEC\$W_NUMALT and CRB\$L_INTD+VEC\$W_MAPALT, respectively. It then returns with the low-order bit set in R0. Otherwise, it returns with the low-order bit of R0 clear.

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

- 5 If alternate map registers have been successfully allocated, set the alternate-map-lock bit in the CRB (VEC\$V_ALTLOCK in CRB\$L_INTD+VEC\$W_MAPALT).

The driver-loading procedure calls the unit initialization routine once for each unit associated with the driver. If the unit initialization routine contains the code described previously, it permanently allocates a set of map registers for each CRB associated with the driver, which is a set of registers for each device controller that the driver serves. Because VMS records the allocation of standard and alternate map registers in separate areas of the ADP and CRB, a MicroVAX 3600-series or MicroVAX II driver could permanently allocate a set of registers from both areas.

12.2.3 Loading Map Registers

Once a driver fork process has assigned a data path and allocated a set of map registers, it can request VMS to load the map registers with physical page-frame numbers (PFNs) by invoking the VMS macro LOADUBA.³ LOADUBA calls the VMS routine IOC\$LOADUBAMAP to load each allocated map register with the following 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 (UNIBUS drivers only).
- The page-frame number of a page in memory.
- A bit setting to indicate that the transfer operates in longword-aligned, random-access mode on the buffered data path; this bit is set when VEC\$V_LWAE is set in VEC\$B_DATAPATH (UNIBUS drivers only).

IOC\$LOADUBAMAP loads the PFN of the first page of the transfer into the first allocated map register, the PFN 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 prevent a prefetch from an invalid page.

To calculate the PFN used in the I/O transfer, IOC\$LOADUBAMAP uses three fields that VMS has written into the UCB:

- UCB\$W_BOFF—Byte offset in the first page of the transfer
- UCB\$W_BCNT—Number of bytes to transfer
- UCB\$L_SVAPTE—Virtual address of the page-table entry that contains the PFN of the first page of the transfer

³ MicroVAX 3600-series and MicroVAX II DMA driver writers also use the LOADUBA macro to load a set of standard map registers (registers 0 through 495).

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

IOC\$LOADUBAMAP determines the data path's number (for UNIBUS devices), the number of the first map register, the address of the first map register, and the number of allocated map registers from the CRB and the ADP, as follows:

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

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

VMS supplies a similar macro (LOADALT) and routine (IOC\$LOADALTMAP) that MicroVAX 3600-series or MicroVAX II drivers can use to load a set of previously allocated alternate map registers (registers 496 through 8191). LOADALT and IOC\$LOADALTMAP load the alternate map registers in the same manner as LOADUBA and IOC\$LOADUBAMAP load standard map registers.

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

12.2.4 Computing the Starting Address of a Transfer

The driver fork process must calculate the starting address of a DMA transfer and load this address into the appropriate device register. MicroVAX I device drivers perform the procedure outlined in Section 12.2.8. UNIBUS drivers and other Q22 bus drivers that use a set of standard map registers take the following steps to make the calculation:

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

MicroVAX 3600-series and MicroVAX II drivers that use a set of alternate map registers perform a similar procedure, as follows:

- 1 Write the byte-offset-in-page field of the UCB (UCB\$W_BOFF) into bits 0 through 8 of a general register.

⁴ One example of a device that does not treat the extended memory bits in this fashion is the DRV11-WA, the code for which is listed in Appendix F. For the DRV11-WA, code in XADRIVER stores bits 7 and 8 of the map register number in a discrete device bus address extension register, then clears the extended address bits of the device's CSR. In contrast, XADRIVER handles the DR11-W according to the method described previously.

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

- 2 Get the number of the starting alternate map register for the transfer from `CRB$L_INTD+VEC$W_MAPALT`. Write bits 0 through 6 of this 13-bit value into bits 9 through 15 of the general register.
- 3 Write bits 0 through 15 of the general register into the device's buffer address register.
- 4 Write bits 7 through 12 of the map register number, acquired in step 2, into the extended memory bits of the appropriate device register (usually the control and status register (CSR)).

12.2.5 Computing the Transfer Length

Generally, a device driver must indicate to the device the size of a DMA transfer by writing to a device register. If a device expects the transfer size as a word count, for instance, the start-I/O routine computes the length of the transfer in words by dividing the byte count field of the UCB (`UCB$W_BCNT`) by 2. The routine loads the computed value into the device's word-count register. One of the FDT routines that processes the I/O request must ensure that the byte count for the transfer is even. An odd byte count results in the user's not receiving the last byte of data.

12.2.6 Activating the Device

Because a driver fork process can address device registers as though they were any other virtual address, the loading of the device buffer address register and CSR are simple procedures. The driver locates the CSR address of the device in the interrupt dispatch block (IDB), as follows:

UCB → CRB → IDB → CSR address

The CSR address is the virtual address of a device register. All other device registers are located at constant offsets from the CSR address. If, for example, the CSR is the first device register and the device's word-count register is the third device register, the device driver can describe the device register offsets and load the word-count register with the following series of instructions:

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

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

12.2.7 Completing a DMA Transfer

After a UNIBUS, MicroVAX 3600-series, MicroVAX II, or MicroVAX I device driver fork process activates a DMA device, the driver waits for a device interrupt by invoking a VMS macro that suspends execution of the driver. When the device requests a hardware interrupt, the interrupt dispatcher gains control.

The dispatcher saves R0 through R5 and transfers control to the driver's interrupt service routine. If the interrupt service routine can match the interrupt with a suspended driver fork process, it reactivates the driver fork process at the point where execution was suspended. Most driver fork processes almost immediately invoke the VMS macro IOFORK.

IOFORK calls the 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 processor-specific fork queue corresponding to the device's fork IPL. 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 cleanup operations, such as purging the data path and deallocating adapter resources used in the DMA transfer.

12.2.7.1 Purging the Data Path

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

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

- 1 Saves the contents of R4 on the stack.
- 2 Locates the CRB as follows:
R5 → UCB → CRB
- 3 Obtains the starting address of the UNIBUS adapter's register space and stores it in R2.
- 4 Extracts the number of the data path to be purged from the CRB and loads it into R1.
- 5 Stores the address of the data path register in R4.
- 6 Instructs the UNIBUS adapter or Q22 bus interface to purge the data path. The routine then modifies R0 through R2 to contain the following information:

R0 Success/failure status. If the purge completes without error, the routine sets SSS_NORMAL in this register. If a data-path error does occur, R0 is clear and the hardware is reset.

R1 Contents of the data-path register.

R2 Address of the first adapter map register.

The address of the CRB remains in R3. This address, along with the information in R1 and R2, is used as input to the error logging routine in the event of a data-path error.

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

- 7 Restores the information stored on the stack to R4 and returns to the address in the driver immediately after the invocation of the PURDPR macro.
- 8 Some machine implementations also check for memory errors that might have occurred during the DMA operation, and, if an error is detected, log it.

If a data-path error occurs during a data-path purge, the driver should retry the entire DMA transfer.

12.2.7.2 Releasing a Buffered Data Path

A driver fork process releases a buffered data path by invoking the VMS macro RELDPR. RELDPR calls a VMS routine, IOC\$RELDATAP, that determines which data path was assigned to the driver fork process and releases the data path to a waiting driver. The driver must be executing at fork IPL.

The data-path number is stored in the CRB. IOC\$RELDATAP locates it as follows:

UCB → CRB → number of the data path

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

UCB → CRB → ADP → data-path wait queue

If another driver is waiting for a buffered data path, IOC\$RELDATAP grants that driver fork process the data path, restores its context from its UCB fork block, and transfers control to the saved driver PC. When IOC\$RELDATAP can allocate no more data paths, the routine returns to the driver that released the data path. This diversion of driver processing is transparent to the driver fork process.

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

12.2.7.3 Releasing Map Registers

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

UCB → CRB → number of the first map register

UCB → CRB → number of allocated map registers

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

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

- 1 Dequeues the fork block and restores driver context

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

- 2 Satisfies the map-register request, if possible
- 3 Reactivates the driver fork process at the instruction following the driver's request for map registers
- 4 Repeats steps 1 through 3

If the standard-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 fork process that released the map registers.

VMS supplies a similar macro (RELALT) and routine (IOC\$RELALTMAP) that MicroVAX 3600-series and MicroVAX II drivers can use to release a set of alternate map registers (registers 496 through 8191). RELALT and IOC\$RELALTMAP perform the release in the same manner as RELMPR and IOC\$RELMAPREG. Note that VMS records the allocation of standard and alternate map registers in separate areas of the ADP and CRB.

12.2.8 Considerations for MicroVAX I DMA Devices

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

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

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

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

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

UNIBUS and Q22 Bus Device Support

12.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

```
MOVL   UCB$L_CRB(R5),R1           ;Get CRB address
MOVL   CRB$L_AUXSTRUC(R1),R2      ;Memory alloc failure during
                                      ; controller initialization?
BEQL   70$                        ;If equal, yes, leave offline
MOVL   R2,UCB$A_DL_BUF_VA(R5)    ;Save buffer's virtual address
EXTZV  #VA$V_VPN,#VA$$_VPN,R2,R1;Get virtual page number
                                      ; of buffer
MOVL   G^MMG$GL_SPTBASE,R0       ;Get base address of SPTs
MOVL   (R0)[R1],R0               ;Get the PTE contents
BICL3  #^C<VA$M_BYTE>,R2,R1     ;get buffer offset (BA00-BA08)
ASSUME PTE$$_PFN GE 13
INSV   R0,#9,#13,R1              ;Copy BA09-BA21
MOVL   R1,UCB$A_DL_BUF_PA(R5)    ;Save buffer's physical address
```

70\$: RSB

- 2 Moves the low word (bits 0 to 15) of the buffer physical address into the device's buffer address register.
- 3 Moves the extended address bits of the buffer's physical address into the device's extended address register or the device's CSR, as required by the device.
- 4 Activates the device as described in Section 12.2.6.
- 5 If the transfer size exceeds the size of the buffer, returns to step 1.

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

A MicroVAX I driver should complete the transfer as described in Section 12.2.7. The driver should call `IOC$PURGDATAP` in order to detect and log any memory errors that might have occurred during the transfer.

12.3 Interrupt Dispatching in a UNIBUS/Q22 Bus System

The *interrupt dispatcher* is a combination of hardware and software that routes interrupts from a device on the UNIBUS or Q22 bus to the appropriate device driver's interrupt service routine. Although there are slight differences in the implementation of the interrupt dispatcher in different VAX systems, it performs the same tasks in any given VAX environment.

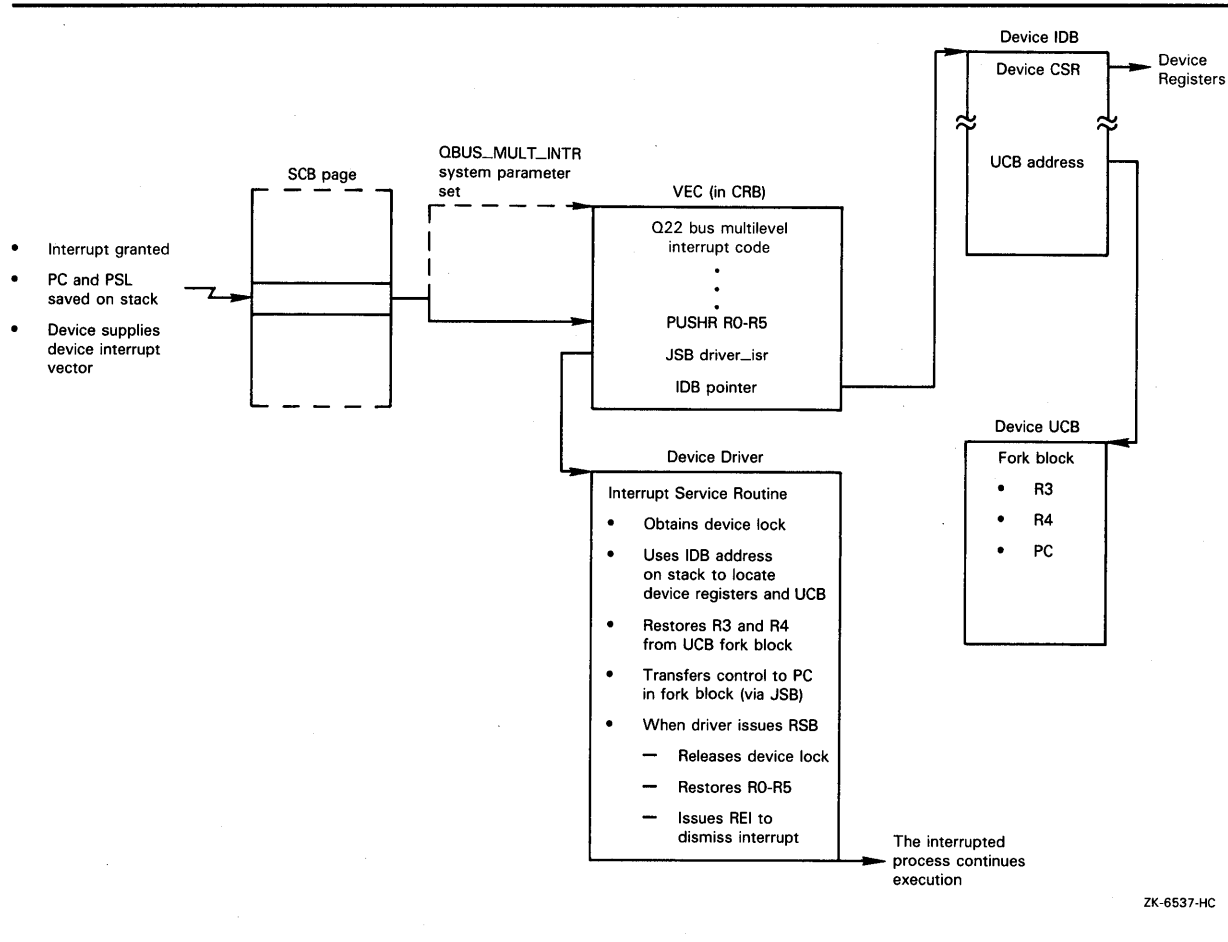
When a processor grants a device interrupt, the processor microcode first saves the PC and PSL of the currently executing code on the interrupt stack. The device responds to the grant by supplying a *device interrupt vector* in the range of 0 to 777₈ to the processor. VAX/VMS uses the device interrupt vector to locate the correct interrupt transfer vector for the device. The interrupt transfer vector structure (VEC) contains a short routine which issues a JSB instruction to the device driver's interrupt service routine. Execution continues at the location of the transfer vector.

UNIBUS and Q22 Bus Device Support

12.3 Interrupt Dispatching in a UNIBUS/Q22 Bus System

This is a somewhat simplified view of the interrupt dispatcher's activities. Figure 12-5 and Figure 12-6 depict the flow of interrupt dispatching from the time that a processor grants the interrupt to the processing of the interrupt within a device driver's interrupt service routine. The following subsections provide a more complete description of the role of each component in the servicing of device interrupts.

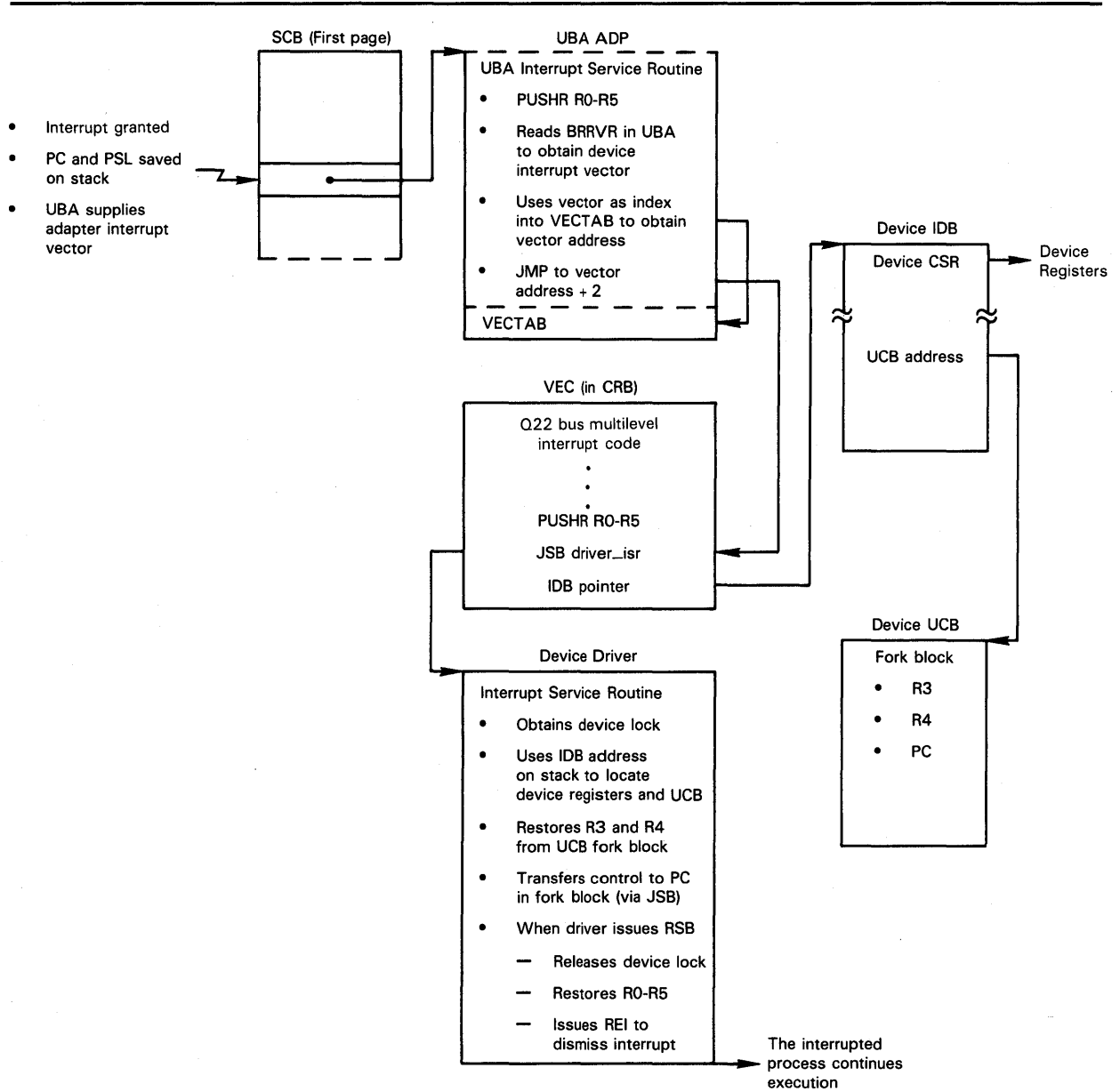
Figure 12-5 Direct-Vector Interrupt Dispatching



UNIBUS and Q22 Bus Device Support

12.3 Interrupt Dispatching in a UNIBUS/Q22 Bus System

Figure 12–6 Non-Direct-Vector Interrupt Dispatching



ZK-6536-HC

12.3.1 Direct-Vector and Non-Direct-Vector Interrupt Dispatching

The system control block (SCB) contains the vectors that the VAX architecture uses to dispatch all interrupts and exceptions. The size of the SCB is system dependent. Page 1, the only SCB page defined by the VAX architecture, contains the addresses of software and hardware interrupt service routines and exception service routines.

UNIBUS and Q22 Bus Device Support

12.3 Interrupt Dispatching in a UNIBUS/Q22 Bus System

Table 12-2 VAX System UNIBUS/Q22 Bus Interrupt Dispatching

| VAX System | Type of Interrupt Dispatching | Location of Adapter Dispatch Table |
|---|-------------------------------|------------------------------------|
| VAX-11/750, VAX-11/730, MicroVAX 3600 series, MicroVAX II, MicroVAX I | Direct | SCB pages 2 and 3 |
| VAX-11/780, VAX-11/785, VAX 8600, VAX 8650, VAX 8670 | Non-Direct | ADP vector-jump table |
| VAX 8200/8250/8300/8350 | Direct | SCB page 2 ¹ |
| VAX 8530/8550/8700/8800/8830/8850 | Direct | SCB page 2 ¹ |
| VAX 6200 series | Direct | SCB page 2 ¹ |

¹Subsequent pages may be used if there is more than one DWBUA in the system.

The SCB therefore has an initial, albeit system-dependent role, in servicing device interrupts. A UNIBUS/Q22 bus system employs either of two methods to dispatch a device interrupt (see Table 12-2):

- If the system in question is a MicroVAX 3600 series, MicroVAX II, or MicroVAX I—or uses a direct-vector UNIBUS adapter (UBA)—it dispatches a device interrupt directly through page 2 (or subsequent pages, for VAX systems with more than one such UNIBUS) of the SCB. It takes the device interrupt vector and uses it as an index into the appropriate SCB page, thus obtaining the address of the appropriate interrupt transfer vector structure (VEC) for the device. This process is known as *direct-vector interrupt dispatching*.
- In a VAX system that employs a non-direct-vector UNIBUS adapter, the adapter posts an interrupt that is dispatched through a vector in page 1 of the SCB that points to a UBA interrupt service routine. For each non-direct-vector UBA adapter, the VMS adapter initialization procedure creates four such interrupt service routines, each corresponding to a device BR (bus request) level, and places them in an area of nonpaged pool specially allocated at the end of the adapter control block.

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

- 1 Saves R0 through R5 on the interrupt stack.
- 2 Reads the UNIBUS adapter's Bus Request Receive Vector register (BRRVR) to determine the vector address of the device requesting the interrupt.
- 3 Uses the vector address as an index into the adapter dispatch table to locate the interrupt transfer vector for the device in the CRB. For each non-direct-vector UBA, an adapter dispatch table (also known as the vector-jump table) is located after the UBA interrupt service routines in nonpaged pool.
- 4 Transfers control by means of a JMP instruction to a location within the interrupt transfer routine contained in the VEC structure. This process is known as *non-direct-vector interrupt dispatching*.

UNIBUS and Q22 Bus Device Support

12.3 Interrupt Dispatching in a UNIBUS/Q22 Bus System

From another point of view, direct-vector interrupt dispatching and non-direct-vector interrupt dispatching are characterized by the following two factors:

- Location of the adapter dispatch table
- Contents of the interrupt transfer routine

The following sections further examine the differences in the dispatching methods. Figure 12-5 and Figure 12-6 present the flow of tasks performed within the context of both direct-vector and non-direct-vector interrupt dispatching.

12.3.2 Adapter Dispatch Table

The *adapter dispatch table* contains 128 longword vectors, each of which corresponds to a device interrupt vector. Each longword vector within the adapter dispatch table contains either the address of an *interrupt transfer vector* structure (VEC), located within the channel request block (CRB) of the device's controller or, if no device is using the vector, the address of the adapter's unexpected interrupt service routine. In either case, the address contained in the adapter dispatch table is longword aligned.

The location of the adapter dispatch table, as signified by the contents of ADP\$L_VECTOR, is system dependent:

- The MicroVAX 3600 series, MicroVAX II, MicroVAX I, and those VAX systems that employ direct vector UNIBUS adapters situate the adapter dispatch table in the second and subsequent pages of the system control block (SCB), as described previously.
- Those VAX systems that employ non-direct-vector interrupt dispatching situate the adapter dispatch table in a region of nonpaged pool (known also as the vector-jump table and commonly referred to as VECTAB).

12.3.3 Interrupt Transfer Vector and Interrupt Transfer Routine

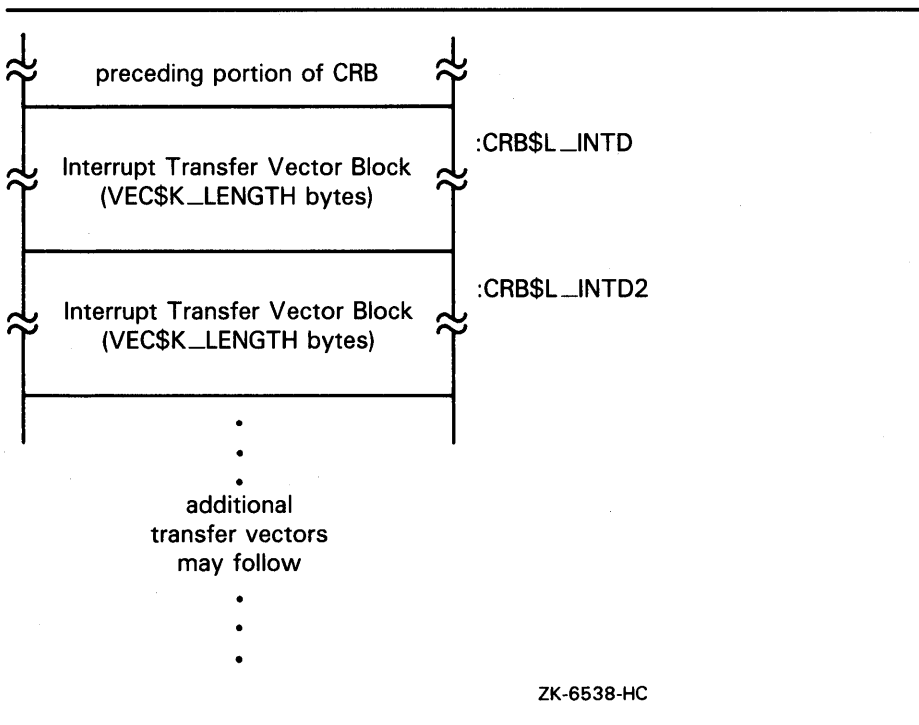
The *interrupt transfer vector* data structure (VEC) is located within the channel request block (CRB) corresponding to the interrupting device's controller, as shown in Figure 12-7.

The interrupt transfer vector structure (see Figures 12-8 and A-7) starts with several lines of executable code known as the interrupt transfer routine. It also stores several pieces of data, including pointers to the unit and controller initialization routines in the device driver, the address of the interrupt dispatch block (IDB), and the address of the adapter control block (ADP). The interrupt transfer vector may also include information reflecting the disposition of the adapter's map registers.

UNIBUS and Q22 Bus Device Support

12.3 Interrupt Dispatching in a UNIBUS/Q22 Bus System

Figure 12-7 VEC Structures Within a CRB



There may be one or more interrupt transfer vectors within a single CRB, as shown in Figure 12-7. VMS creates the appropriate number of interrupt transfer vector structures within a CRB according to the value specified in the /NUMVEC qualifier to the SYSGEN command CONNECT. The default value is 1.

VMS automatically initializes the interrupt dispatching instructions and the data structure locations in each of the specified vectors.

The *interrupt transfer routine* is a piece of executable code at the beginning of each interrupt transfer vector. It is the interrupt transfer routine that ultimately transfers control to the device driver's interrupt service routine and, to a certain extent, establishes the context for its execution.

For those VAX systems employing non-direct-vector interrupt dispatching, the interrupt transfer routine consists of only one instruction:

```
JSB    @#^driver-isr-address
```

For those VAX systems employing direct-vector interrupt dispatching, the interrupt transfer routine consists of the following two instructions:

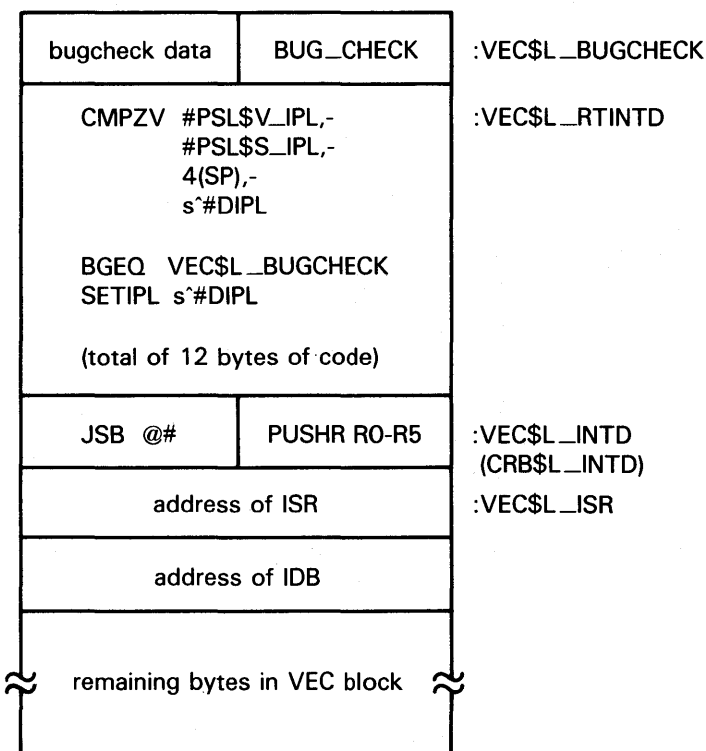
```
PUSHR  #^M<R0,R1,R2,R3,R4,R5>  
JSB    @#^driver-isr-address
```

Note: If the VAX system is a MicroVAX 3600-series or MicroVAX II system with multilevel device interrupt dispatching enabled, these two instructions are preceded by some instructions that check the legality of the Q22 bus configuration and conditionally lower IPL. See Section 12.3.4 for a description of this optional function of the interrupt transfer routine.

UNIBUS and Q22 Bus Device Support

12.3 Interrupt Dispatching in a UNIBUS/Q22 Bus System

Figure 12–8 Interrupt Transfer Vector Block (VEC)



ZK-6539-HC

The driver-loading procedure obtains the address of the interrupt service routine for each interrupt transfer vector structure from the reinitialization portion of the driver prologue table (see Section 6.1). This section of the DPT contains one or more DPT_STORE macros that identify the addresses of the interrupt service routines. For example:

```
DPT_STORE,CRB,CRB$_INTD+VEC$_ISR,D,isr_for_1st_vector
DPT_STORE,CRB,CRB$_INTD2+VEC$_ISR,D,isr_for_2nd_vector
DPT_STORE,CRB,CRB$_INTD+(2*VEC$_K_LENGTH)+VEC$_ISR,D,isr_for_3rd_vector
```

The number of DPT_STORE macros that identify interrupt service routines must equal the number of vectors given in the /NUMVEC qualifier to the SYSGEN command CONNECT to avoid errors in device initialization or interrupt handling.

Immediately following the interrupt transfer routine in the CRB is the address of the interrupt dispatch block (IDB) associated with the CRB. When the JSB instruction executes, a pointer to the address of the IDB is pushed onto the top of the stack as though it were a return address. The driver interrupt service routine can use this IDB address as a pointer into the I/O database. See Figure 12–5 and Figure 12–6 for an illustration of the context available to a driver's interrupt service routine when it is called by the interrupt transfer routine.

UNIBUS and Q22 Bus Device Support

12.3 Interrupt Dispatching in a UNIBUS/Q22 Bus System

12.3.4 Multilevel Device Interrupt Dispatching for Q22 Bus Devices

VAX peripheral devices request interrupts at IPLs 20 through 23. IPLs 20 through 23 generally correspond with the four bus request levels of the UNIBUS (BR4 through BR7) and Q22 bus (BIRQ4 through BIRQ7).

The UNIBUS also has four bus grant lines (BG4 through BG7). Because of this, interrupt dispatching for UNIBUS devices inherently occurs at four levels. When a UNIBUS device requests an interrupt at BR4, for example, from a processor executing at an IPL lower than IPL 20, the processor grants the interrupt to the device at IPL 20 (BG4). If the processor is already executing at IPL 20 or above, the device interrupt remains pending.

The MicroVAX 3600-series and MicroVAX II Q22 bus architecture has but one bus grant line (BIAK). As a result, the central processor must, by default, grant all Q22 bus device interrupts at a single IPL (IPL 23), even though it arbitrates interrupt requests according to the bus request line used. When a Q22 bus device requests an interrupt at BIRQ4, for example, from a processor executing at an IPL lower than IPL 20, the processor grants the interrupt, unconditionally raising IPL to IPL 23. If the processor is already executing at IPL 20 or above, the interrupt remains pending.

There are certain consequences of this implementation of interrupt dispatching on the configuration and behavior of Q22 bus devices:

- 1 Because the MicroVAX 3600 series and MicroVAX II dispatch Q22 bus interrupts at a single IPL, it is essential that Q22 bus devices that request interrupts at a high BIRQ be positioned on the bus closer to the CPU than devices that interrupt at a low BIRQ. (To determine the BIRQ level of any given Q22 bus device, refer to its hardware user's guide.)
- 2 It is possible for a Q22 bus peripheral that requests interrupts at a low BIRQ to block the granting of an interrupt to a peripheral that requests interrupts at a higher BIRQ. For instance, the processor could grant an interrupt to a BIRQ4 device, elevating its IPL to IPL 23 in the process. While executing at IPL 23, the processor would not grant the interrupt request of a BIRQ7 device. In a real-time environment, where I/O operations to one peripheral must always have priority over lesser forms of I/O, this behavior can cause problems.

VMS incorporates a means by which system programmers, concerned about real-time performance issues, can avoid these problems and implement multilevel interrupt dispatching for devices on a MicroVAX 3600-series and MicroVAX II system.

The default behavior of a MicroVAX 3600-series or MicroVAX II interrupt dispatcher is sufficient for a typical MicroVAX 3600-series or MicroVAX II system. Only system managers and system programmers involved in real-time system environments should attempt to use the multilevel device interrupt dispatching capability of VMS Version 5.0. Such users should possess a thorough understanding of the VMS interrupt dispatching mechanism and the means by which VMS synchronizes access to structures in the I/O database.

If you must enforce real-time device priorities in your Q22 bus system, you can do so by setting the QBUS_MULT_INTR system parameter. This static parameter causes VMS to set up the proper code and data structures to enable multilevel device interrupt dispatching at system initialization.⁵

⁵ Other VAX systems ignore the QBUS_MULT_INTR system parameter.

UNIBUS and Q22 Bus Device Support

12.3 Interrupt Dispatching in a UNIBUS/Q22 Bus System

When you bootstrap a MicroVAX 3600-series system or MicroVAX II system with the `QBUS_MULT_INTR` system parameter set, VMS initializes data structures for each device and implements multilevel device interrupts as follows:

- Locates the address of the device driver's interrupt service routine in the driver's DPT and stores it in the appropriate VEC data structure.
- Adjusts the corresponding vectors in the second page of the SCB so that they point to the multilevel device interrupt dispatching code in the interrupt transfer vector (that is, replacing the address of `CRB$_INTD` with `CRB$_INTD+VEC$_RTINTD`).
- Sets up data and code at `VEC$_RTINTD` in the first interrupt transfer vector for the device. This code performs special checks for the legality of the Q22 bus configuration and conditionally lowers IPL, when necessary, to service the interrupts of low priority devices.

When multilevel device interrupt dispatching is enabled, interrupt dispatching proceeds as in the default case. However, after the processor raises IPL to IPL 23 to grant the interrupt, the dispatching of the interrupt results in the lowering of IPL to device IPL, if necessary, to service the interrupt. As a result, the processor, executing at the lower IPL, will be free to grant interrupts from higher priority devices.

Prior to implementing this feature in a MicroVAX 3600-series system or MicroVAX II system, users should perform the following tasks:

- 1 Ensure that the Q22 bus is properly configured.
- 2 Adapt any existing non-DIGITAL-supplied device driver so that it correctly initializes any secondary vector (VEC) structures it uses and also refers to fields in the channel request block (CRB) and VEC by the proper symbolic offsets.

12.3.4.1 Ensuring That the Q22 Bus Is Properly Configured

The MicroVAX 3600-series and MicroVAX II Q22 bus architecture mandates that devices with the ability to interrupt at a high BIRQ level (for instance, BIRQ7) must be positioned on the bus closer to the CPU than devices that interrupt at lower BIRQ levels. In a Q22 bus system, when the processor grants an interrupt to a device, the processor passes the interrupt down the BIAK line to the first device on the bus. If the device is not the one requesting an interrupt, it is responsible for propagating the acknowledgment grant to the next device on the bus, and so on. However, if a device coincidentally initiates an interrupt just before it would be required to pass the grant further down the bus, it may instead "steal" the grant for itself.

The problems of an illegally configured Q22 bus can be illustrated in the following two examples.

Consider the case of a device that interrupts at BIRQ6 (IPL 22). Assume the CPU is at IPL 21. The bus arbitrator will grant the device its interrupt and send the grant down the Q22 bus. However, if at this precise moment a device that is closer to the CPU on the grant path, and that interrupts at BIRQ5 (IPL 21), initiates an interrupt, it may not propagate the grant as it should. Instead, it may steal the grant and assume ownership of the interrupt. Thus, there exists the possibility of an IPL 21 device successfully interrupting a processor executing at IPL 21. The end result is an unpredictable break in synchronization that could have a multitude of consequences.

UNIBUS and Q22 Bus Device Support

12.3 Interrupt Dispatching in a UNIBUS/Q22 Bus System

Consider also an instance where the second device interrupts at a lower IPL, such as BIRQ4 (IPL 20). Although it essentially is executing below the IPL of the CPU (IPL 21), this device, too, may steal the interrupt grant. The break in synchronization in this instance will result in a reserved operand fault when the driver's interrupt service routine issues the REI instruction, as the VAX architecture does not permit an REI from a lower IPL to a higher IPL.

If the MicroVAX 3600-series system or MicroVAX II system employs the multilevel device interrupt dispatching option, VMS introduces special code in the interrupt transfer vector data structure that helps prevent violations of system synchronization resulting from an illegally configured Q22 bus (see Figure 12-8). This code, located at offset `VEC$L_RTINTD`, checks the device IPL of the interrupting peripheral against the IPL in the processor status longword (PSL) of the interrupted thread of code. If the device IPL is not greater than the IPL in the saved PSL, VMS generates an `ILLQBUSCFG` bugcheck, signifying that the Q22 bus is illegally configured.

12.3.4.2 Effects of Enabling Multilevel Device Interrupt Dispatching on Device Drivers

Before enabling multilevel device interrupts in a MicroVAX 3600-series system or MicroVAX II system, you should first ensure that all existing device drivers have been adapted according to the following guidelines:

- If the driver creates or accesses any secondary VEC data structures, it must take steps to initialize properly the multilevel device interrupt dispatching code (at offset `VEC$L_RTINTD` in the primary VEC structure) in these secondary structures. Failure to properly initialize these structures in the system can negate any performance increases expected as a result of enabling multilevel device interrupt dispatching.
- The device IPLs of certain Q22 bus devices may differ from those of corresponding UNIBUS devices. To ensure that the DPT specifies the correct device IPL, refer to the device's hardware user's guide.
- Wherever the driver implicitly refers to the longword in the VEC structure that contains the address of the interrupt service routine, it should explicitly use the symbol `VEC$L_ISR`. For example, you should replace any instance of `CRB$L_INTD+4` with `CRB$L_INTD+VEC$L_ISR`.
- If the driver assumes that the contents of the device's SCB vector (that is, the vector in the adapter dispatch table) always points to `CRB$L_INTD`, it must be appropriately modified to reflect the implementation of multilevel device interrupt dispatching. The transfer vector can legally point to any longword-aligned address between `CRB$L_INTD+VEC$L_RTINTD` and `CRB$L_INTD+VEC$L_INTD`. Because the MicroVAX 3600 series and MicroVAX II utilize direct-vector interrupt dispatching, the transfer vector will never point to `VEC$L_INTD+2`.

Although certain of the symbolic offsets defined in the data structure definition macro `$VECDEF` have negative values, driver code can uniformly refer to the contents of the VEC structure in the form `CRB$L_INTD+VEC$x_`*symbol*.

13 MASSBUS Device Support

The MASSBUS adapter (MBA) is the hardware interface between the backplane interconnect and MASSBUS storage devices. The MASSBUS is the communication path linking the MASSBUS adapter to the mass storage devices.

The MASSBUS adapter performs the following functions that allow communication between devices and memory:

- Mapping of virtual addresses to physical addresses
- Buffering of data for transfers between main memory and the MASSBUS
- Transfer of interrupts from MASSBUS devices to the backplane interconnect

A MASSBUS adapter supports any combination of up to eight device controllers. Typical MASSBUS controllers include the TM03 tape controller and the RP06, RM03, and RM80 disk controllers. Only one controller can transfer data over the MASSBUS at a time.

The TM03 tape controller supports up to eight tape drives. In contrast to tape controllers, there is a one-to-one relationship between a disk controller and its device; each controller supports only one disk drive. The VMS system interprets and maintains the I/O database differently, depending upon whether the controller is single or multiunit.

Each MASSBUS controller connected to a MASSBUS adapter is assigned a unit number in the range 0 to 7. The method of unit number assignment is controller specific, but you can obtain the number from either unit plugs or switch packs. In the case of a controller for several devices, the unit number is distinct from the subunit numbers assigned to the individual drives connected to the controller.

Figure 13-1 illustrates a possible MASSBUS configuration.

13.1 MASSBUS Adapter Registers

The MASSBUS adapter has three sets of registers:

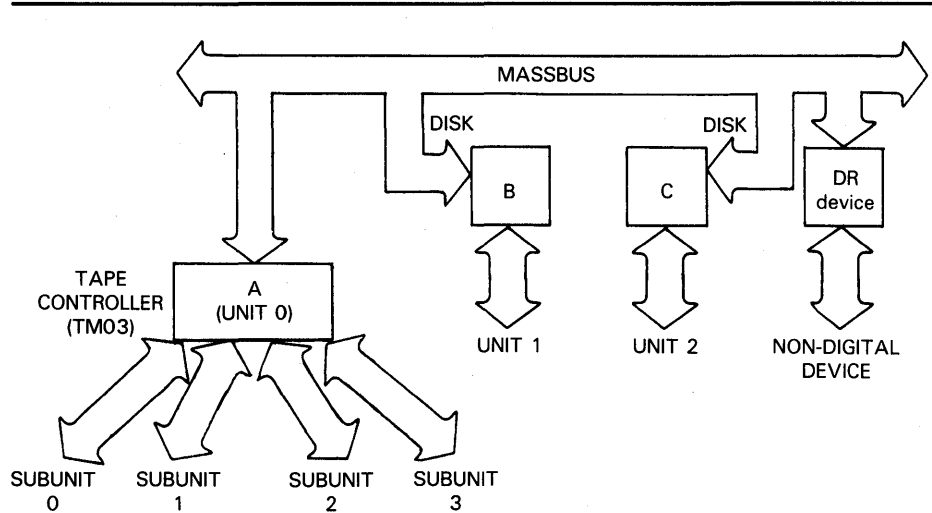
- The MASSBUS adapter's registers
- External registers for each device (controller) on the MASSBUS
- 256 map registers

To allow competing devices to share these resources, access to and modification of all MASSBUS adapter registers (internal, external, and map registers) are governed by certain rules and conventions. In particular, access to registers might, at times, require ownership of either the device controller

MASSBUS Device Support

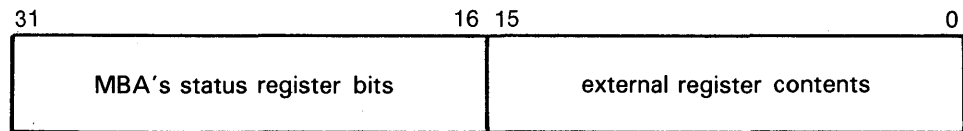
13.1 MASSBUS Adapter Registers

Figure 13-1 MASSBUS Configuration



ZK-939-82

Figure 13-2 MASSBUS External-Register Longword



ZK-1796-84

or the MASSBUS adapter itself, or both. Subsequent sections in this chapter discuss the methods of obtaining such ownership of these shared resources.

MASSBUS adapter external registers are device dependent and accessible whether or not the driver owns the MASSBUS adapter. However, in the case of multiunit MASSBUS adapter controllers, the driver might need to own the controller before it can gain access to a register.

MASSBUS adapter external registers are each 16 bits wide, but they must be accessed as longwords. When a driver reads an external register, the MASSBUS adapter concatenates the high-order 16 bits of the MBA's status register (one of the MBA's internal registers) with the contents of the specified external register. Figure 13-2 illustrates the resulting longword.

On a write to an external register, the MASSBUS adapter uses the low-order 16 bits of the longword source operand to update the external register.

MASSBUS adapter internal and map registers are 32 bits in length. They must be accessed as longwords or the processor will signal a machine check exception. The driver for a MASSBUS device must obtain exclusive ownership of the MASSBUS adapter before modifying any of the MBA's internal or map registers.

MASSBUS Device Support

13.1 MASSBUS Adapter Registers

Bits 21 through 30 of each of the MBA's map registers are reserved; they cannot be written. Use of the MBA's map registers is analogous to use of the UNIBUS adapter's map registers with the following exceptions:

- Because the MASSBUS can handle only one transfer at a time, ownership of the MASSBUS adapter implies ownership of all its map registers. Thus, the driver need not independently request map registers.
- The MBA's map registers do not contain a byte-offset field. The driver loads the full MASSBUS adapter virtual address, including the byte alignment, into the MASSBUS adapter virtual address register (MBA\$L_VAR, one of the MBA's internal registers) at the start of a data transfer. Use of the MBA\$L_VAR register is described in Section 13.1.1.
- The MBA's map registers do not contain a data path field; the MASSBUS adapter has a single data path, and ownership of the adapter implies ownership of the path. Thus, the driver need not allocate the data path independently.

13.1.1 Loading MASSBUS Adapter Registers

To prepare for a data transfer over the MASSBUS, the driver that owns the MASSBUS adapter uses the LOADMBA macro to load the MBA's map registers and associated internal registers. The LOADMBA macro invokes the subroutine IOC\$LOADMBAMAP, which performs the following steps:

- Determines the number of map registers needed to map the data area by adding the contents of UCB\$W_BCNT to UCB\$W_BOFF, adjusting the sum to the next even multiple of 512, and dividing the result by 512.
- Loads the specified number of map registers, beginning with map register 0, with the contents of the page-table entries to which UCB\$L_SVAPTE points. This step maps the data area for the transfer into the low portion of the MBA's virtual address space. The routine also loads the next map register beyond the number used to map the data area with zeros (an invalid map entry). This procedure stops the transfer should a hardware failure occur.
- Loads the MBA\$L_VAR register with the zero-extended contents of UCB\$W_BOFF. Because the first byte of the data area is located at offset UCB\$W_BOFF within the page of memory mapped by map register 0, the UCB\$W_BOFF contains the virtual address of the start of the data area in MASSBUS adapter virtual address space.
- Loads the complement (negative) of UCB\$W_BCNT into the MBA's byte-count register (MBA\$L_BCR).

Note that if a driver is to perform a data transfer in the reverse direction (for example, read reverse on a tape), it must modify the contents of the MBA\$L_VAR, as established by IOC\$LOADMBAMAP, so that it points to the last byte of the data area. This is done by adding one less than the contents of UCB\$W_BCNT to the contents of the MBA\$L_VAR register.

During the progress of a data transfer over the MASSBUS, the MBA\$L_VAR register is continuously updated so that it points to the current position in the data area. The *VAX Hardware Handbook* illustrates the mapping of the contents of the MBA\$L_VAR register into physical memory.

MASSBUS Device Support

13.1 MASSBUS Adapter Registers

13.1.2 MASSBUS Adapter Registers and Offsets

During system initialization, VMS builds an adapter control block (ADP), a channel request block (CRB), and an interrupt dispatch block (IDB) for each MASSBUS adapter. The system also allocates 4KB of system virtual address space for the adapter's register I/O space. The base of this I/O register virtual address space is placed in `IDB$L_CSR`. Thus, you can access MASSBUS adapter registers using the base register virtual address plus some offset. The `$MBADEF` macro defines the offsets for MASSBUS adapter registers. The major symbols defined by this macro are shown in Table 13-1.

Table 13-1 Major Offsets Defined by \$MBADEF

| Symbol | MBA Register Name | Hex Offset |
|-------------------------|----------------------------|------------|
| <code>MBA\$L_CSR</code> | Configuration register | 0 |
| <code>MBA\$L_CR</code> | Control register | 4 |
| <code>MBA\$L_SR</code> | Status register | 8 |
| <code>MBA\$L_VAR</code> | Virtual-address register | C |
| <code>MBA\$L_BCR</code> | Byte-count register | 10 |
| <code>MBA\$L_DR</code> | Diagnostic register | 14 |
| <code>MBA\$L_SMR</code> | Selected map register | 18 |
| <code>MBA\$L_CAR</code> | Command-address register | 1C |
| <code>MBA\$L_ERB</code> | External register base | 400 |
| <code>MBA\$L_AS</code> | Attention-summary register | 414 |
| <code>MBA\$L_MAP</code> | Base of map registers | 800 |

The MASSBUS adapter's internal registers occupy the low-order 1024 bytes of address space even though there are only eight internal MBA registers. Beyond the internal registers, there are eight blocks of 32 longwords (128 bytes) each, one block for each of the eight device controllers that can be connected to a single MASSBUS adapter. Each of these blocks provides space for the device registers of each controller. Beyond the device-register space is the area reserved for the MASSBUS adapter's 256 map registers.

Figure 13-3 illustrates the relative positions of the MASSBUS adapter's registers and the values device drivers use to gain access to them. The base address of the MASSBUS adapter's address space, stored in `IDB$L_CSR`, is the address of the first of the MASSBUS adapter's internal registers. `IDB$L_CSR` represents the internal register's virtual location, while the `MBA$L_` symbols represent register values as defined by `$MBADEF`. Note that the MASSBUS adapter's register space occupies only the first 3KB out of the 8KB allotted to physical I/O address space. However, by convention, VMS allocates 4KB of virtual addresses to each MASSBUS adapter.

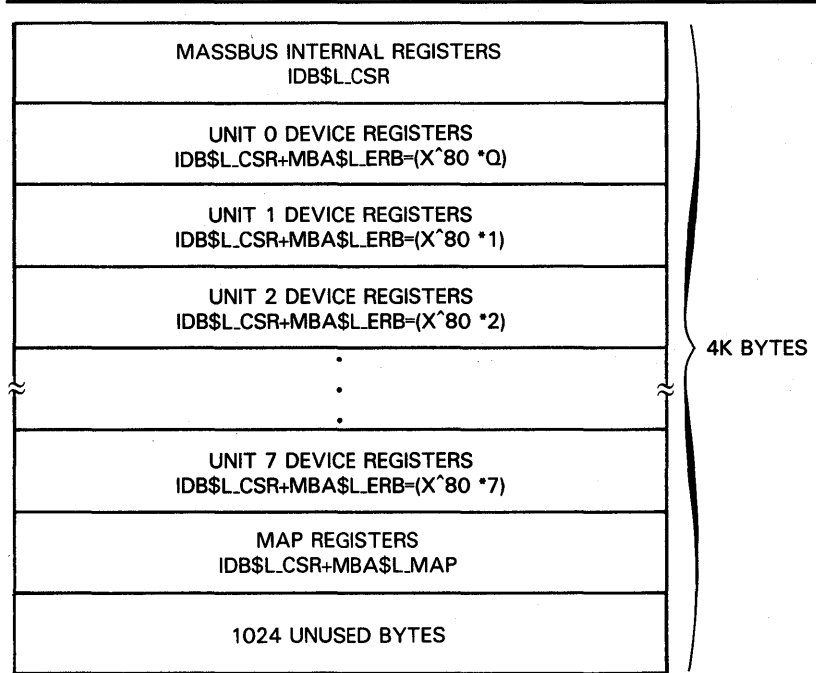
To address a map register in the MASSBUS adapter, the driver constructs the following address:

$$\text{IDB\$L_CSR} + \text{MBA\$L_MAP} + \text{map-register-index}$$

MASSBUS Device Support

13.1 MASSBUS Adapter Registers

Figure 13-3 Location of MASSBUS Registers in Physical Address Space



ZK-940-82

To address a device register, the driver constructs the following address:

$$\text{IDB\$L_CSR} + \text{MBA\$L_ERB} + (\text{unit-number} * 80_{16}) + \text{register-displacement}$$

An individual driver should define offsets for the registers of its device. During execution, the driver computes a register address by summing the MBA's starting virtual address (the contents of IDB\$L_CSR), MBA\$L_ERB, the unit number of the device controller multiplied by 80₁₆, and the offset of the specified register.

The attention-summary register (MBA\$L_AS), as shown in Table 13-1, appears to reside within the external-register space reserved for MASSBUS adapter controller 0. Actually, the attention-summary register is a composite register. Each of the MASSBUS adapter's controllers contributes one bit of information to the register. This composite register appears in each of the eight device register spaces at offset 10₁₆ from the base of the device registers for that device. Thus, MBA\$L_AS can be defined as any of the values 410₁₆, 490₁₆, 510₁₆, 590₁₆, and so on. For convenience, it has been defined as 410₁₆.

MASSBUS Device Support

13.1 MASSBUS Adapter Registers

13.1.3 Modifying MASSBUS Adapter Registers

The driver for a MASSBUS device must obtain ownership of the MBA before modifying any of the MBA's internal registers or map registers. A driver obtains ownership of the MBA by invoking either the REQCHAN macro or the REQSCHAN macro, depending on whether the device is connected to a single-unit MASSBUS controller or a multiunit MASSBUS controller.

For dedicated controllers, invoke the REQCHAN macro. Because the controller is dedicated to its single device, there is never any contention for the controller.

For multiunit devices, however, invoke the REQSCHAN macro to obtain MBA ownership because several devices can share the controller, and so must contend for its use. The controller for several devices relegates the MASSBUS adapter to a secondary position. Thus, for multiunit controllers, invoke REQCHAN to gain ownership of the controller, and invoke REQSCHAN to obtain the MASSBUS adapter.

13.2 I/O Database for MASSBUS Devices

During initialization, the system creates an ADP, a CRB, and an IDB for each MASSBUS adapter included in the configuration. The driver-loading procedure subsequently builds additional data structures for each device controller connected to a MASSBUS adapter. The type of structure created depends upon whether the device controller is a dedicated controller or the controller of several devices.

The system builds a unit control block (UCB) for each single-unit controller. Figure 13-4 illustrates the I/O database for a MASSBUS adapter with one dedicated controller attached to it. Note that the ADP, CRB, and IDB all correspond to the MASSBUS adapter and can logically be considered a single, extended data block. The UCB corresponds to the device/controller pair. Because of the one-to-one correspondence between a dedicated controller and its device, the system does not need to distinguish between the two and thus does not maintain separate data blocks for each piece of hardware.

A controller of several devices, however, requires separate data structures for the controller and each of its subunits (devices). The driver-loading procedure builds a CRB/IDB pair for the controller, as well as a UCB for each subunit. Figure 13-5 shows the I/O database created for a MASSBUS adapter with one disk unit and two tape units.

Figure 13-5 does not include several pointers used in interrupt dispatching. In particular, the IDB associated with the MASSBUS adapter maintains an array of up to eight longwords that point to the data structures associated with the eight possible MASSBUS controllers attached to the MASSBUS.

For dedicated controllers, the IDB longword points to the device's UCB; whereas, for a controller for several devices, the longword (or longwords) points to a field within the CRB associated with the controller. The low bit of this longword, when set, indicates a multiunit vector. The software checks this bit to determine whether the longword points to a single UCB or a multiunit CRB.

MASSBUS Device Support

13.2 I/O Database for MASSBUS Devices

Figure 13-4 I/O Database for MASSBUS Disk Unit

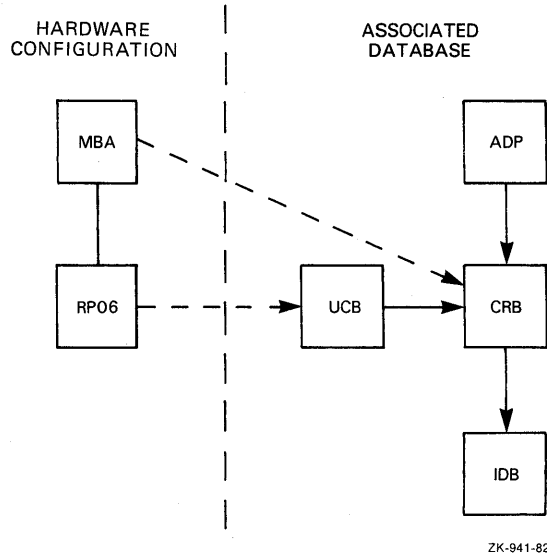
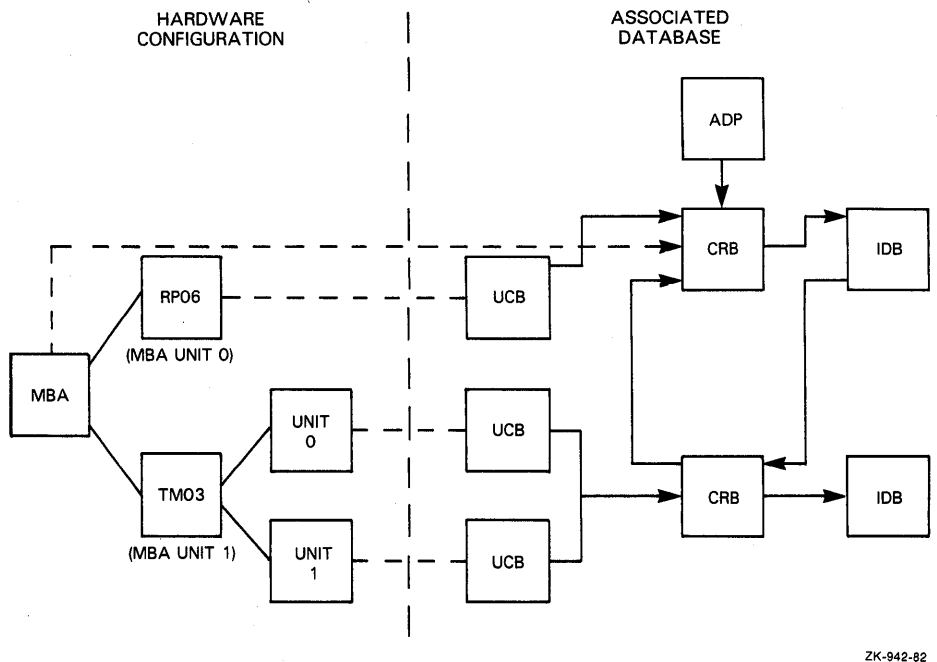


Figure 13-5 I/O Database for MASSBUS Disk and Tape Units

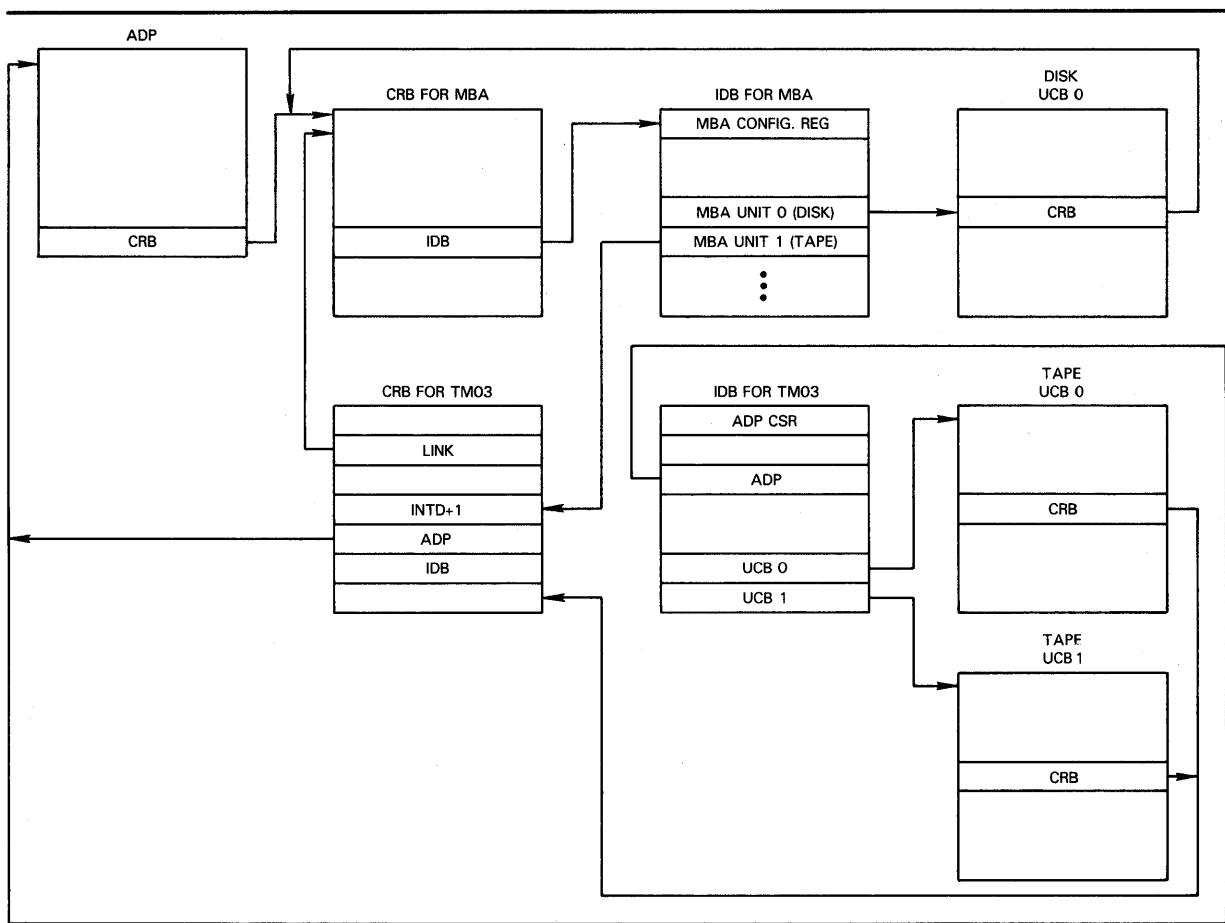


Also not pictured in Figure 13-5 is how multiunit IDBs also maintain an array of longwords. Each longword points to the individual UCBs for the units attached to the controller. Figure 13-6 illustrates in more detail the set of I/O data structures for the MASSBUS adapter and its devices.

MASSBUS Device Support

13.2 I/O Database for MASSBUS Devices

Figure 13-6 I/O Data Structures Used in Dispatching a MASSBUS Device Interrupt



ZK-943-82

13.3 MASSBUS Adapter Operations

The MASSBUS accepts two kinds of operations: data transfer operations and nondata transfer operations. Data transfer operations require the use of MASSBUS adapter shared resources, while nondata transfers do not.

Before a driver can activate a data transfer operation on the MASSBUS, the driver must request and receive ownership of the MASSBUS adapter on behalf of the device unit. However, drivers must not initiate nondata transfer operations while they have control of the MASSBUS adapter. Section 13.4.1 explains this statement further.

The MASSBUS adapter generates interrupts when data transfers terminate and when attention conditions arise on devices. When an interrupt occurs on the MASSBUS adapter, the MASSBUS adapter's interrupt dispatcher determines whether the interrupt is for a data transfer or an attention condition.

Data transfer interrupts occur when a data transfer either completes or is aborted. When the interrupt occurs, the MBA's status register (MBA\$L_SR) contains information about the condition that caused the interrupt.

MASSBUS Device Support

13.3 MASSBUS Adapter Operations

Attention interrupts occur when nondata transfers on MASSBUS devices terminate, or when the device undergoes an exceptional condition, such as coming on line.

The MASSBUS adapter's attention-summary register controls attention-interrupt handling. This register contains eight bits of data, one for each of the eight possible controllers that can be connected to the MASSBUS adapter. When a device incurs an attention condition, the hardware sets the corresponding bit in the attention-summary register and generates a MASSBUS adapter interrupt.

If the attention condition occurs while a data transfer operation for another device is in progress, the hardware sets the bit in the attention-summary register but suppresses the attention interrupt. The interrupt generated when the data transfer is completed allows the MASSBUS adapter's interrupt dispatcher to gain control, handle the data transfer interrupt, check the attention-summary register and then invoke the proper driver to handle the attention condition.

13.4 MASSBUS Adapter's Interrupt Dispatching

When interrupts occur on the MASSBUS adapter, the MASSBUS adapter's interrupt dispatcher gains control. This routine first determines whether the interrupt is the result of a data transfer or an attention condition. The routine checks to see if the MASSBUS adapter is owned and, if so, by whom.

13.4.1 Checking for MASSBUS Adapter Ownership

There are two conditions by which the interrupt dispatcher can determine that the interrupt is an attention interrupt:

- If the MASSBUS adapter is not owned
- If the MASSBUS adapter is owned, but the owner is not expecting an interrupt (UCB\$V_INT in UCB\$L_STS is clear)

When the MASSBUS adapter is owned and the owner expects an interrupt, the interrupt is assumed to be the result of a data transfer operation.

As mentioned earlier, a driver must not initiate nondata transfers on the MASSBUS adapter while it owns the adapter. For example, consider a MASSBUS adapter attached to two disk units, A and B. Disk A is performing an IO\$_SEEK (a nondata transfer operation that completes fairly quickly), while at the same time, disk B is performing an IO\$_RECAL operation (a nondata transfer operation that takes about 0.5 seconds to complete).

The driver for disk A correctly initiates its operation without obtaining possession of the MASSBUS adapter channel, but the disk B driver initiates its operation while it owns the MASSBUS adapter. Both of these operations, upon completion, set the bit in the attention-summary register that corresponds to their respective drive units, and initiate an interrupt. We will assume that disk A's IO\$_SEEK is completed first. The operation sets disk A's bit in the attention-summary register and generates the MASSBUS adapter's interrupt.

MASSBUS Device Support

13.4 MASSBUS Adapter's Interrupt Dispatching

The MASSBUS adapter's interrupt dispatcher finds that the adapter is owned, and that the owner is expecting an interrupt. Therefore, the interrupt dispatcher incorrectly assumes that it is handling a data transfer interrupt, and, moreover, that this interrupt is the one for which the owner of the MBA is waiting.

As a result, the MASSBUS adapter's interrupt dispatcher returns control, through the fork block in the MASSBUS adapter owner's UCB, to the driver for disk B, even though disk B's operation has not completed. The disk B driver will now incorrectly assume that the device has completed its operation, which can cause serious problems.

13.4.2 Dispatching a Device Interrupt

Once the MASSBUS adapter's interrupt dispatcher determines the type of interrupt, it dispatches the interrupt to the driver. The interrupt dispatcher handles attention interrupts and data transfer interrupts in the same way, with one exception: on an attention interrupt, the interrupt dispatcher clears the MASSBUS adapter's status register (MBA\$L_SR) before dispatching the interrupt to the driver. The status register contains information used only in data transfer interrupt dispatching.

How the interrupt dispatcher dispatches the interrupt to the driver differs depending on the type of controller.

The MASSBUS adapter's interrupt dispatcher handles a solicited interrupt on a dedicated controller by transferring control to the driver through the fork block in the UCB. On unsolicited interrupts on dedicated controllers, the interrupt dispatcher calls the driver's unsolicited interrupt service routine.

On dedicated controllers, the MASSBUS adapter's interrupt dispatcher always clears the attention bit in the attention-summary register before it calls back the driver after an interrupt.

Dispatching interrupts to the driver of a device that shares its controller with several other devices differs in two ways from dispatching interrupts to the driver of a device with a dedicated controller.

First, the interrupt dispatcher never clears the attention bit. This task is left to the driver because some controllers that control more than one device use this bit to synchronize their activities, and guarantee the integrity of device registers only while the bit is set. If the interrupt dispatcher clears the bit before returning control to the driver, the driver can no longer rely on the contents of the device's registers.

Second, a controller that controls several devices needs another interrupt dispatcher to handle simultaneous requests from its several subunits. This second-level interrupt dispatcher resides in the driver. After an interrupt, the MASSBUS adapter's interrupt dispatcher indirectly calls this second driver's interrupt dispatcher using code in the controller's CRB. The driver-loading procedure installs this code when it establishes the I/O database.

MASSBUS Device Support

13.5 Special Considerations for MASSBUS Device Drivers

13.5 Special Considerations for MASSBUS Device Drivers

MASSBUS adapter considerations affect a driver's device unit initialization routine, start-I/O routines and, for multiunit controllers only, the driver's use of the DPTAB macro. MBA considerations also affect interrupt handling, as described in Section 13.4.2. The next sections in this chapter discuss programming details for writing a MASSBUS device driver.

13.5.1 Unit Initialization Routine

All drivers for MASSBUS adapter devices initialize two fields in the UCB (as well as initializing device-specific fields): UCB\$B_SLAVE and UCB\$B_SLAVE+1. The first of these fields should contain the controller's MASSBUS adapter unit number, which marks the controller's position on the MASSBUS adapter. The second of these contains the offset, in longwords, from the start of the MASSBUS adapter's external registers to this controller's device registers. The value of this longword offset is always 32 times the unit number of the controller.

Initialization of a device attached to a dedicated controller is simple because the device unit number and the controller position number on the MASSBUS adapter are always equal. To initialize the field UCB\$B_SLAVE, copy to it the contents of UCB\$W_UNIT. To initialize UCB\$B_SLAVE+1, multiply the contents of UCB\$W_UNIT by 32. The driver's fork process or interrupt service routine later uses this information to compute a pointer to this device's registers. By convention, R4 points to the MASSBUS adapter configuration register, and R5 points to the UCB of this device.

Thus, the following two instructions cause R3 to point to the device registers during normal system operation:

```
MOVZBL    UCB$B_SLAVE+1(R5),R3
MOVAL     MBA$L_ERB(R4)[R3],R3
```

For devices connected to a controller that controls several devices, determination of the controller's MBA position is more complex. When the unit initialization routine is invoked, the following values are in the following registers:

R3 Address of controller's device registers
R4 Address of the MBA's configuration register
R5 Address of device's UCB

The driver computes the MBA position of the controller by using R3 and R4 to determine the number of bytes from the start of the MBA's external registers to the start of the device's device registers. The difference, when divided by 128, is the controller's MBA position number.

MASSBUS Device Support

13.5 Special Considerations for MASSBUS Device Drivers

13.5.2 The MASSBUS Adapter and the I/O Database

The UCB of a device connected to a single-unit controller, at offset `UCB$L_CRB`, contains the address of the MASSBUS adapter's CRB. This CRB in turn contains, at offset `CRB$L_INTD+VEC$L_IDB`, the address of the MASSBUS IDB. This IDB points to the base address of the MASSBUS adapter registers at offset `IDB$L_CSR`.

A controller that controls several devices maintains a more complicated I/O database. The device UCB, at offset `UCB$L_CRB`, points to the controller's CRB, and this structure points to the CRB for the MASSBUS adapter at offset `CRB$L_LINK`. Also, the controller's CRB points to its own IDB at offset `CRB$L_INTD+VEC$L_IDB`. This IDB points to the controller's device registers at offset `IDB$L_CSR`.

Thus, the UCB for a device always points to that device's primary CRB, whether it is the MASSBUS adapter's CRB or the controller's CRB. The primary CRB points to the secondary CRB, if one exists for the device.

Figure 13-6 shows these relationships among I/O data structures.

13.5.3 Start-I/O Routine

Depending on the function being executed, the start-I/O routine for a MASSBUS device performs all or some of the following tasks:

- Requests, if necessary, controller data channel(s) as described in Section 13.5.3.1
- Clears errors on the MASSBUS adapter by placing the value -1 into the MBA's status register; this is a write-ones-to-clear register (MASSBUS device registers and the MBA's registers are all longwords)
- Invokes the `LOADMBA` macro to load the MBA's map registers as described in Section 13.5.3.2
- Loads device registers to start the function
- Waits for a device interrupt or timeout
- Releases, if necessary, controller data channel(s) as described in Section 13.5.3.3
- Finishes the request like other drivers

13.5.3.1 Requesting Controller Data Channels

Device drivers for MASSBUS devices must request and receive ownership of the MASSBUS adapter channel before loading the MBA's internal registers or map registers. In addition, drivers for devices connected to multiunit controllers must obtain ownership of the controller channel before modifying the contents of controller registers that can be shared among the units connected to the controller.

Drivers for dedicated controllers must request ownership of the MASSBUS adapter channel by invoking the macro `REQPCAN`.

MASSBUS Device Support

13.5 Special Considerations for MASSBUS Device Drivers

Device drivers for controllers that control several devices invoke the REQCHAN macro when the operation requires ownership of only the primary channel (the controller's channel). However, if the operation requires ownership of both primary and secondary channels (a data transfer operation), the driver must first obtain the controller channel and then request the MASSBUS adapter channel by invoking the REQSCHAN macro.

Again, the driver needs ownership of both channels only when performing a data transfer, and must release the channels before initiating a nondata transfer. Thus, a driver must obtain ownership of the MASSBUS adapter channel some time before initiating a data transfer and must either not own the channel or release such ownership before it invokes the WFIKPCH macro, or issue the WFIRLCH macro, following the start of a nondata transfer operation.

13.5.3.2 Loading Map Registers

MASSBUS device drivers invoke the LOADMBA macro before they initiate a data transfer, to load the MBA's map registers, the MBA's virtual-address register (MBA\$L_VAR), and the MBA's byte-count register (MBA\$L_BCR). Drivers cannot modify these registers during a transfer. The LOADMBA macro expects the following register contents:

- The address of the MBA's configuration register (MBA\$L_CSR) in R4
- The address of the device UCB in R5

LOADMBA preserves the contents of R3 but modifies R0 through R2. The macro performs the following steps:

- 1 Uses the contents of UCB\$W_BCNT and UCB\$W_BOFF to determine the number of pages that contain pieces of the I/O buffer.
- 2 Beginning with the page-table entry to which UCB\$L_SVAPTE points and continuing for the number of page-table entries determined in step 1, copies the page-frame numbers from the page-table entries to the corresponding map registers, starting at map register 0.
- 3 Ensures that the valid bit is clear in the map register that immediately follows the last map register loaded with a PFN. This prevents a hardware fault or prefetch from modifying memory.
- 4 Moves the negative value of the transfer byte count (UCB\$W_BCNT) into the MBA's byte-count register (MBA\$L_BCR).
- 5 Moves the byte offset in the first page of the transfer (UCB\$W_BOFF) into the MBA's virtual-address register (MBA\$L_VAR).
- 6 Returns to the start-I/O routine that invoked it.

If the I/O operation about to be initiated by the driver is a reverse operation (a read-reverse on tape), the driver must modify the contents of the MBA's virtual-address register set up by LOADMBA. Because reverse operations access the I/O buffer from its highest address through its lowest address, the value to be loaded into the MBA's virtual-address register must be the virtual address, in MBA's virtual memory, of the last byte of the buffer. This number is equal to one less than the sum of the contents of UCB\$W_BOFF and UCB\$W_BCNT.

MASSBUS Device Support

13.5 Special Considerations for MASSBUS Device Drivers

13.5.3.3 Releasing Controller Data Channels

The driver releases the controller data channels by invoking the RELCHAN macro. RELCHAN releases all controller channels (both primary and secondary) currently owned by the device. To release only the secondary channel and retain ownership of the primary channel, a driver can invoke the RELSCHAN macro.

13.5.4 DPTAB Macro

The device driver for a MASSBUS device that shares its controller with other devices must set the DPT\$V_SUBCNTRL bit in the **flags** argument of the DPTAB macro. Setting this bit causes the driver-loading procedure to create a second CRB and an IDB for the controller.

13.6 Interrupt Service Routines for MASSBUS Devices

The MASSBUS interrupt dispatcher (MBA\$INT) gains control when it receives an interrupt from the MASSBUS adapter. Because data transfers in progress suppress attention interrupts on the MASSBUS adapter, and because several devices can request attention simultaneously, several device drivers might need to be informed of the interrupt.

MBA\$INT determines which drivers should be invoked as a result of the interrupt and then passes control to these drivers. For data transfer interrupts, MBA\$INT preserves the value contained in the MBA's status register at the time of the interrupt so that the driver can have access to this value.

For I/O operations that involve no data transfer, MBA\$INT clears this register before invoking the driver. MBA\$INT only preserves the contents of registers R2 through R5. Drivers that use other registers must save the contents of those registers, and must restore them before exiting from the interrupt service routine.

13.6.1 Transferring Control to the Interrupt Service Routine

The method by which MBA\$INT invokes a driver depends upon whether the driver serves a device connected to a dedicated controller or a device that shares its controller with several other devices. Furthermore, if the device is connected to a dedicated controller, the method of transfer from MBA\$INT to the driver depends upon whether or not the interrupt is expected.

For a device on a dedicated controller whose driver is expecting an interrupt, MBA\$INT restores the driver context saved in the UCB fork block and transfers control (using a JSB instruction) to the instruction that follows the wait-for-interrupt instruction.

For a device on a dedicated controller whose driver is not expecting interrupts, MBA\$INT obtains the address of the driver's unsolicited interrupt service routine from the driver dispatch table and calls the routine.

For a device that shares its controller with several other devices, MBA\$INT transfers control to the driver's interrupt service routine by simulating a direct transfer, through an interrupt vector, to the controller's CRB. The CRB contains code that transfers control to the interrupt service routine.

MASSBUS Device Support

13.6 Interrupt Service Routines for MASSBUS Devices

MBA\$INT first pushes the processor status longword (PSL) onto the stack. The routine then calls (with a JSB instruction that leaves an address within MBA\$INT on the stack) the code within the CRB. This code contains the following sequence of instructions, where XX\$INT is the address of the interrupt service routine and XX\$IDB is the address of the controller's IDB:

```
PUSHR    #~M<R2,R3,R4,R5>
JSB      XX$INT
.LONG    XX$IDB
```

The execution of the previous instruction sequence, plus the instructions executed by MBA\$INT (the pushing of the PSL onto the stack and the JSB), places a simulated interrupt frame onto the stack, including a saved PSL, a saved PC, saved registers, and a pointer to an address in the IDB.

13.6.2 Returning Control to MBA\$INT

The way in which a driver returns control to MBA\$INT depends on the way in which MBA\$INT invoked it. Drivers for dedicated controller devices return to MBA\$INT through an RSB instruction, although the RSB can execute as a result of the driver's invoking the IOFORK macro.

Drivers of devices that share a controller return control to MBA\$INT by removing the indirect pointer to the IDB from the top of the stack, restoring registers R2 through R5, and executing an REI instruction. This sequence, executed within the driver's interrupt service routine, eliminates the simulated interrupt frame from the stack before returning to MBA\$INT.

13.6.3 Considerations for Interrupt Service Routines

Drivers for dedicated controller devices attached to the MASSBUS do not have interrupt service routines. Instead, MBA\$INT handles all the functions that a driver interrupt service routine normally provides.

Drivers of devices that share a controller on the MASSBUS must have their own interrupt service routines. In general, these routines perform the same functions as the interrupt service routines for UNIBUS and Q22 bus devices (discussed in Chapter 9). However, the two types of drivers diverge in two areas.

One difference between UNIBUS/Q22 bus and MASSBUS drivers concerns the number of registers saved by the interrupt service routine. When the interrupt dispatcher transfers service control to a MASSBUS driver interrupt service routine, registers R2 through R5 are pushed onto the stack. UNIBUS/Q22 bus drivers save R0 through R5.

After handling an interrupt, both MASSBUS and UNIBUS/Q22 bus driver interrupt service routines execute an REI instruction. For UNIBUS/Q22 bus devices, the REI dismisses a real interrupt, whereas the MASSBUS driver's REI returns control to MBA\$INT.

14 Generic VAXBI Device Support

This chapter provides information needed to write and load a device driver for a non-DIGITAL-supplied device attached to the VAXBI bus. VMS provides special support for such devices in the system initialization routines for the VAX 8200/8250/8300/8350, VAX 8530/8550/8700/8800/8830/8840, and VAX 6200-series systems. Because of the many and varied implementations of VAXBI devices, however, VMS support must of necessity be very general. Some devices may more fully utilize the VAXBI interface than others; a device may incorporate its interface initialization logic in microcode, whereas another may defer initialization to code in its driver.

The *VAXBI Options Handbook* includes a description and guidelines for possible VAXBI device implementations. Refer to that manual for further discussion of all VAXBI topics discussed in brief in Section 14.2 and elsewhere in this chapter.

14.1 Overview

A VAXBI device driver refers to the same data structures and contains the same routines as a traditional VMS driver. A VAXBI device driver deviates from the traditional VMS driver almost exclusively in code that initializes the VAXBI interface or supports direct-memory-access (DMA) transfers for devices that address memory across the VAXBI bus. Section 14.4 discusses tasks that drivers of various VAXBI devices may perform in their initialization routines to supplement VMS initialization and that initialization performed by device microcode. Section 14.5 contains a general discussion of how some VAXBI devices and their drivers manage DMA transactions.

Section 14.3 describes those data structures the VMS adapter initialization routine creates and prepares for a generic VAXBI device, while Section 14.8 discusses the method by which its driver can be loaded into the operating system. The final section of this chapter provides reference material and includes a description of the backplane interconnect interface chip (BIIC) registers.

14.2 VAXBI Concepts

The VAXBI serves as the I/O bus for the VAX 8200/8250/8300/8350, VAX 8530/8550/8700/8800/8830/8840, and VAX 6200-series systems (see Figure 1-3).¹ The VAX 8200/8250/8300/8350 systems can have a single VAXBI; the VAX 8530/8550/8700/8800/8830/8840 and VAX 6200-series systems can have multiple VAXBI buses.

Each location on a VAXBI bus is called a *node*. A single VAXBI bus can service 16 nodes. In the case of the VAX 8200/8250/8300/8350 systems, these nodes can be processors, memory, and adapters; the VAX 8530/8550/8700/8800/8830/8840 and VAX 6200-series systems permit only

¹ The VAXBI is also the system bus for the VAX 8200/8250/8300/8350 systems.

Generic VAXBI Device Support

14.2 VAXBI Concepts

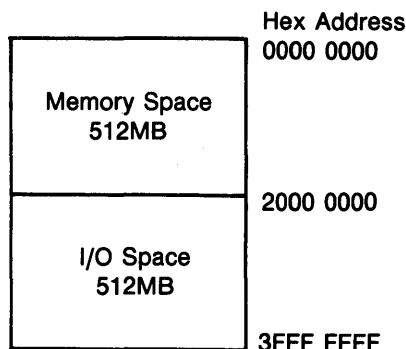
adapters to be attached to the VAXBI bus.² A node receives its *node ID*, a number from 0 to 15, from a plug on the VAXBI backplane slot into which the node module is inserted.

An *adapter* is a node that connects other buses, communication lines, and peripheral devices to the VAXBI bus. This chapter uses the term *device* to refer to a device or combination of devices serviced by a single adapter or controller.

14.2.1 VAXBI Address Space

Each VAXBI bus supports 30-bit addressing capability. This gigabyte of physical address space is split equally between memory and I/O address space, as shown in Figure 14-1.

Figure 14-1 VAXBI Address Space



ZK-5541-86

All memory locations on a VAXBI bus are addressed using physical addresses in VAXBI memory space (from 00000000_{16} through $1FFFFFFF_{16}$). A VAXBI device that accesses memory directly (or indirectly through a memory-interconnect-to-VAXBI adapter), or its driver, must perform virtual-to-physical translation before transmitting a memory address on the bus. (See Section 14.5 for additional information).

VAXBI I/O address space (physical addresses 20000000_{16} through $3FFFFFFF_{16}$) is partitioned as illustrated in Figure 14-2. Figure 14-3 shows the structure of an I/O-space address.

² For VAX 8530/8550/8700/8800, VAX 8830/8840, and VAX 6200-series systems, the memory-interconnect-to-VAXBI adapter (NBI PBI, or DWMB) or, more specifically, the NBIB, PBIB or DWMB/A/B resides at a node on a VAXBI bus, monitoring and controlling transactions to the memory interconnect (NMI, NMIs, or XMI) where the processors and memory reside.

Generic VAXBI Device Support

14.2 VAXBI Concepts

Figure 14–2 Description of VAXBI I/O Address Space

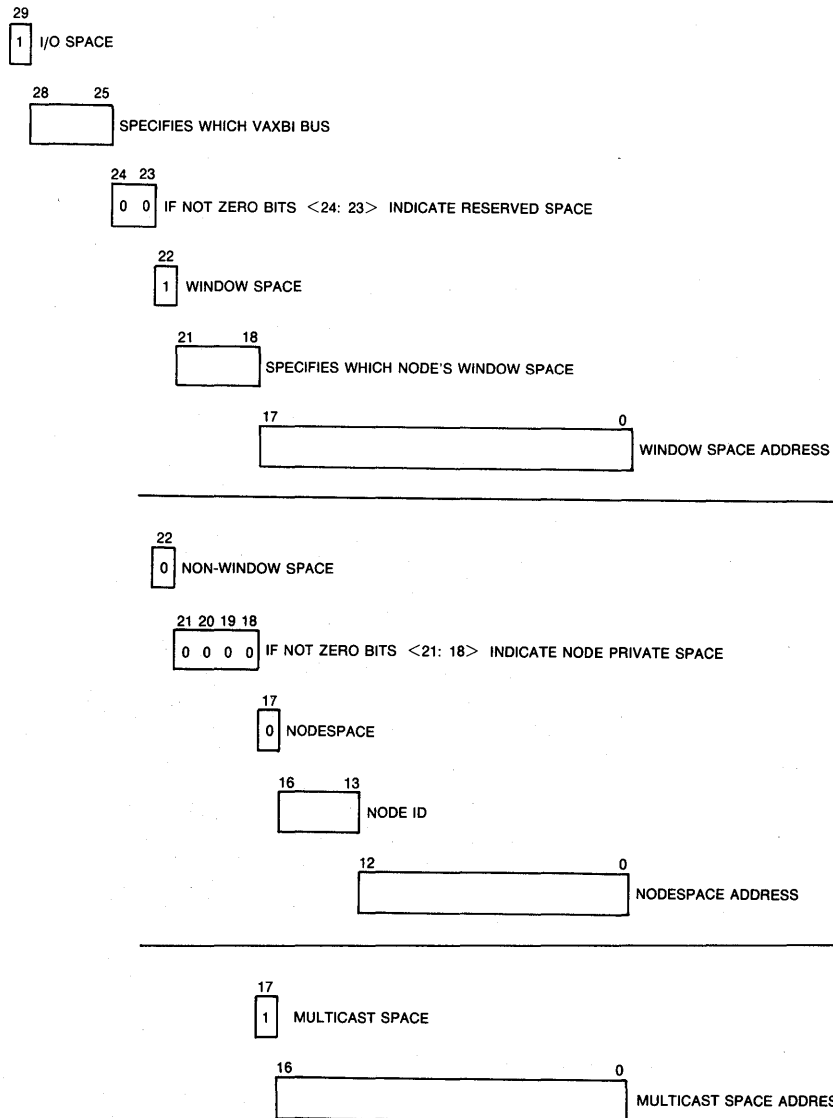
| | Hex Address |
|---|------------------------|
| Node 0 Nodespace (8KB) | 2000 0000 |
| ⋮ | 2000 1FFF |
| Node 15 Nodespace (8KB) | 2001 E000 |
| Multicast Space (128KB) | 2001 FFFF 2002 0000 |
| Node Private Space (3.75MB) | 2003 FFFF 2004 0000 |
| Node 0 Window Space (256KB) | 203F FFFF 2040 0000 |
| ⋮ | 2043 FFFF |
| 15 Window Space (256KB) | 207C 0000 207F FFFF |
| RESERVED | |
| RESERVED (for multiple VAXBI systems) (480MB) | 3FFF FFFF |

ZK-5542-86

Generic VAXBI Device Support

14.2 VAXBI Concepts

Figure 14-3 Physical Addresses in VAXBI I/O Address Space



ZK-5543-86

Generic VAXBI Device Support

14.2 VAXBI Concepts

As shown in Figures 14–2 and 14–3, VAXBI architecture grants each of the 16 nodes on a VAXBI bus two discrete sections in I/O address space.

| | |
|--------------|--|
| Node space | <p>An 8KB block of addresses consisting of 256 bytes of <i>BIIC CSR space</i>, followed by <i>user interface CSR space</i>. A device can access the control and status registers (CSRs) of its backplane interconnect interface chip by using BIIC CSR space addresses. Device-specific registers reside in user interface CSR space.</p> <p>Because the VMS adapter initialization routine virtually maps node space for each VAXBI node on each VAXBI bus, a device driver can access both BIIC registers and device registers using virtual addresses. (See Sections 14.4 and 14.5 for a discussion of driver access to registers.)</p> |
| Window space | <p>A 256KB block used by a VAXBI adapter to map an I/O transfer to a target bus. Because VMS does not automatically map window space to virtual addresses, a driver that manipulates addresses in window space must itself allocate and fill sufficient system page-table entries for the range of its window space addresses. (See Section 14.4.)</p> |

Note that *node private space* contains locations used for the storage of bootstrap firmware and software. VAXBI nodes are not permitted to issue or respond to VAXBI transactions targeting locations in node private space.

14.2.2 Backplane Interconnect Interface Chip (BIIC)

The *backplane interconnect interface chip* (BIIC) serves as the primary interface between the VAXBI bus and the user interface logic of a node. The BIIC supplies the logic necessary for a node to initiate and respond to transactions on the VAXBI bus, arbitrate bus ownership, send and receive interrupt requests, and monitor bus errors.

A node can enable, control, and monitor such activities by accessing the set of BIIC registers located in the first 256 bytes of its node space. Because the VMS adapter initialization routine virtually maps node space addresses, drivers for VAXBI devices can use virtual addresses to access BIIC registers. In addition, given the virtual address of the base of a device's node space, a driver can use the symbolic offsets, masks, and bit fields defined by the VMS macro `$BIICDEF` (in `SYS$LIBRARY:LIB.MLB`). Table 14–1 describes these symbols.

14.3 Initialization Performed by VMS

During the phase of system initialization known as adapter initialization VMS performs a set of system-specific tasks to identify and configure each device it discovers at each of the 16 nodes on each VAXBI bus in the system configuration.

Generic VAXBI Device Support

14.3 Initialization Performed by VMS

The adapter initialization module configures DIGITAL-supplied and non-DIGITAL-supplied devices alike, performing the following activities as part of its initialization cycle:

- 1 Tests for the presence of a device at the node by issuing a MOVL instruction, the target of which is a system virtual address temporarily mapped to the first longword of its node space. If this instruction is successful, it returns the contents of the BIIC Device Type Register of the addressed node to the processor.³
- 2 Records the contents of the low 16 bits of the BIIC Device Type Register, plus an I/O bus identifier in the slot in the CONFREGL array that corresponds to the VAXBI bus and node at which it found the device,⁴ and compares this value against a table of recognized device types.
- 3 If it *recognizes* the device, maps the number of pages specified in the table for the device type, and places the system virtual address of the base of the mapped node space in the slot in the SBICONF array that corresponds to the VAXBI bus and node at which it found the device.⁵

If it does *not* recognize the device, maps the entire 8KB of the node's node space into VMS virtual address space by allocating 16 system page-table entries (SPTEs) and associating them with the 16 page-frame numbers (PFNs) of the physical addresses assigned to this node's node space on this VAXBI bus. The adapter initialization module then saves the base system virtual address of the resulting 8KB range in the longword slot corresponding to this node in the SBICONF array.

- 4 Performs such additional tasks as allocating and filling in data structures in a device-specific manner. For a non-DIGITAL-supplied device attached to a VAXBI bus, VMS creates generic versions of the channel request block, interrupt dispatch block, and adapter control block—and fills in the appropriate vectors in the system control block—as discussed in Section 14.3.1.

For devices it *does* recognize, VMS additionally calls a VMS-supplied subroutine, the address of which it obtains from the device-type table, that performs further device-specific initialization.

For devices it does *not* recognize, VMS must defer device-specific initialization to the device driver's initialization routine.

³ If no device exists at a given VAXBI node address, the CPU becomes aware of this in a system-specific way. For example, the VAX 8200/8250/8300/8350 systems experience a machine check, whereas the VAX 8530/8550/8700/8800/8830/8840 and VAX 6200-series systems determine that the node is vacant by reading an NXM (nonexistent memory) error from the BIIC Bus Error Register of the NBIB, PBIB, or DWMBA on the VAXBI being examined.

⁴ The CONFREGL array is a set of longwords in system pool pointed to by EXE\$GL_CONFREGL. The CONFREGL array contains an entry for each possible VAXBI node. For VAX 8200/8250/8300/8350 systems, with one VAXBI, this array has 16 entries. For VAX 8530/8700/8800/8830/8840 and VAX 6200-series systems, this array has 16 entries for each VAXBI bus on the system.

⁵ The SBICONF array is a set of longwords, similar in structure to the CONFREGL array and pointed to by MMG\$GL_SBICONF, that lists the system virtual addresses of the base of the node space for each node on a VAXBI bus.

Generic VAXBI Device Support

14.3 Initialization Performed by VMS

14.3.1 Data Structures

The adapter initialization module creates and prepares a channel request block, interrupt dispatch block, and an adapter control block in the manner described in this section. For each data structure it creates, VMS fills in the first three longwords with the standard VMS header information (that is, the structure type, size, and links).

Channel Request Block

For the newly created channel request block (CRB), VMS performs the following tasks:

- Sets up the resource wait queue header (CRB\$L_WQFL and CRB\$L_WQBL)
- Sets the bit CRB\$V_UNINIT in CRB\$B_MASK to indicate to the System Generation Utility that, although the CRB exists, its controller initialization routine has not yet been called
- Initializes four interrupt dispatchers (CRB\$L_INTD, CRB\$L_INTD2, and so on) so that they have the effect of pushing general registers R0 through R5 onto the stack, and issuing a JSB instruction

The adapter initialization module always creates the four vectors, in contrast to the methods by which UNIBUS/Q22 bus drivers control the number of vectors created (see Section 12.3.3). The destination of the JSB instruction at initialization is a standard null interrupt handler which merely dismisses the interrupt. Later, when the specific device driver is loaded for the device (see Section 14.8), the driver's interrupt service routine address replaces this null interrupt handler in the dispatchers. As necessary, the driver specifies the addresses of its interrupt service routines as follows:

```
DPT_STORE, CRB, CRB$L_INTD, D, isr_for_1st_vector
DPT_STORE, CRB, CRB$L_INTD2+VEC$L_ISR, D, isr_for_2nd_vector
DPT_STORE, CRB, CRB$L_INTD+(2*VEC$K_LENGTH)+VEC$L_ISR, D, isr_for_3rd_vector
DPT_STORE, CRB, CRB$L_INTD+(3*VEC$K_LENGTH)+VEC$L_ISR, D, isr_for_4th_vector
```

Interrupt Dispatch Block

VMS initializes the interrupt dispatch block (IDB) in the following manner:

- Sets the number of device units controlled by this interrupt dispatch block (IDB\$W_UNITS) to 1. The list of unit control block (UCB) addresses in this IDB, as a result, is one longword in size. The driver-loading procedure writes a UCB address into this longword whenever it creates a new UCB associated with the controller. Because there is only one slot in this array, drivers for non-DIGITAL-supplied multiunit controllers must use a different mechanism to locate the UCB of interest at the time of an interrupt.
- Copies the virtual address of the base of this device's node space to IDB\$L_CSR from the corresponding slot in the SBICONF array.

Generic VAXBI Device Support

14.3 Initialization Performed by VMS

Adapter Control Block

VMS creates a truncated adapter control block (ADP) for a non-DIGITAL-supplied VAXBI device (48 bytes as opposed to the traditional 600 bytes). The ADP it creates contains no fields reserved for the allocation and accounting of data paths or map registers. VMS prepares this generic ADP in the following manner:

- Copies the virtual address of the base of this device's node space to ADP\$L_CSR from IDB\$L_CSR.
- Places the VAXBI node ID of this device in ADP\$W_TR.
- Stores the value AT\$_GENBI (signifying the *generic* VAXBI ADP type) in ADP\$W_ADPTYPE.
- Calculates the address of the first of the four interrupt vectors for this node in the system control block (SCB), and places it in ADP\$L_AVECTOR. A driver can determine the addresses of the other three SCB vectors by adding 64, 128, or 192, respectively, to the address of this first SCB vector.
- Saves the offset of this first SCB vector from the start of its SCB page in ADP\$W_BI_VECTOR. (Refer to Section 14.3.2 for a description of the SCB.)
- Places in ADP\$L_BI_IDR a longword mask with a single bit set, as appropriate to the VAX system, that specifies which VAXBI node should become the destination of interrupts from this node. In VAX 8200/8250/8300/8350 systems, the VAXBI node of the primary processor becomes the destination for interrupts. In VAX 8530/8550/8700/8800, VAX 8830/8840, and VAX 6200-series systems, it is the VAXBI node of the NBIB, PBIB, or DWMB A/B on the particular VAXBI bus on which this device resides that becomes the destination for such interrupts.
- Stores in ADP\$L_MBASC B—and in each of the device's four SCB vectors—the address of the interrupt dispatcher. The actual stored value is CRB\$L_INTD+1, CRB\$L_INTD2, and so on, the set low bit of the address indicating that the interrupt stack be used to service the interrupt. Certain powerfail recovery operations use the contents of ADP\$L_MBASC B to refresh the SCB vectors.
- Saves in ADP\$L_MBASPTE the contents of the first of the 16 SPTES that map the device's node space. Certain recovery operations use the contents of ADP\$L_MBASPTE to restore correct SPT E values and remap node space following a power failure.
- Places in ADP\$L_BIMASTER the address of the ADP of the memory-interconnect-to-VAXBI-adapter (NBI, PBI, or DWMB A). Note that there is no memory-interconnect-to-VAXBI adapter for VAX 8200/8250/8300/8350 configurations.

Generic VAXBI Device Support

14.3 Initialization Performed by VMS

14.3.2 System Control Block

The system control block (SCB) consists of one or more pages of vectors. For all VAX processing systems, the first half page contains vectors used in exception dispatching. VMS uses the remainder of the first page, as well as subsequent pages, in a system-specific way.

For VAX 8200/8250/8300/8350 systems, VMS assigns the vectors from 100_{16} to $1FC_{16}$ to VAXBI devices in the order of their node IDs.

The VAX 8530/8550/8700/8800/8830/8840 and VAX 6200-series system architectures relegate vectors 100_{16} to $1FC_{16}$ to NMI nexus vectors. Page 1 is reserved for the first "offsettable" device that exists in the system. (An "offsettable" device is an adapter such as the BI-to-UNIBUS adapter (DWBUA) that passes interrupts from devices on another bus to the VAXBI and, from there, to the memory interconnect (NMI or XMI) and the processor.) If there is more than one "offsettable" device, an additional SCB page is needed for each.

Ultimately, the vectors for other devices attached to each of the six possible VAXBI buses of the system are contained in the six corresponding SCB pages from page 26 to page 31. In a 4-VAXBI system, for instance, vectors for devices connected to VAXBI 0 and VAXBI 1 on NBI/PBI/DWMBA 0 are assigned to pages 28 and 29 of the SCB, respectively; vectors for devices connected to VAXBI 0 and VAXBI 1 on NBI/PBI/DWMBA 1 are likewise assigned to pages 30 and 31. In a 6-VAXBI system, the vectors are assigned in a similar fashion, starting at page 26.

Generally, a VAX processor obtains a device vector from the BIIC registers of the node that has requested the interrupt (see Figure 14-4). Information supplied in the device vector allows the processor to index to the corresponding interrupt-dispatching vector in the appropriate page of the SCB. For VAX 8200/8250/8300/8350 systems, such information includes the interrupt level of the device and its VAXBI node ID. A similar vector for VAX 8530/8550/8700/8800, VAX 8830/8840, or VAX 6200-series devices further specifies the appropriate NBI/PBI/DWMBA vector offset and the number of the VAXBI bus.

The specific SCB interrupt-dispatching vector, thus found, transfers control to the interrupt-dispatching code in the device's CRB. Upon an interrupt from this device, the SCB vector directs flow into the interrupt dispatcher in the CRB, which saves the register contents and dispatches to the interrupt service routine established by the device driver.

14.4 Initialization Performed by the VAXBI Device Driver

All generic VAXBI device drivers must specify *GENBI* as the adapter type in the **adapter** argument to the DPTAB macro.

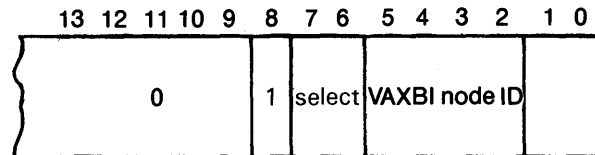
The device driver's initialization routines are expected to initialize the device-specific aspects of the VAXBI device. For non-DIGITAL-supplied devices, the initialization routines perform the sort of tasks that the adapter initialization module performs for the DIGITAL-supplied devices it discovers on a VAXBI bus. For single-unit devices, a separate unit initialization routine may not be necessary.

Generic VAXBI Device Support

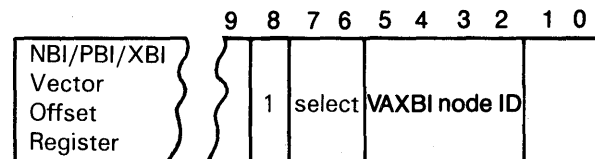
14.4 Initialization Performed by the VAXBI Device Driver

Figure 14-4 VAXBI Device Vectors

For VAX 8200/8250/8300/8350



For VAX 8530/8550/8700/8800/8830/8850 and VAX 6200 Series Devices



↑ "select" indicates one of four interrupt vectors

ZK-5545-86

The VMS System Generation Utility (SYSGEN) calls the controller initialization routine at IPL 31, passing it the following values in the listed general registers:

- R4 pointing to the system virtual address of the device's node space
- R5 pointing to the IDB
- R6 pointing to the DDB
- R8 pointing to the CRB

After the controller initialization routine has completed, SYSGEN calls the driver's unit initialization routine at IPL 31, and passes it the following values in the listed general registers:

- R3 pointing to the system virtual address of the device's node space
- R5 pointing to the UCB

Hardware initialization might include such activities as writing values to BIIC and device-specific registers, examining the results of the BIIC self test, mapping a node's window space, building data structures to control the device, and linking these structures into chains of similar data structures.

This section provides some ideas and guidelines for code that may be necessary in an initialization routine. There is no requirement that driver code perform all of the functions discussed here. The needs of various devices differ, and some devices make more demands on driver software than others.

Code examples in the section assume that R4 initially contains the virtual address of the base of the device's node space and R8 contains the virtual address of the device's CRB.

Generic VAXBI Device Support

14.4 Initialization Performed by the VAXBI Device Driver

14.4.1 Examining BIIC Self-Test Status

According to the hardware specification for all devices attached to a VAXBI bus, a VAXBI node undergoes a self test on power failure recovery and at system boot time. The BIIC indicates the successful completion of the self test by setting BIIC\$V_STS and by clearing BIIC\$V_BROKE in BIIC\$L_BICSR.

A driver unit initialization routine should test these bits before performing any transaction on the VAXBI bus. If BIIC\$V_STS is clear, then self test is still under way. If BIIC\$V_BROKE is set, then the driver action is implementation-specific. In any event, a driver should not set UCB\$V_ONLINE in UCB\$L_STS if the node is not usable.

The maximum duration of the BIIC self test is ten seconds. If a VAXBI node implements the maximum self-test time, then the driver unit initialization routine may have to spin wait for the setting of BIIC\$V_STS (for instance, by embedding the testing instructions in an invocation of the TIMEDWAIT macro). Driver unit initialization routines should perform this spin wait only when UCB\$V_POWER in UCB\$L_STS is set. Otherwise, the driver is being loaded by SYSGEN, and a long spin wait at high IPL will have adverse effects on the rest of the VMS system.

Normally, only diagnostics initiate a self test by setting the SST bit in the BIIC. A VAXBI driver that sets this bit must take special precautions to avoid a machine check and to avoid undetected corruption of VAXBI memory. These precautions include the following steps:

- 1 Use the \$PRTCTINI macro to begin a machine check protection block, supplying the location of the end of the block in the **label** argument and the mask value # <MCHK\$M_NEXM!MCHK\$M_LOG> in the **mask** argument. (Note that you must include an invocation of the \$MCHKDEF macro in the driver to use these symbols.) Code within the block executes at IPL 31.
- 2 Disable arbitration on the VAXBI node being reset by setting BIIC\$V_ARBCNTRL in BIIC\$L_BICSR.
- 3 Set BIIC\$V_SST and BIIC\$V_STS simultaneously to initiate the self test. Do not set BIIC\$V_SST in the same instruction that disables arbitration.
- 4 Use the \$PRTCTEND macro to end the machine check protection block. You must specify in the **label** argument the same value you specified in the **label** argument to the \$PRTCTINI macro.
- 5 Do not access the BIIC registers for at least one millisecond. You may not even check the state of the STS bit during this interval.
- 6 Do not access any other address on the VAXBI node until the self test has completed.

Generic VAXBI Device Support

14.4 Initialization Performed by the VAXBI Device Driver

14.4.2 Clearing BIIC Errors, Setting Interrupts, and Enabling Interrupts

There is a set of tasks that a VAXBI driver should perform during initialization that ensures that interrupts are properly enabled and delivered to an appropriate VAXBI target node. These tasks include the following:

- Clearing any outstanding set bits in the Bus Error Register.
- Setting the target node for interrupts in the Interrupt Destination Register.
- Setting the device interrupt vector in the Error Interrupt Control Register.
- Setting the device interrupt vector in the User Interface Interrupt Control Register.
- Enabling hard and soft error interrupts as required by the device. Typically hard errors are enabled and soft errors are disabled.
- Enabling interrupts upon certain types of transactions to user interface CSR space.

It is important that the interrupt vectors and destination be set up *before* BIIC hard error and soft error interrupts are enabled. An error occurring while error interrupts are enabled but the vector is not initialized could lead to an invalid condition.

14.4.2.1 Clearing the Bus Error Register

The following example clears all set bits in the Bus Error Register (BIIC\$_BER) to prevent spurious or pending error interrupts at initialization.

```
MOVL    BIIC$_BER(R4),-           ;Clear all set write-1-to-clear
        BIIC$_BER(R4)             ; bits in BIIC$_BER
```

14.4.2.2 Loading the Interrupt Destination Register

The Interrupt Destination Register (BIIC\$_IDR) specifies which VAXBI node should become the destination of interrupts from this node. In VAX 8200/8250/8300/8350 systems, the VAXBI node of the primary CPU becomes the destination for interrupts. In VAX 8530/8550/8700/8800, VAX 8830/8840, and VAX 6200-series systems, the VAXBI node of the NBIB, PBIB, or DWMBAB on the particular VAXBI on which this device resides becomes the destination for such interrupts.

The VMS system initialization procedure described in Section 14.3 creates a 32-bit mask with the appropriate bit set and stores it in ADP\$_BI_IDR. If a driver must set the Interrupt Destination Register, it can simply move this value to the BIIC register:

```
MOVL    CRB$_INTD+VEC$_ADP(R8),R0 ;Get ADP address
MOVL    ADP$_BI_IDR(R0),-         ;Write to IDR
        BIIC$_IDR(R4)
```


Generic VAXBI Device Support

14.4 Initialization Performed by the VAXBI Device Driver

14.4.2.3 Setting Interrupt Vectors

A VAXBI node uses the Error Interrupt Control Register (BIIC\$L_EICR) to determine the SCB vector through which to interrupt when a BIIC at this node detects a bus error. The User Interface Interrupt Control Register (BIIC\$L_UICR) similarly controls the operation of interrupts initiated by the device at this node. A driver can also use the Error Interrupt Control Register to support a device that generates secondary interrupt vectors.

Because the VMS system initialization procedure described in Section 14.3 saves the offset of the node's first SCB vector from the start of its SCB page in ADP\$W_BI_VECTOR, a driver can initialize both of these registers by using code similar to that in the following example:

```
MOVL  CRB$L_INTD+VEC$L_ADP(R8),R0      ;Get ADP address
MOVZWL ADP$W_BI_VECTOR(R0),R2         ;Get device vector
MOVL  BIIC$L_UICR(R4),BIIC$L_UICR(R4) ;Clear user vector
MOVL  R2,BIIC$L_UICR(R4)             ;Set user vector
BISL  #1<BIIC$V_LEVEL+BIIC$S_LEVEL-1>,R2
                                           ;OR in interrupt level
                                           ;BR7 in this case
MOVL  BIIC$L_EICR(R4),BIIC$L_EICR(R4) ;Clear error vector
MOVL  R2,BIIC$L_EICR(R4)             ;Set error vector
```

Note that the driver clears both vectors before it actually sets them. Clearing BIIC\$L_UICR and BIIC\$L_EICR causes any pending interrupt to be cleared. Also note that the interrupt level must be set in BIIC\$L_EICR, in this case BR7. If the level is not set, an interrupt will never be generated.

14.4.2.4 Enabling Error Interrupts

Finally, to enable interrupts that report errors detected by the node's BIIC, the controller initialization routine can set the soft error interrupt-enable or hard error interrupt-enable bits in the VAXBI Control and Status Register. The BIIC sets bits in the Bus Error Register (BIIC\$L_BER) to reflect the type of bus error reported by the interrupt.

```
BISL  #<BIIC$M_SEIE!BIIC$M_HEIE>,-    ;Soft error interrupt enable
      BIIC$L_BICSR(R4)                 ;Hard error interrupt enable
```

14.4.2.5 Enabling BIIC Options

Device registers are in the area of node space called user interface CSR space, and are located following the 256 bytes reserved for the BIIC-required registers. Use of user interface CSR space is implementation-dependent.

For the processor to be alerted to various transactions directed at user interface CSR space, the controller initialization routine of devices that support such transactions should set appropriate bits in the BCI Control and Status Register (BIIC\$L_BCICR). See Table 14-1 for definitions of these bits.

The following example enables a node to alert the node specified as the interrupt destination (in BIIC\$L_IDR) when a retry timeout, STOP command, or read or write transaction is directed at its user interface CSR space.

```
BISL  #<BIIC$M_STOPEN!-                ;Stop enable
      BIIC$M_RTTOEVEN!-                 ;Retry timeout enable
      BIIC$M_UCSREN>,-                 ;User CSR enable
      BIIC$L_BCICR(R4)
```

Generic VAXBI Device Support

14.4 Initialization Performed by the VAXBI Device Driver

14.4.3 Mapping Window Space

Each VAXBI, starting at address 20400000₁₆ in its I/O address space, provides 16 address blocks of 256K bytes apiece, called *window space*. VAXBI nodes can use window space if it is necessary to map VAXBI transactions to memory space on a target bus, although only such nodes as the DWBUA adapter currently use this feature.

Whereas the VMS initialization routine maps each VAXBI node's node space to virtual addresses, it does not automatically map each node's window space. If a device needs to use its window space, it is up to the driver's unit initialization routine to map this space.

First of all, the driver must determine the starting physical address of the node's window space. Figure 14-3 illustrates how VAXBI addresses are constructed. Drivers can use the following VMS-supplied macros (in SYS\$LIBRARY:LIB.MLB) to access pertinent VAXBI addresses and values:

\$IO8SSDEF (for VAX 8200/8250/8300/8350 systems)
\$IO8NNDEF (for VAX 8530/8550/8700/8800 systems)
\$IO8NNDEF and \$IO8PSDEF (for VAX 8830/8840 systems)
\$IO9CCDEF (for VAX 6200-series systems)

A driver calculates the starting address of a node's window space by first determining the offset to the start of the VAXBI node's window space from the beginning of its VAXBI I/O space. To do so, it performs the following tasks:

- 1 Extracts the VAXBI node ID from bits <3:0> of ADP\$W_TR.
- 2 Multiplies the VAXBI node ID with the size of window space. The driver obtains this value from the following symbols:

| Symbol | System |
|------------------|-----------------------------------|
| IO8SS\$AL_NDSPER | VAX 8200/8250/8300/8350 |
| IO8NN\$AL_NDSPER | VAX 8530/8550/8700/8800/8830/8840 |
| IO9CC\$C_BIWSIZ | VAX 6200 series |

- 3 Adds the address of the window space of VAXBI node 0 to this value. The driver obtains this value from the following symbols:

| Symbol | System |
|-------------------|-----------------------------------|
| IO8SS\$AL_NODESP | VAX 8200/8250/8300/8350 |
| IO8NN\$AL_NODESP | VAX 8530/8550/8700/8800/8830/8840 |
| IO9CC\$C_BIWINDOW | VAX 6200 series |

The driver for a device on a VAX system configured with more than one VAXBI bus (for instance, the VAX 8530/8550/8700/8800/8830/8840 or VAX 6200-series systems) must proceed to calculate the start of the I/O address space of the VAXBI bus to which the device is attached. It adds the result of the following steps to the value it has obtained above:

- 1 Determine the offset to the I/O address space of the VAXBI bus to which the node is attached.

Generic VAXBI Device Support

14.4 Initialization Performed by the VAXBI Device Driver

For VAX 6200-series systems, a driver first must obtain the XMI node ID of the NBI, PBI, or DWMBA to which the VAXBI is connected. It determines the address of the NBI's, PBI's or DWMBA's ADP at offset ADP\$L_BIMASTER of the node's ADP. It then finds the XMI node of the NBI, PBI, or DWMBA at offset ADP\$W_XBIA_TR of the ADP.

For VAX 8530/8550/8700/8800/8830/8840 configurations, a driver obtains the VAXBI bus number from bits <7:4> from offset ADP\$W_TR of the node's ADP.

- 2 Multiply this value by 2000000_{16} , the amount of physical address space allocated for each VAXBI bus.
- 3 Add to this value the base of I/O address space. The driver obtains this value from the following symbols:

| Symbol | System |
|------------------|-----------------------------------|
| IO8SS\$AL_IOBASE | VAX 8200/8250/8300/8350 |
| IO8NN\$AL_IOBASE | VAX 8530/8550/8700/8800/8830/8840 |
| IO9CC\$AL_IOBASE | VAX 6200 series |

After performing these calculations, the driver must associate each page of window space to be used with a system page-table entry (SPTe) that maps the page-frame number (PFN) of the physical page in window space to a system virtual address. VMS includes the routine LDR\$ALLOC_PT, described in Appendix B, that allocates system page-table entries (SPTes) for a specified number of pages.

Because LDR\$ALLOC_PT executes at IPL\$_SYNCH (holding the MMG spin lock in a VMS multiprocessing system), the controller initialization routine must fork from IPL\$_POWER to fork IPL (using the CRB fork block) prior to calling it. See Section 11.1.5 for a discussion of forking in a driver initialization routine.

Finally, once the SPTes have been allocated, the driver moves the PFNs of the window space pages into the SPTes, sets their valid bits, and initializes them in a device-specific manner.

14.5 DMA Transfers

The method by which a device accomplishes direct-memory-access (DMA) transfers depends upon the characteristics of the device. As part of a VAXBI read or write transaction, such a device must place on the VAXBI bus a physical address, the target of which is a memory node or a node (such as an NBIB adapter) that transmits the request to memory across another bus.

For the DMA device to successfully access the memory pages of a buffer involved in an I/O transfer, it must be given sufficient information as to the size and location of these buffer pages, the type of transaction that is requested, an offset into the first page of the buffer, and the length of the transaction. In addition, if the size of the transaction causes it to exceed the boundaries of a page, the device must have some means of accessing the remaining pages—even if they are, as is most likely, scattered throughout physical memory.

Generic VAXBI Device Support

14.5 DMA Transfers

As a result, devices make use of several types of structures, the purpose of which is to help generate a succession of contiguous physical addresses on the VAXBI bus, that map to the various pages of the buffer involved in the transfer. Some possible strategies of this sort include the following:

- A physically contiguous buffer in memory
- System page tables in system memory
- Process page tables locked in system memory
- Map registers in the device's VAXBI I/O address space

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

VMS supplies two separate techniques, applied by traditional VMS drivers and described in Section 6.3.1.

- *Direct I/O*, the technique used most commonly by DMA drivers, locks the user buffer in memory as well as the page-table entries that map it. Such a driver calls a VMS-supplied FDT routine that prepares the user buffer for direct I/O.
- *Buffered I/O* is the strategy whereby the driver FDT routine allocates a buffer from nonpaged pool. It is this intermediate buffer that is involved in the DMA transfer. The FDT routine copies the data from the user buffer to the system buffer for a write request; I/O postprocessing routines deliver data from the system buffer to the user buffer for a read request.

That DMA drivers may make use of either VMS direct I/O or buffered I/O is one way by which these drivers can supply specific information needed by the device to accomplish a DMA transfer. Those driver FDT routines that call a VMS direct-I/O FDT routine provide the following information in the device's unit control block (UCB):

| | |
|---------------|---|
| UCB\$L_SVAPTE | Virtual address of the system page-table entry (SPTE) for the first page used in the transfer |
| UCB\$W_BOFF | Byte offset in the first page of the transfer buffer |
| UCB\$W_BCNT | Size in bytes of the transfer |

FDT routines for buffered I/O call EXE\$ALLOCBUF, EXE\$DEBIT_BYTCNT_ALO, or EXE\$DEBIT_BYTCNT_BYTLM_ALO to obtain a nonpaged pool buffer (debiting a job's byte count quota in the last two routines) and initialize the same UCB fields with the following information:

| | |
|---------------|---|
| UCB\$L_SVAPTE | Virtual address of system buffer used in the I/O transfer |
| UCB\$W_BOFF | Number of bytes to be charged to the process for the transfer |
| UCB\$W_BCNT | Size in bytes of the transfer |

Generic VAXBI Device Support

14.5 DMA Transfers

If a driver's fork process must manipulate the data in any way at fork level (that is, outside of the driver's FDT routines), then it needs a virtual address it can use to access the data. Such a requirement could cause the driver writer to consider structuring the driver so that it uses buffered I/O. For short transfers, this need also could be accommodated by the driver's loading an SPTE with the correct PFN and computing the associated system virtual address. The drivers for the disks that have ECC correction applied by the host do this when there is an ECC error detected. The controller can tell the driver that the error in the data in memory can be corrected by applying some pattern to a part of the data, but the fork process has to perform the correction, not the controller.

```
MOVL   IRP$L_SVAPTE(R3),R2           ;Get address of system buffer
SUBW3  #12,8(R2),UCB$W_BCNT(R5)      ;Calculate system buffer length
BICW3  #C<VA$M_BYTE>, (R2),UCB$W_BOFF ;Put offset in buffer
EXTZV  #VA$V_VPN,#VA$S_VPN(R2),R2   ;Get system virtual page number
MOVL   G^MMG$GL_SPTBASE,R1          ;Get address of system page table
MOVAL  (R1)[R2],UCB$L_SVAPTE(R5)    ;Get system virtual address of page
```

14.5.1 Example: DMB32 Asynchronous/Synchronous Multiplexer

The DMB32 asynchronous/synchronous multiplexer can use any of four different modes of address translation for DMA accesses. Under each of these modes, the DMB32 requires that its driver supply an address by which it can either directly or indirectly obtain the pages of the buffer that is involved in the transfer. The four different translation modes require such addresses in one of the following forms:

- 1 System virtual address of a buffer
- 2 System virtual address of a page-table entry
- 3 Physical address of a page table
- 4 Address of a physically-contiguous buffer

System Virtual Address of a Buffer and System Virtual Address of a Page-Table Entry

The DMB32 can itself perform the first two types of address translation because it can read entries in the VMS system page table (see the *VAX/VMS Internals and Data Structures* manual for a description of page-table entries). The controller initialization routine of a DMB32 device driver supplies the physical address and length of the VMS system page table, plus the virtual address and length of the VMS global page table. It also sets a page-table-valid bit in a device maintenance register.

As a result, a driver for a DMB32 device could use either direct I/O or buffered I/O, and accordingly load a device register with the system virtual address of the page-table entry that maps the buffer or the system virtual address of the buffer itself. After the driver has loaded other device registers with a buffer offset value and a transfer size—and set the “start” bit in a DMB32 line-control register—the DMB32 performs the transfer without any additional mapping or other driver intervention.

Generic VAXBI Device Support

14.5 DMA Transfers

Physical Address of a Page Table

In this mode, the DMB32 can be given the physical address of a page table that maps the I/O transfer. The DMB32 architecture mandates that each page-table entry be four bytes long and that the page table be aligned on a longword boundary. Also, each page is 512 bytes long. However, the page table can be anywhere in memory, possibly at a range of VAXBI I/O-space addresses belonging to the node to which the DMB32 adapter is attached. To perform a DMA transfer under this addressing mode, the DMB32 adapter requires the offset of the first byte of the buffer that is in the page described by the page-table entry. Each page-table entry contains bits <29:9> of the physical address of the page that is to be accessed.

In this case, the driver must extract the PFNs of the pages involved in the transfer and insert them into the page table of the device. The following is an example of a routine that translates a system virtual address to a physical address. It returns the physical address at the top of the stack.

```
VIRT_TO_PHYAD:
    PUSHL    (SP)                                ;Create slot at top of stack
                                                ; for return value
    PUSHR    #^M<R0,R1,R2,R3>                    ;Save registers
    BICL3    #-512,R1,R0                          ;R0 = byte offset of address
    EXTZV    #VA$V_VPN,-                          ;Extract VPN
            #VA$$_VPN,R1,R2                       ; and put it in R2
    MOVL     G^MMG$GL_SPTBASE,R3                 ;R3 => system page table
    MOVL     (R3)[R2],R3                          ;R3 => PTE
    EXTZV    #PTE$V_PFN,-                          ;Get page frame number of buffer
            #PTE$$_PFN,R3,R3                     ; page into R3
    ASHL     #VA$V_VPN,R3,R3                      ;Shift into place for physical
                                                ; address
    BISL3    R0,R3,20(SP)                         ;Put result into stack slot
    POPR     #^M<R0,R1,R2,R3>                    ;Restore registers
    RSB
```

Physical Address of a Buffer

If the device can neither read system page tables nor has its own scatter-gather map—and must perform a DMA transfer that spans physical pages—it must rely upon the actual contiguity of the physical pages involved in the transfer. Because there is no guarantee that this is the state of the user's buffer, the driver must allocate an intermediate buffer consisting of contiguous physical pages. The driver never deallocates this buffer unless the driver is being unloaded by means of SYSGEN's RELOAD command. (The driver unloading routine can call COM\$DRVDEALMEM to do so.) The best time to allocate such a buffer is during the device's initialization, when memory is most likely to be contiguous.

The VMS routine EXE\$ALOPHYCNTG, described in Appendix C, allocates such a buffer. The size of the buffer that should be allocated depends on the device's characteristics and the size of the transfers requested on the device. A buffer of four pages is likely to be large enough for most disk transfers, for example; but if you have enough memory on your system, you might want to make your buffer the size of a disk track in order to reduce disk latency. In any event, large transfers to the device can be segmented into transfers the size of your intermediate buffer.

The start-I/O routine of such a driver copies the data from the user's buffer into the intermediate, physically contiguous buffer by means of the routine IOC\$MOVFRUSER.

Generic VAXBI Device Support

14.5 DMA Transfers

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

- 1 Determines the physical address of the buffer from the system virtual address returned by EXE\$ALOPHYCNTG
- 2 Moves the address to the device address register
- 3 Activates the device
- 4 If the transfer size exceeds the size of the buffer, returns to step 1

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

14.6 Unit Initialization Routine

A generic VAXBI device driver may include a unit initialization routine, in addition to its controller initialization routine, if it services a multiunit device.

SYSGEN attempts to create a UCB and call the unit initialization routine for the number of units specified in the **maxunits** argument to the DPTAB macro.

When called in the process of driver loading, the unit initialization routine of a generic VAXBI device driver must therefore determine if the unit it is currently servicing actually exists. Prior to returning control to SYSGEN, the routine must place in R0 a success status (low bit set) if the unit exists or a failure status (low bit clear) if it does not. If SYSGEN receives failure status, it deallocates the UCB for the unit and proceeds to configure the next unit in a similar manner.

14.7 Register Dumping Routine

In the event of a device error or a VAXBI bus error, a driver's register dumping routine should contain code that makes certain interesting registers available for error logging. Apart from any device registers that should be saved, the following BIIC registers may contain information important in determining the cause of the error: the Device Register (BIIC\$_DTREG), the VAXBI Control and Status Register (BIIC\$_BICSR), the Bus Error Register (BIIC\$_BER), the Error Interrupt Control Register (BIIC\$_EICR), and the Interrupt Destination Register (BIIC\$_IDR).

The following is an example of part of a register dumping routine that saves the contents of these BIIC registers in an error buffer.

```
MOVL  BIIC$_DTREG(R4), (R0)+ ;Device Type Register
MOVL  BIIC$_BICSR(R4), (R0)+ ;BIIC CSR Register
MOVL  BIIC$_BER(R4), (R0)+ ;Bus Error Register
MOVL  BIIC$_EICR(R4), (R0)+ ;Error Interrupt Control Register
MOVL  BIIC$_IDR(R4), (R0)+ ;Interrupt Destination Register
```

Generic VAXBI Device Support

14.8 Loading a VAXBI Device Driver

14.8 Loading a VAXBI Device Driver

The System Generation Utility (SYSGEN) loads the device driver into system virtual memory, creates additional data structures for the device unit, connects the device's interrupt vectors, and calls the device driver's controller initialization routine and unit initialization routine.

Chapter 15 discusses the SYSGEN commands commonly used during driver loading. The following discussion pertains to those aspects of the loading process that specifically relate to the support of non-DIGITAL-supplied VAXBI devices.

Because the autoconfigure facility cannot recognize non-DIGITAL-supplied VAXBI devices, the system startup procedure (or a later invocation of SYSGEN) must explicitly request that SYSGEN connect the device.⁶ SYSGEN responds to such explicit requests by utilizing the data structures created by the adapter initialization module for the unknown VAXBI device to load the associated device driver and invoke its initialization routines.

For example, suppose that an unknown VAXBI device were located at node 3 on a given VAXBI bus, and that the software device driver for this device were known as "ZZDRIVER". During adapter initialization, VMS would have encountered an unknown type of VAXBI device at node 3 and would have performed the following operations:

- Mapped the node space for node 3 into system virtual memory
- Constructed various data structures to govern the future operation of this device

SYSGEN executes in response to the following commands:

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> CONNECT ZZA0:/ADAPTER=3
```

SYSGEN performs the following activities:

- 1 Searches the list of ADPs in the system to find the ADP for this VAXBI node (node 3) and, in turn, locates the corresponding CRB and IDB by following pointers in the ADP.
- 2 Loads ZZDRIVER into system virtual memory. If the /DRIVER qualifier is specified, SYSGEN loads the specified driver instead.
- 3 Creates a UCB for device ZZA0 and places the address of the device's CRB in that UCB. SYSGEN also initializes other UCB fields at this time.
- 4 Sets the first entry in the IDB UCB array (IDB\$L_UCBLST) to point to the new UCB.
- 5 Creates a DDB for the ZZA device/controller combination. This allows user programs to assign I/O channels to device ZZA0 later. This DDB, in turn, points to the location in memory where ZZDRIVER has been loaded and to the UCB for the ZZA0 device.

⁶ Because the autoconfigure facility will never be called for a non-DIGITAL-supplied device, any unit delivery routine that a VAXBI device driver may include will never be called.

Generic VAXBI Device Support

14.8 Loading a VAXBI Device Driver

- 6 Calls the controller initialization routine in ZZDRIVER at IPL 31.
- 7 Calls the unit initialization routine in ZZDRIVER at IPL 31.

Note: If you did not specify *GENBI* as the adapter type in the adapter argument to the *DPTAB* macro, the *CONNECT* command will fail with the following error message:

```
%SYSGEN-E-INVVEC, invalid or unspecified interrupt vector
```

14.9 BIIC Register Definitions

Each VAXBI node is required to implement a minimum set of registers contained in specific locations within the node's node space. VMS automatically maps each node's node space at boot time and provides the macro *\$BIICDEF* (in *SYS\$LIBRARY:LIB.MLB*) to define offsets to the BIIC registers and their significant bit fields.

The contents of the BIIC registers are illustrated in Figure 14-5 and described in Table 14-1. See the *VAXBI Options Handbook* for a discussion of the BIIC and the rules for configuring its registers.

Note: Fields marked "Reserved to DIGITAL" are reserved for DIGITAL's future use and should contain zeros.

Generic VAXBI Device Support

14.9 BIIC Register Definitions

Figure 14–5 Backplane Interconnect Interface Chip (BIIC) Registers

| |
|--------------------|
| BIIC\$_DTREG |
| BIIC\$_BICSR |
| BIIC\$_BER |
| BIIC\$_EICR |
| BIIC\$_IDR |
| BIIC\$_IPIMR |
| BIIC\$_IPIDR |
| BIIC\$_IPISR |
| BIIC\$_SAR |
| BIIC\$_EAR |
| BIIC\$_BCICR |
| BIIC\$_WSR |
| BIIC\$_IPISTPF |
| unused |
| unused |
| unused |
| BIIC\$_UICR |
| unused (172 bytes) |
| BIIC\$_GPR0 |
| BIIC\$_GPR1 |
| BIIC\$_GPR2 |
| BIIC\$_GPR3 |

ZK-6623-HC

Generic VAXBI Device Support

14.9 BIIC Register Definitions

Table 14–1 Contents of the BIIC Registers

| Field Name | Contents |
|--------------|---|
| BIIC\$_DTREG | <p>Device Register.</p> <p>BIIC\$_DTREG consists of the following two words:</p> <p>BIIC\$_DEVTYPE Device type. This field is written by device hardware and self-test microcode. It contains two bit fields:</p> <p style="padding-left: 2em;">BIIC\$_MEMNODE (bits <14:8>), when clear, indicates a memory node.</p> <p style="padding-left: 2em;">BIIC\$_NONDEC (bit 15), when clear, indicates a DIGITAL-supplied device; it should be 1 otherwise.</p> <p>BIIC\$_REVCODE Revision code.</p> |
| BIIC\$_BICSR | <p>VAXBI Control and Status Register.</p> <p>The following fields are defined within BIIC\$_BICSR.</p> <p>BIIC\$_NODE_ID¹ Node ID. This field is automatically loaded during the powerup sequence. Reserved to DIGITAL.</p> <p>BIIC\$_ARBCNTL Arbitration mode used by the node. Currently, all arbitration modes except dual round-robin arbitration are reserved to DIGITAL. Correspondingly, these two bits should be clear. When these two bits are set, arbitration is disabled, thus preventing a node from starting a VAXBI transaction.</p> <p>BIIC\$_SEIE Soft error interrupt enable. When set, this bit allows the node to generate an interrupt when the soft error summary bit (BIIC\$_SES) in this register is set.</p> <p>BIIC\$_HEIE Hard error interrupt enable. When set, this bit allows the node to generate an interrupt when the hard error summary bit (BIIC\$_HES) in this register is set.</p> <p>BIIC\$_UWP Unlock write pending. When set, this bit signals that the master port interface at this node has successfully completed an IRCI (Interlock Read with Cache Intent) transaction. The node clears this bit when it successfully completes a corresponding UWMCI (Unlock Write Mask with Cache Intent) instruction.</p> <p><9> Reserved to DIGITAL. Must be zero.</p> |

¹Read-only field.

Generic VAXBI Device Support

14.9 BIIC Register Definitions

Table 14–1 (Cont.) Contents of the BIIC Registers

| Field Name | Contents |
|------------|--|
| | <p>BIIC\$V_SST Node reset. This bit is normally used by diagnostics to initiate the BIIC internal self test. Prior to initiating a BIIC self test, a node should disable arbitration by setting both bits in BIIC\$V_ARBCNTL. When BIIC\$V_SST is set, the self-test status bit (BIIC\$V_STS) in this register must also be set.</p> <p>Reads to BIIC\$V_SST return a zero.</p> |
| | <p>BIIC\$V_STS Self-test status. When set, this bit indicates that the BIIC has passed its self test. The controller initialization routine of a VAXBI device driver should inspect this bit and the BIIC\$V_BROKE bit before proceeding with any VAXBI transactions. During the self-test sequence, BIIC\$V_STS will automatically be reset by the BIIC to allow the proper recording of the new self-test results at the end of self test.</p> |
| | <p>BIIC\$V_BROKE² Broke bit. When cleared by the device's self test, this bit indicates that the device has passed its self test. The controller initialization routine of a VAXBI device driver should inspect this bit and the BIIC\$V_STS bit before proceeding with any VAXBI transactions.</p> |
| | <p>BIIC\$V_INIT² Initialization bit.</p> |
| | <p>BIIC\$V_SES¹ Soft error summary. When set, this bit indicates that one or more of the soft error bits in the Bus Error Register (BIIC\$_BER) is set.</p> |
| | <p>BIIC\$V_HES¹ Hard error summary. When set, indicates that one or more of the hard error bits in the Bus Error Register (BIIC\$_BER) is set.</p> |
| | <p>BIIC\$V_BIICTYPE¹ BIIC type. These bits <23:16> always contain 00000001.</p> |
| | <p>BIIC\$V_BIICREVN¹ BIIC revision number.</p> |
| BIIC\$_BER | <p>Bus Error Register.</p> <p>The following bits are defined within BIIC\$_BER. Bits <30:16> are hard error bits and bits <2:0> are soft error bits.</p> <p>BIIC\$V_NPE² Null bus parity error.</p> <p>BIIC\$V_CRD² Corrected read data.</p> <p>BIIC\$V_IPE² ID parity error.</p> <p>BIIC\$V_UPEN¹ User parity enabled.</p> <p><14:4> ¹ Reserved to DIGITAL. Must be zero.</p> <p>BIIC\$V_ICE² Illegal confirmation error.</p> <p>BIIC\$V_NEX² Nonexistent address.</p> |

¹Read-only field.

²Write-one-to-clear bit. Write-type transactions cannot set this bit.

Generic VAXBI Device Support

14.9 BIIC Register Definitions

Table 14–1 (Cont.) Contents of the BIIC Registers

| Field Name | Contents |
|-------------|---|
| | BIIC\$_BTO ² Bus timeout. |
| | BIIC\$_STO ² Stall timeout. |
| | BIIC\$_RTO ² Retry timeout. |
| | BIIC\$_RDS ² Read data substitute. |
| | BIIC\$_SPE ² Slave parity error. |
| | BIIC\$_CPE ² Command parity error. |
| | BIIC\$_IVE ² IDENT vector error. |
| | BIIC\$_TDF ² Transmitter during fault. |
| | BIIC\$_ISE ² Interlock sequence error. |
| | BIIC\$_MPE ² Master parity error. |
| | BIIC\$_CTE ² Control transmit error. |
| | BIIC\$_MTCE ² Master transmit check error. |
| | BIIC\$_NMR ² NO ACK to multiresponder command received. |
| | <31> Reserved to DIGITAL. Must be zero. |
| BIIC\$_EICR | <p>Error Interrupt Control Register. This register supplies information the node uses to request and monitor the status of both BIIC-detected and forced-error interrupts: that is, those interrupts signaled by either the setting of a bit in the Bus Error Register (BIIC\$_BER) or the setting of the force bit (BIIC\$_EIFORCE) in this register, respectively. The node can initiate BIIC-detected error-interrupt requests only if the appropriate error-interrupt enables (BIIC\$_SEIE and/or BIIC\$_HEIE) are set in the VAXBI Control and Status Register (BIIC\$_BICSR).</p> <p>The following fields are defined within BIIC\$_EICR.</p> |
| | <1:0> ¹ Reserved to DIGITAL. Must be zero. |
| | BIIC\$_EIVECTOR 12-bit vector used in error interrupt sequences. |
| | <15:14> ¹ Reserved to DIGITAL. Must be zero. |
| | BIIC\$_LEVEL These four bits (<19:16>) correspond to the four interrupt levels (INT <7:4>) of the VAXBI bus. A set bit causes the corresponding level to be used when INTR commands under control of this register are transmitted. |
| | BIIC\$_EIFORCE Force bit. When set, this bit posts an error interrupt request in the same way as a bit set in the Bus Error Register (BIIC\$_BER), except that the request is not qualified by the bits BIIC\$_HEIE and BIIC\$_SEIE in BIIC\$_BICSR. |
| | BIIC\$_EISENT ² INTR sent. |
| | <22> Reserved to DIGITAL. Must be zero. |
| | BIIC\$_EIINTC ² INTR complete. When set, this bit indicates that the vector for an error interrupt has been successfully transmitted or an INTR command sent under the control of this register has been successfully aborted. |

¹Read-only field.

²Write-one-to-clear bit. Write-type transactions cannot set this bit.

Generic VAXBI Device Support

14.9 BIIC Register Definitions

Table 14–1 (Cont.) Contents of the BIIC Registers

| Field Name | Contents |
|---------------------------|---|
| | <p>BIIC\$_EIINTAB² INTR abort. When set, this bit indicates that an INTR command under the control of this register has been aborted (that is, a NO ACK or illegal confirmation code has been received). This bit is a status bit set by the BIIC and can be reset only by the user interface.</p> <p><31:25>¹ Reserved to DIGITAL. Must be zero.</p> |
| BIIC\$_IDR | Interrupt Destination Register. The low-order word of this register indicates which nodes are to be selected by INTR commands. |
| BIIC\$_IPIMR | Interprocessor Interrupt Mask Register. The high-order word of this register indicates which nodes are permitted to send interprocessor interrupts to this node. |
| BIIC\$_IPIDR | Force-bit IPINTR/STOP Destination Register. The low-order word of this register indicates which nodes are to be targeted by force-bit IPINTR or STOP commands sent by this node. |
| BIIC\$_IPISR ² | IPINTR Source Register. The BIIC stores in the high-order word of this register the decoded ID of a node that sends an IPINTR command to this node. |
| BIIC\$_SAR | Starting Address Register. The Starting Address Register and Ending Address Register define storage blocks in either memory or I/O space. They must not be configured to include node space or multicast space. |
| | The low-order 18 bits of this register must be zero. This means that memories are multiples of 256K bytes. Software should set up the Starting Address Register before the Ending Address Register. |
| BIIC\$_EAR | Ending Address Register. |
| | The low-order 18 bits of this register must be zero. This means that memories are multiples of 256K bytes. Software should set up the Starting Address Register before the Ending Address Register. See the description of the Starting Address Register (BIIC\$_SAR) for further details.. |
| BIIC\$_BCICR | BCI Control Register. |
| | The following fields are defined within BIIC\$_BCICR. |
| | <2:0> ¹ Reserved to DIGITAL. Must be zero. |
| BIIC\$_RTOEVEN | RTO EV enable. |
| BIIC\$_PNXTEN | Pipeline NXT enable. |
| BIIC\$_IPINTREN | IPINTR enable. |
| BIIC\$_INTREN | INTR enable. |
| BIIC\$_BICSREN | BIIC CSR Space enable. |
| BIIC\$_UCSREN | User Interface CSR Space enable. |
| BIIC\$_WINVALEN | WRITE Invalidate enable. |
| BIIC\$_INVALEN | INVAL enable. |
| BIIC\$_IDENT | IDENT enable. |
| BIIC\$_RESEN | RESERVED enable. |

¹Read-only field.

²Write-one-to-clear bit. Write-type transactions cannot set this bit.

Generic VAXBI Device Support

14.9 BIIC Register Definitions

Table 14–1 (Cont.) Contents of the BIIC Registers

| Field Name | Contents |
|----------------|---|
| | BIIC\$_STOPEN STOP enable. |
| | BIIC\$_BDCSTEN BDCST enable. |
| | BIIC\$_MSEN Multicast Space enable. |
| | BIIC\$_IPINTRF IPINTR/STOP force. |
| | BIIC\$_BURSTEN Burst enable. |
| | <31:18> ¹ Reserved to DIGITAL. Must be zero. |
| BIIC\$_WSR | Write Status Register. The following fields are defined within BIIC\$_WSR. |
| | <27:0> ¹ Reserved to DIGITAL. Must be zero. |
| | BIIC\$_GPRO ² Indicates that a VAXBI transaction has written to General Purpose Register 0 (BIIC\$_GPRO). |
| | BIIC\$_GPR1 ² Indicates that a VAXBI transaction has written to General Purpose Register 1 (BIIC\$_GPR1). |
| | BIIC\$_GPR2 ² Indicates that a VAXBI transaction has written to General Purpose Register 2 (BIIC\$_GPR2). |
| | BIIC\$_GPR3 ² Indicates that a VAXBI transaction has written to General Purpose Register 3 (BIIC\$_GPR3). |
| BIIC\$_IPISTPF | Force-Bit IPINTR/STOP Command Register. The following fields are defined within BIIC\$_IPISTPF. |
| | <10:0> ¹ Reserved to DIGITAL. Must be zero. |
| | BIIC\$_MIDEN Master ID Enable. |
| | BIIC\$_CMD These four bits indicate the command code for either an IPINTR or STOP transaction that is initiated by setting the IPINTR/STOP force bit (BIIC\$_INTRF in BIIC\$_BCICR). |
| | <31:16> ¹ Reserved to DIGITAL. Must be zero. |
| BIIC\$_UICR | User Interface Interrupt Control Register. This register controls the operation of interrupts initiated by the device. The following fields are defined within BIIC\$_UICR. |
| | <1:0> ¹ Reserved to DIGITAL. Must be zero. |
| | BIIC\$_UIVECTOR These 12 bits contain the vector used during user interface interrupt sequences (unless the external vector bit (BIIC\$_EXVECTOR in BIIC\$_UICR) is set). The vector is transmitted when this node wins an IDENT arbitration that matches the conditions given in BIIC\$_UICR. |

¹Read-only field.

²Write-one-to-clear bit. Write-type transactions cannot set this bit.

Generic VAXBI Device Support

14.9 BIIC Register Definitions

Table 14–1 (Cont.) Contents of the BIIC Registers

| Field Name | Contents |
|------------------------------|--|
| <14> | Reserved to DIGITAL. Must be zero. |
| BIIC\$V_EXVECTOR | When set, the BIIC solicits the interrupt vector from the node rather than transmitting the vector contained in BIIC\$L_UICR. |
| BIIC\$V_UIFORCE | These four bits correspond to the four interrupt levels (INT <7:4>). When a bit is set, the BIIC generates an interrupt at the indicated level. |
| BIIC\$V_UISENT ² | These four bits correspond to the four interrupt levels (INT <7:4>). A set bit indicates that an INTR command for the corresponding level has been successfully transmitted. |
| BIIC\$V_UIINTC ² | These four bits correspond to the four interrupt levels (INT <7:4>). A set bit indicates that the vector for an interrupt at the corresponding level has been successfully transmitted or that an INTR command sent under the control of this register has been successfully aborted. |
| BIIC\$V_UIINTAB ² | These four bits correspond to the four interrupt levels (INT <7:4>). A set bit indicates that an INTR command at the corresponding level, sent under the control of this register, has been aborted (that is, a NO ACK or illegal confirmation code has been received). |
| BIIC\$L_GPRO | General Purpose Register 0. |
| BIIC\$L_GPR1 | General Purpose Register 1. |
| BIIC\$L_GPR2 | General Purpose Register 2. |
| BIIC\$L_GPR3 | General Purpose Register 3. |

²Write-one-to-clear bit. Write-type transactions cannot set this bit.

15 Loading a Device Driver

You can load a non-DIGITAL-supplied 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.

15.1 Preparing a Driver for Loading into the Operating System

To prepare a device driver for loading, perform the following steps:

- 1 Write the device driver in one or more source files. If the driver comprises several source files, you must insert a .PSECT directive before any generated code in all files except the file that contains the DPTAB and DDTAB macro invocations. The following .PSECT must be used:

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

If a single source file contains the driver, you must not specify any .PSECT directives. The declaration of the DPTAB and DDTAB macros correctly establishes driver program sections (\$\$\$105_PROLOGUE and \$\$\$115_DRIVER, respectively).

- 2 Assemble the source file(s) with the system's macro library (SYS\$LIBRARY:LIB.MLB). For example:

```
$ MACRO MYDRIVER.MAR+SYS$LIBRARY:LIB.MLB/LIBRARY
```

- 3 Link the object file with the VMS global symbol table, which is located in SYS\$SYSTEM and called SYS.STB. If the driver consists of several source files, you must specify the file that contains the driver prologue table as the first file in the list. The linker-options file must contain a BASE statement specifying a zero base for the executable image. The following is an example of the creation of the options file and the LINK command used to link a driver:

```
$ CREATE MYDRIVER.OPT
BASE=0
CTRL/Z
$ LINK /NOTRACE MYDRIVER1[,MYDRIVER2,...],-
_$ MYDRIVER.OPT/OPTIONS,-
_$ SYS$SYSTEM:SYS.STB/SELECTIVE_SEARCH
```

The resulting image must consist of a single image section. The linker will report that the image has no transfer address.

Once you have linked or relinked a driver, you should copy its image to the SYS\$LOADABLE_IMAGES or SYS\$SYSTEM directory. The SYSGEN LOAD and CONNECT commands first search for a driver in the SYS\$LOADABLE_IMAGES directory. If they do not find the driver, they then search the SYS\$SYSTEM directory.

Loading a Device Driver

15.2 Loading a Driver

15.2 Loading a Driver

Once the driver has been linked correctly, it is ready to be loaded. To load the driver into system virtual memory, run the System Generation Utility (SYSGEN) from the system manager's account or from an account having CMKRNL privilege, using the following command:

```
$ RUN SYS$SYSTEM:SYSGEN
```

SYSGEN responds with a prompt and waits for further input:

```
SYSGEN>
```

The *VMS System Generation Utility Manual* describes the full set of SYSGEN commands. The sections that follow describe those commands SYSGEN uses to load drivers:

| SYSGEN Command | Privilege Required |
|--------------------|--------------------|
| LOAD | CMKRNL |
| CONNECT | CMKRNL |
| RELOAD | CMKRNL |
| SHOW/ADAPTER | CMEXEC |
| SHOW/CONFIGURATION | CMEXEC |
| SHOW/DEVICE | CMEXEC |

SYSGEN takes special steps to ensure that drivers that do not adhere to multiprocessing synchronization standards do not coexist in a system with drivers that are properly synchronized. The procedure that SYSGEN follows to accomplish this is discussed in Section 15.3. In addition, SYSGEN provides an automatic configuration service for UNIBUS/Q22 bus devices, as described in Section 15.4.

15.2.1 LOAD Command

To load a device driver, issue the LOAD command.

Note: If the controller has only a single unit attached to it, you can issue the CONNECT command to perform the driver-loading tasks normally performed by the LOAD command, as well as its task of creating the device's I/O database (see Section 15.2.2).

Format

```
LOAD filespec
```

Parameter

filespec

Name of a file containing an executable driver image. The driver-loading procedure compares the name field (DPT\$T_NAME) in the driver prologue table of the driver being loaded with the names of the drivers in the current system configuration. If the procedure discovers that a driver with the same name already exists in the configuration, it will not load the new driver. If it does not find a configured driver with the same name, it loads the new

Loading a Device Driver

15.2 Loading a Driver

driver into contiguous locations in nonpaged pool, and links the DPT into the system's linked list of DPTs (headed by IOC\$GL_DPTLIST).

The LOAD command uses SYS\$LOADABLE_IMAGES as the default device/directory name, and EXE as the default file type. If it cannot find the driver in the SYS\$LOADABLE_IMAGES directory, it searches for it in SYS\$SYSTEM.

Example

```
SYSGEN> LOAD CRDRIVER
```

This command loads the driver found in SYS\$LOADABLE_IMAGES:CRDRIVER.EXE (the card-reader driver).

15.2.2 CONNECT Command

The CONNECT command creates data structures in the I/O database for a specified device. The device-connecting procedure performs the following general functions:

- If the CONNECT command specifies a new device unit on an *existing* controller, it creates a unit control block for the new unit and calls the driver's unit initialization routine.
- If the CONNECT command specifies a device unit on a *new* controller, it creates a device data block, channel request block, interrupt dispatch block, and unit control block and then calls both the controller initialization and unit initialization routine in the driver. (Note that, because system initialization creates the CRB and IDB for a generic VAXBI device, the CONNECT command for such a device omits the creation and initialization of these structures.)

The CONNECT command can also load into system memory a driver that has not been previously loaded. (See the following discussion of the /DRIVERNAME qualifier and the description of the LOAD command in Section 15.2.1 for information on driver loading.)

CAUTION: The database-loading procedure does little error checking. If you specify a vector that has already been defined, the procedure rejects the CONNECT command. However, if the CONNECT command specifies an incorrect CSR address, the I/O database is apt to become corrupted and will likely cause a system failure.

Format

```
CONNECT device
```

Loading a Device Driver

15.2 Loading a Driver

Parameter

device

Name of the device to be connected. Specify the device name in the format *ddcu* where

dd = device code (up to 9 alphabetic characters)

c = controller designation (alphabetic)

u = unit number

For example, LPA0 specifies the line printer (LP) on controller A at unit number 0. When specifying the device name, do *not* follow it with a colon (:).

The device code and controller specification must be a unique and accurate device name and controller combination. If data structures for the specified device/controller already exist, the device-connecting procedure does not create any data structures or perform any initialization operations. If the device/controller name does not accurately name a device, the procedure creates spurious data structures.

The device-connecting procedure examines the I/O database for data structures that support the specified device. The procedure creates the following data structures if they do not exist:

- DDB for the specified device/controller combination (*ddcu*).
- CRB and IDB for the specified controller. The device connecting procedure creates these data structures whenever it creates a DDB for a UNIBUS, MASSBUS, or Q22 bus device.
- UCB for the device unit. The device-connecting procedure creates a UCB whenever it creates a DDB, or when a UCB for the specified device does not exist. If a UCB already exists, the procedure ceases its modifications to the I/O database and continues its other tasks.

After it creates these data structures, the procedure initializes them as follows:

- Performs the initialization operations specified by the DPT_STORE macros in the initialization and reinitialization portions of the DPT.
- Relocates all addresses in the DDT and FDT to absolute system virtual addresses.
- Raises IPL to IPL\$_POWER on the local processor so that initialization is not interrupted.
- If it created a new CRB (or is connecting a generic VAXBI device), calls the controller initialization routine, if one is specified by CRB\$_INTD+VEC\$_INITIAL.
- Calls the unit initialization routine if one is specified by DDT\$_UNITINIT. If the DDT\$_UNITINIT does not specify a unit initialization routine, the device-connecting procedure calls the unit initialization routine (if any) specified by CRB\$_INTD+VEC\$_UNITINIT.

Loading a Device Driver

15.2 Loading a Driver

Required Qualifiers

/[NO]ADAPTER=nexus

Nexus value of the UNIBUS adapter, MASSBUS adapter, or other controller to which the device unit is attached. The nexus can be a number or a generic name as listed by the /ADAPTER qualifier to the SYSGEN command SHOW. (See Section 15.2.4 for a discussion of the SHOW/ADAPTER command.) For generic VAXBI devices, this value is the VAXBI node number. Table 15-1 lists typical nexus assignments for UNIBUS and MASSBUS adapters.

Table 15-1 Conventional Nexus Assignments

| Adapter | VAX-11/725 VAX-11/730 | VAX-11/750 | VAX-11/780 | VAX 8200 | VAX 8530 |
|----------------|--------------------------|------------|--|----------------------------------|---|
| | | | VAX-11/785 VAX 8600 VAX 8650 VAX 8670 | VAX 8250 VAX 8300 VAX 8350 | VAX 8550 VAX 8700 VAX 8800 ¹ |
| UNIBUS | | | | | |
| 0 | 3 | 8 | 3 | 0 | 0 |
| 1 | - | 9 | 4 | - | 0 |
| 2 | - | - | 5 | - | 0 |
| 3 | - | - | 6 | - | 0 |
| MASSBUS | | | | | |
| 0 | - | 4 | 8 | - | - |
| 1 | - | 5 | 9 | - | - |
| 2 | - | 6 | 10 | - | - |
| 3 | - | - | 11 | - | - |

¹The VAX 8530/8550/8700/8800 systems can provide up to four VAXBI buses. A DWBUA can be situated only at node 0 on a VAXBI and, thus, can have a nexus value of 0, 16, 32, or 48.

All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X). Issue the CONNECT command with the /NOADAPTER qualifier to connect drivers associated with software devices. The mailbox driver is an example of this type of driver.

/CSR=csr-addr

UNIBUS or Q22 bus address of the device's control and status register (CSR). All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X). Table 15-2 provides additional information on vector and CSR assignments for UNIBUS and Q22 bus devices.

/CSR_OFFSET=value

Offset from the CSR address of a multiple-device controller board to the CSR address of the device. The /CSR_OFFSET qualifier is only required for a multi-device board, such as the DMF32. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X). Table 15-2 provides additional information on vector and CSR assignments for UNIBUS and Q22 bus devices.

Loading a Device Driver

15.2 Loading a Driver

/VECTOR=vector-addr

Q22 bus or UNIBUS address of the interrupt vector for the device. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X). Table 15-2 provides additional information on vector and CSR assignments for UNIBUS and Q22 bus devices.

/VECTOR_OFFSET=value

Offset from the interrupt vector of a multiple-device board to the interrupt vector of the device being connected. The /VECTOR_OFFSET qualifier is only required for a multi-device board, such as the DMF32. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X). Table 15-2 provides additional information on vector and CSR assignments for UNIBUS and Q22 bus devices.

Optional Qualifiers

/NUMVEC=vector-cnt

Number of interrupt vectors for the device. If this qualifier is omitted, the default number of vectors is 1. The number specified by the /VECTOR qualifier is the address of the lowest vector. Vectors must be contiguous.

/DRIVERNAME=driver

Name of the driver for the device to be connected. If the driver for the specified device has not yet been loaded, the CONNECT command will load its driver. First, it will attempt to load the driver whose name is specified in this qualifier, defaulting to a file type of EXE in device/directory SYS\$LOADABLE_IMAGES. (If it cannot find the driver in SYS\$LOADABLE_IMAGES, the CONNECT command checks SYS\$SYSTEM.)

If the /DRIVERNAME qualifier is omitted, CONNECT follows one of two procedures to supply a default name. If the device to be connected is the first unit on the controller, CONNECT concatenates the first two characters of the device code with "DRIVER," (for example, LPDRIVER). Otherwise, CONNECT obtains the driver name from the field DDB\$T_DRVNAME.

Consult the SYSGEN device table in Table 15-2 for the driver names of the devices supported by VMS.

/ADPUNIT=unit-number

Unit number of a device on the MASSBUS adapter. The unit number for a disk drive is the number of the plug on the drive. For magnetic tape drives, the unit number corresponds to the tape controller's number.

/MAXUNITS=max-unit-cnt

Maximum number of units attached to the controller. This number determines the size of the UCB list appended to the IDB. If specified, this value overrides the maximum number of units designated in the DPT. The maximum number of units is stored in the field IDB\$W_UNITS.

Example

```
SYSGEN> CONNECT LPA0 /ADAPTER=UBO/CSR=%0777514/VECTOR=%0200
```

This command loads the driver LPDRIVER, if it is not already loaded, and creates the data structures (DDB, CRB, IDB, and UCB) needed to describe LPA0. It also causes the driver's controller and unit initialization routines to be executed.

15.2.3 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 is busy, the driver cannot be reloaded; SYSGEN issues an error message.

CAUTION: Use the RELOAD command only when all devices supported by the driver are inactive. The checks for activity made by the RELOAD command might not detect all device activity, and changing a driver while an I/O request is being processed will cause a system failure.

Format

RELOAD filespec

Parameter

filespec

Name of a file containing an executable driver image. The driver-reloading procedure compares the name DPT\$T_NAME of the driver being loaded with the names of the drivers in the current system configuration. If no such driver is configured, the driver-reloading procedure loads the driver as described in the discussion of the LOAD command in Section 15.2.1.

If the SYSGEN reloading procedure finds a driver with the specified name in the configuration, it first determines that the current driver can be replaced in the following steps:

- Confirms that the DPT\$V_NOUNLOAD flag of the current driver is not set.
- Ensures that no devices that use the current driver are busy, as indicated by the UCB\$V_BSY bit set in UCB\$L_STS.
- If these checks succeed, calls the current driver's driver unloading routine, if one has been specified in the **unload** argument of the DPTAB macro.

The driver unloading routine executes in process context at IPL\$POWER. It cannot lower IPL or obtain spin locks.

Registers at the time of the call contain the following values:

| Register | Value |
|----------|----------------|
| R6 | Address of DDB |
| R10 | Address of DPT |

A driver unloading routine can take steps to ensure that no thread of code or structure exists in the system that may reference the space occupied by the version of the driver about to be unloaded, a timer-queue element (TQE), for instance.

A driver unloading routine can use COM\$DRVDEALMEM to return system buffers allocated by the driver to nonpaged pool. The driver unloading routine returns status in R0 to the driver-reloading procedure.

Loading a Device Driver

15.2 Loading a Driver

Upon receiving success status, the driver-reloading procedure replaces the current driver with the new driver in the following manner:

- 1 Loads the new driver into contiguous locations in nonpaged pool.
- 2 Searches the I/O database for references to the driver. If any DDB refers to the driver being reloaded, the driver-reloading procedure must reinitialize data structure fields according to the reinitialization instructions in the new DPT. (see Section 6.1).

Fields that must be reinitialized when a driver is reloaded include those that contain relative addresses within the driver:

- Addresses of the interrupt service routines
 - Addresses of the unit and controller initialization routines
 - Address of the driver dispatch table
- 3 Calls the driver's controller initialization routine. (It does not call the unit initialization routine.)
 - 4 Removes the newly replaced driver from the system's linked list of DPTs (headed by IOC\$GL_DPTLIST) and deallocates the nonpaged system space the old driver occupied.
 - 5 Links the address of the new DPT to the system's DPT list.

15.2.4 SHOW/ADAPTER Command

The SHOW/ADAPTER command displays nexus numbers and generic names of UNIBUS and MASSBUS adapters, VAXBI adapters, memory controllers, and interconnection devices such as the DR32 and CI. Use of the SHOW/ADAPTER command requires CMEXEC privilege.

Format

SHOW/ADAPTER

Example

```
SYSGEN> SHOW/ADAPTER
```

```
CPU Type: VAX 8530
```

| Nexus | Generic Name or Description |
|-------|------------------------------------|
| 32 | CIO |
| 34 | BI - NBIB Adapter |
| 38 | BI Combo Board |
| 39 | BI - AIE Adapter with NI port only |
| 48 | UBO |
| 50 | BI - NBIB Adapter |
| 52 | BI - Disk Adapter (KDB50) |

15.2.5 SHOW/CONFIGURATION Command

The SHOW/CONFIGURATION command displays the device name, number of units, nexus number and type, and shows the CSR and vector addresses of devices connected to or autoconfigured in the system.

Format

SHOW/CONFIGURATION

Optional Qualifiers

/ADAPTER=nexus

Nexus value of the UNIBUS adapter, MASSBUS adapter, or other interconnect to be displayed. The nexus value can be expressed as an integer or as one of the generic names listed by the SHOW/ADAPTER command.

/COMMAND_FILE

Option by which you instruct SYSGEN to format all device data produced by the SHOW/CONFIGURATION command into CONNECT/ADAPTER=nexus commands and write them to a specified output file. By executing the commands in this file, you can remove a device from floating address space without completely reconnecting the CSR and vector addresses of the remaining devices. See the *VMS System Generation Utility Manual* for more details.

/OUTPUT=filespec

Name of a file into which SHOW/CONFIGURATION is to write device configuration information.

Example

```
SYSGEN> SHOW/CONFIGURATION/ADAPTER=UB1
```

```
System CSR and Vectors on 24-JUL-1988 14:58:26.08
```

```
Name: LPA Units: 1 Nexus:4 (UBA) CSR: 777514 Vector1: 200 Vector2: 000
Name: DYA Units: 2 Nexus:4 (UBA) CSR: 777170 Vector1: 264 Vector2: 000
Name: XMA Units: 1 Nexus:4 (UBA) CSR: 760070 Vector1: 300 Vector2: 304
Name: XMB Units: 1 Nexus:4 (UBA) CSR: 760100 Vector1: 310 Vector2: 314
Name: XMC Units: 1 Nexus:4 (UBA) CSR: 760110 Vector1: 320 Vector2: 324
Name: TTA Units: 8 Nexus:4 (UBA) CSR: 760130 Vector1: 330 Vector2: 334
Name: TTB Units: 8 Nexus:4 (UBA) CSR: 760140 Vector1: 340 Vector2: 344
Name: TTC Units: 8 Nexus:4 (UBA) CSR: 760150 Vector1: 350 Vector2: 354
Name: TTD Units: 8 Nexus:4 (UBA) CSR: 760160 Vector1: 360 Vector2: 364
Name: TTE Units: 8 Nexus:4 (UBA) CSR: 760170 Vector1: 370 Vector2: 374
```

15.2.6 SHOW/DEVICE Command

The SHOW/DEVICE command displays the following information:

- Name of the driver
- Starting virtual address of the driver (that is, the address of its DPT)
- Ending virtual address of the driver
- Generic device/controller name associated with the driver

Loading a Device Driver

15.2 Loading a Driver

- Addresses of the DDB, CRB, and IDB for the generic device/controller supported by the driver
- Unit number and UCB address of each device unit associated with the driver

The SHOW/DEVICE command requires CMEXEC privilege.

Format

SHOW/DEVICE [=driver-name]

Parameter

driver-name

Name of the driver for which the information is to be displayed. If a driver name is not specified, the command displays information about all drivers and devices known to the system.

Example

```
SYSGEN> SHOW/DEVICE=TMDDRIVER
```

```
__DRIVER__START__END__DEV__DDB____CRB____IDB____UNIT__UCB
TMDDRIVER 8009DF00 8009F020
                                     MTA 800BA660 800BA6C0 800BA360
                                               0 8009F020
                                               1 8009F0C0
```

15.3 Loading Uniprocessing and Multiprocessing Drivers

In a VMS multiprocessing environment, the presence of a device driver that does not adhere to multiprocessing synchronization conventions can be fatal to proper system functions. VMS takes steps to either prohibit the enabling of multiprocessing in a VAX system that has such a driver present or prevent the loading of such a driver if multiprocessing has already been enabled.

To accomplish this, the VMS driver-loading routine assumes that any driver that can run in a VMS multiprocessing environment uses the spin lock synchronization macros and loads the appropriate I/O database fields. (See Section G.3 for information on how to produce a driver that can execute in a VMS multiprocessing environment.) Use of the spin lock synchronization macros causes VMS to set the SMP-modified bit in the DPT (DPT\$V_SMPMOD in DPT\$L_FLAGS).

If multiprocessing has *not* been enabled on the system, the driver-loading routine checks the SMP-modified bit in the DPT and takes either of the following actions:

- If the SMP-modified bit is set, the driver-loading routine loads the driver and calls its controller and unit initialization routines, as discussed in Section 15.2.
- If the SMP-modified bit is *not* set, the driver-loading mechanism sets the unmodified-driver bit (SMP\$V_UNMOD_DRIVER) in SMP\$GL_FLAGS, thus prohibiting the subsequent enabling of multiprocessing on the system. It then loads the driver and calls its controller and unit initialization routines. If such a driver has been successfully loaded into a VMS system, you cannot subsequently enable multiprocessing.

Loading a Device Driver

15.3 Loading Uniprocessing and Multiprocessing Drivers

If multiprocessing is currently enabled on the system, the driver-loading mechanism checks the SMP-modified bit in the DPT and takes either of the following actions:

- If the SMP-modified bit is set, the driver-loading mechanism loads the driver and calls its controller and unit initialization routines.
- If the SMP-modified bit is *not* set, the driver-loading mechanism does not load the driver, returning the error status `SS$_NONSMODRV` to its caller.

15.4 The SYSGEN Autoconfiguration Facility

Traditionally, SYSGEN is invoked near the end of system initialization processing during the execution of the system startup command procedure. This procedure generally issues a `SYSGEN AUTOCONFIGURE ALL` command, the result of which is that SYSGEN scans various device tables to determine devices VMS expects to be connected to each UNIBUS, Q22 bus, MASSBUS, and VAXBI bus configured in the system. Ultimately, as the autoconfigure facility discovers the data structures associated with the devices recognized by VMS, it loads the associated device drivers and invokes their initialization routines.

To configure devices attached to the UNIBUS or Q22 bus, SYSGEN goes through the steps described in subsequent sections of this chapter. Because the autoconfigure facility cannot recognize non-DIGITAL-supplied VAXBI devices, the system startup procedure (or a later invocation of SYSGEN) must explicitly request that SYSGEN connect the device. SYSGEN responds to such explicit requests by utilizing the data structures created by the VMS adapter initialization module for the unknown VAXBI device to load the associated device driver and invoke its initialization routines.

SYSGEN automatically configures a UNIBUS or Q22 bus as follows:

- It initializes the base of floating space to `3008` and `7600108` for vectors and CSRs, respectively.
- It tests fixed and floating CSR address space for all known DIGITAL devices.
- When a device is found at a CSR, SYSGEN reserves floating CSR and vector space for that device, if necessary.
- It searches for the name of the driver associated with the device by checking the SYSGEN device table (shown in Table 15-2) and the directory `SYS$LOADABLE_IMAGES` (or `SYS$SYSTEM`). If the driver has already been loaded or exists as an image file in `SYS$LOADABLE_IMAGES` (or `SYS$SYSTEM`), SYSGEN creates and initializes the I/O database for that device and loads the driver's image if necessary. If the device at the CSR is supported by VMS and SYSGEN cannot locate its associated driver's image, it generates an error message. If the device is unsupported and has no corresponding driver's image, SYSGEN ignores the condition.

Loading a Device Driver

15.4 The SYSGEN Autoconfiguration Facility

15.4.1 SYSGEN Device Table

DIGITAL-supplied devices are attached to the UNIBUS or Q22 bus according to the following basic rules:

- 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.

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 Table 15-2, lists all the type A and type B devices supported by VMS. It also lists the type C and type D devices that are recognized by SYSGEN's autoconfiguration procedure.

The base of floating vector space is 300_8 . The base of floating CSR space is 760010_8 .

Table 15-2 lists the characteristics of all devices recognized by SYSGEN. This table indicates the following information for each device type:

- Device name
- Device controller name
- Interrupt vector
- Number of interrupt vectors per controller
- Vector alignment factor
- Address of the first device register for each controller recognized by SYSGEN (the first register is usually, but not always, the CSR)
- Number of registers per controller
- Device driver name
- Indication of whether the driver is or is not supported

Loading a Device Driver

15.4 The SYSGEN Autoconfiguration Facility

Devices not listed in the SYSGEN device table include the following:

- Non-DIGITAL-supplied devices with fixed CSR and vector addresses. These devices have no effect on autoconfiguration. Customer-built devices should be assigned CSR and vector addresses beyond the floating address space reserved for DIGITAL-supplied devices.
- Those DIGITAL-supplied floating-vector devices that the AUTOCONFIGURE command does not recognize. Use the CONNECT command to attach these devices to the system.

Table 15–2 SYSGEN Device Table

| Device Name | Controller Name | Vector | Number of Vectors | Vector Alignment | CSR/Rank | Register Alignment | Driver Name | Support |
|-------------|-----------------|--------|-------------------|------------------|------------------|--------------------|-------------|---------|
| CR | CR11 | 230 | 1 | — | 777160 | — | CRDRIVER | Yes |
| DM | RK611 | 210 | 1 | — | 777440 | — | DMDRIVER | Yes |
| LP | LP11 | 200 | — | — | 777514 | — | LPDRIVER | Yes |
| | | 170 | | | 764004 | | | |
| | | 174 | | | 764014 | | | |
| | | 270 | | | 764024 | | | |
| | | 274 | | | 764034 | | | |
| DL | RL11 | 160 | 1 | — | 774400 | — | DLDRIVER | Yes |
| MS | TS11 | 224 | 1 | — | 772520 | — | TSDRIVER | Yes |
| DY | RX211 | 264 | 1 | — | 777170 | — | DYDRIVER | Yes |
| DQ | RB730 | 250 | 1 | — | 775606 | — | DQDRIVER | Yes |
| PU | UDA | 154 | 1 | — | 772150 | — | PUDRIVER | Yes |
| PT | TU81 | 260 | 1 | — | 774500 | — | PUDRIVER | Yes |
| XE | UNA | 120 | 1 | — | 774510 | — | XEDRIVER | Yes |
| XQ | QNA | 120 | 1 | — | 774440 | — | XQDRIVER | Yes |
| OM | DC11 | Float | 2 | 8 | 774000 | — | OMDRIVER | No |
| | | | | | 774010 | | | |
| | | | | | 774020 | | | |
| | | | | | 774030 | | | |
| | | | | | . | | | |
| | | | | | . | | | |
| | | | | | 32 units maximum | | | |
| DD | TU58 | Float | 2 | 8 | 776500 | — | DDRIVER | Yes |
| | | | | | 776510 | | | |
| | | | | | 776520 | | | |
| | | | | | 776530 | | | |
| | | | | | . | | | |
| | | | | | . | | | |
| | | | | | 16 units maximum | | | |

Loading a Device Driver

15.4 The SYSGEN Autoconfiguration Facility

Table 15–2 (Cont.) SYSGEN Device Table

| Device Name | Controller Name | Vector | Number of Vectors | Vector Alignment | CSR/Rank | Register Alignment | Driver Name | Support |
|-------------|-----------------|--------|-------------------|------------------|------------------|--------------------|-------------|---------|
| OB | DN11 | Float | 1 | 4 | 775200 | — | OBDRIVER | No |
| | | | | | 775210 | | | |
| | | | | | 775220 | | | |
| | | | | | 775230 | | | |
| | | | | | 16 units maximum | | | |
| YM | DM11B | Float | 1 | 4 | 770500 | — | YMDRIVER | No |
| | | | | | 770510 | | | |
| | | | | | 770520 | | | |
| | | | | | 770530 | | | |
| | | | | | 16 units maximum | | | |
| OA | DR11C | Float | 2 | 8 | 767600 | — | OADRIVER | No |
| | | | | | 767570 | | | |
| | | | | | 767560 | | | |
| | | | | | 767550 | | | |
| | | | | | 16 units maximum | | | |
| PR | PR611 | Float | 1 | 8 | 772600 | — | PRDRIVER | No |
| | | | | | 772604 | | | |
| | | | | | 772610 | | | |
| | | | | | 772614 | | | |
| | | | | | 8 units maximum | | | |
| PP | PP611 | Float | 1 | 8 | 772700 | — | PPDRIVER | No |
| | | | | | 772704 | | | |
| | | | | | 772710 | | | |
| | | | | | 772714 | | | |
| | | | | | 8 units maximum | | | |

Loading a Device Driver

15.4 The SYSGEN Autoconfiguration Facility

Table 15–2 (Cont.) SYSGEN Device Table

| Device Name | Controller Name | Vector | Number of Vectors | Vector Alignment | CSR/Rank | Register Alignment | Driver Name | Support |
|-------------|-----------------|--------|-------------------|------------------|---|--------------------|-------------|---------|
| OC | DT11 | Float | 2 | 8 | 777420 777422 777424 777426 . . . 8 units maximum | — | OCDRIVER | No |
| OD | DX11 | Float | 2 | 8 | 776200 776240 | — | ODDRIVER | No |
| YL | DL11C | Float | 2 | 8 | 775610 775620 775630 775640 . . . 31 units maximum | — | YLDRIVER | No |
| YJ | DJ11 | Float | 2 | 8 | Float | 8 | YJDRIVER | No |
| YH | DH11 | Float | 2 | 8 | Float | 16 | YHDRIVER | No |
| OE | GT40 | Float | 4 | 8 | 772000 772010 | — | OEDRIVER | No |
| LS | LPS11 | Float | 6 | 8 | 770400 | — | LSDRIVER | No |
| OR | DQ11 | Float | 2 | 8 | Float | 8 | ORDRIVER | No |
| OF | KW11W | Float | 2 | 8 | 772400 | — | OFDRIVER | No |
| XU | DU11 | Float | 2 | 8 | Float | 8 | XUDRIVER | No |
| XV | DV11 | Float | 3 | 8 | 775000 775040 775100 775140 | — | XVDRIVER | No |
| OG | LK11 | Float | 2 | 8 | Float | 8 | OGDRIVER | No |
| XM | DMC11 | Float | 2 | 8 | Float | 8 | XMDRIVER | Yes |
| TTA | DZ11 | Float | 2 | 8 | Float | 8 | DZDRIVER | Yes |
| XK | KMC11 | Float | 2 | 8 | Float | 8 | XKDRIVER | No |
| OH | LPP11 | Float | 2 | 8 | Float | 8 | OHDRIVER | No |
| OI | VMV21 | Float | 2 | 8 | Float | 8 | OIDRIVER | No |
| OJ | VMV31 | Float | 2 | 8 | Float | 16 | OJDRIVER | No |
| OK | DWR70 | Float | 2 | 8 | Float | 8 | OKDRIVER | No |
| DL | RL11 | Float | 1 | 4 | Float | 8 | DLDRIVER | Yes |

Loading a Device Driver

15.4 The SYSGEN Autoconfiguration Facility

Table 15–2 (Cont.) SYSGEN Device Table

| Device Name | Controller Name | Vector | Number of Vectors | Vector Alignment | CSR/Rank | Register Alignment | Driver Name | Support |
|-------------|-----------------|--------|-------------------|------------------|--------------------------------------|--------------------|-------------|---------|
| MS | TS11 | Float | 1 | 4 | 772524 772530 772534 | — | TSDRIVER | Yes |
| LA | LPA11 | Float | 2 | 8 | 770460 | — | LADRIVER | Yes |
| LA | LPA11 | Float | 2 | 8 | Float | 16 | LADRIVER | Yes |
| OL | KW11C | Float | 2 | 8 | Float | 8 | OLDRIVER | No |
| DY | RX211 | Float | 1 | 4 | Float | 8 | DYDRIVER | Yes |
| XA | DR11W | Float | 1 | 4 | Float | 8 | XADRIVER | Yes |
| XB | DR11B | 124 | — | — | 772410 | — | XBDRIVER | No |
| XB | DR11B | Float | 1 | 4 | 772430 | — | XBDRIVER | No |
| XB | DR11B | Float | 1 | 4 | Float | 8 | XBDRIVER | No |
| XD | DMP11 | Float | 2 | 8 | Float | 8 | XDDRIVER | Yes |
| ON | DPV11 | Float | 2 | 8 | Float | 8 | ONDRIVER | No |
| IS | ISB11 | Float | 2 | 8 | Float | 8 | ISDRIVER | No |
| XD | DMV11 | Float | 2 | 8 | Float | 16 | XDDRIVER | No |
| XE | UNA | Float | 1 | 4 | Float | 8 | XEDRIVER | No |
| XQ | QNA | Float | 1 | 4 | 774460 | — | XQDRIVER | Yes |
| PU | UDA | Float | 1 | 4 | Float | 4 | PUDRIVER | Yes |
| XS | KMS11 | Float | 3 | 8 | Float | 16 | XSDRIVER | No |
| XP | PCL11 | Float | 2 | 8 | 764200 764240 764300 764340 | — | XPDRIVER | No |
| VB | VS100 | Float | 1 | 4 | Float | 16 | VBDRIVER | No |
| PT | TU81 | Float | 1 | 4 | Float | 4 | PUDRIVER | Yes |
| OQ | KMV11 | Float | 2 | 8 | Float | 16 | OQDRIVER | No |
| UK | KCT32 | Float | 2 | 8 | 764400 764440 764500 764540 | — | UKDRIVER | No |
| IX | IEQ11 | Float | 2 | 8 | 764100 | — | IXDRIVER | No |
| TX | DHV11 | Float | 2 | 8 | Float | 16 | YFDRIVER | Yes |
| DT | TC11 | 214 | 1 | — | 777340 | — | DTDRIVER | No |
| VC | VCB01 | Float | 2 | 1 | 777200 | — | VCDRIVER | Yes |
| VC | VCB01 | Float | 2 | 1 | Float | 64 | VCDRIVER | Yes |
| OT | LNV11 | Float | 1 | 4 | 776200 | — | OTDRIVER | No |
| LD | LNV21 | Float | 1 | 4 | Float | 16 | LDDRIVER | No |
| ZQ | QTA | Float | 1 | 4 | 772570 | — | ZQDRIVER | No |
| ZQ | QTA | Float | 1 | 4 | Float | 8 | ZQDRIVER | No |

Loading a Device Driver

15.4 The SYSGEN Autoconfiguration Facility

The **defunits** argument to the DPTAB macro specifies a default number of units to be configured on each controller. The DPTAB macro copies this value to the DPT\$W_DEFUNITS field in the DPT. The SYSGEN autoconfiguration facility reads this field and creates UCBs numbered zero through the default unit number minus one. The default value of **defunits** is 1.

The **deliver** argument to the DPTAB macro specifies the address of a driver-specific unit delivery routine. An offset to this routine is stored in the DPT\$W_DELIVER field in the DPT. When the **deliver** argument is present, the SYSGEN autoconfiguration facility calls the unit delivery routine once for each unit, the number of which is specified in the **defunits** argument.

Note: Because the autoconfigure facility will never be called for a non-DIGITAL-supplied device, any unit delivery routine that a VAXBI device driver may include will never be called.

The unit delivery routine prevents the creation of UCBs for devices that do not respond to a test for their presence.

If the unit delivery routine returns a true status in R0, the unit is configured. If the status in R0 is false, the autoconfiguration facility does not configure the device. If the **deliver** argument is not used, the unit delivery feature is disabled.

SYSGEN calls the unit delivery routine with a JSB instruction in the following context:

- Interrupt priority level is at IPL\$_POWER.
- R0 through R2 are available for use.
- R3 contains the address of the IDB, if one exists. If none exists, the value contained in R3 is zero.
- R4 contains the address of the CSR for the controller.
- R5 contains the number of the unit that the routine must decide whether or not to configure.
- R6 contains the base address of UNIBUS adapter I/O space.
- R7 contains the address of the configuration control block (ACF).
- R8 contains the address of the ADP.

The configuration control block is described in Figure A-2 and Table A-1.

A driver may or may not specify a unit delivery routine. For instance, the DZ11's device driver specifies 8 as the default unit number, but provides no routine to configure eight terminal units automatically for each DZ11's CSR. The RK611 device driver specifies 8 as the default number of units and also specifies the address of a unit delivery routine that is called once for each of the eight possible devices on the controller.

Loading a Device Driver

15.4 The SYSGEN Autoconfiguration Facility

15.4.3 Floating-Vector Address Calculation

To calculate the floating-vector address of a device, SYSGEN 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:

$$\text{CFVB} + (4 * \text{number-of-vectors}) \rightarrow \text{CFVB}$$

15.4.4 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 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 by adding eight to the rounded CFCB:

$$\text{CFCB} + 8 \rightarrow \text{CFCB}$$

15.4.5 Rules for Configuration

The formulas described in Sections 15.4.3 and 15.4.4 reduce to the following maxims:

- Devices with fixed CSR addresses and fixed vector addresses must be attached according to the SYSGEN device table settings.
- 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.

Loading a Device Driver

15.4 The SYSGEN Autoconfiguration Facility

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 devices in the table because future VMS maintenance updates might add new devices to the SYSGEN device table.¹

¹ UNIBUS addresses 764100₈ through 767776₈ are available for non-DIGITAL-supplied devices.

16 Debugging a Device Driver

DELTA and XDELTA are debugging tools that can be used to monitor the execution of user programs and the VMS operating system. When you link DELTA with a user image that runs in a nonprivileged process, DELTA is a user-mode debugging tool. When run in a privileged process, however, DELTA acts as a multimode debugger; it allows you to debug in user mode or to change to kernel mode for debugging. However, DELTA does not support debugging at elevated IPLs.

XDELTA is syntactically identical to DELTA but also allows you to debug code that executes at an elevated IPL. XDELTA is used for stand-alone debugging of driver code and the executive.

This chapter primarily describes the use of XDELTA as a tool for debugging an executing driver image. 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.

The chapter includes discussions of two additional topics:

- Detection and analysis of driver errors in a VMS multiprocessing system
- Detection of corruption in nonpaged pool and the ways in which the corrupting code can be discovered

These topics supplement information presented in the *VMS System Dump Analyzer Utility Manual*.

16.1 Bootstrapping the System with XDELTA

Under 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 the breakpoint at the beginning of VMS initialization. (The BREAKPOINTS system parameter, by default, enables a breakpoint at the end of system initialization. See Section 16.2 for additional information.) Table 16-1 describes the possible values of these flags. Following a boot that includes XDELTA, executing a BPT instruction causes control to transfer to a fault handler located in XDELTA.

Debugging a Device Driver

16.1 Bootstrapping the System with XDELTA

Table 16–1 Boot Flags That Control the Loading of XDELTA

| Flag Value (f) | Meaning |
|----------------|--|
| 0 | Normal nonstop bootstrap (default) |
| 1 | Stop in SYSBOOT (equivalent to @DxyGEN on the VAX–11/780) |
| 2 | Include XDELTA with the system but do not take the initial breakpoint |
| 6 | Include XDELTA with the system and take the initial breakpoint |
| 7 | Include XDELTA with the system, stop in SYSBOOT and take the initial breakpoint at system initialization (equivalent to @DxyXDT on the VAX–11/780) |

The procedures for bootstrapping the system with XDELTA differ depending upon the system on which the operating system is running. Some VAX systems that use a console block storage device supply a special boot command file that automatically includes XDELTA in the system and causes the processor to stop in SYSBOOT and take the initial breakpoint at system initialization. When booting other systems, you must specify the appropriate flag value in the BOOT command. Table 16–2 lists some recommended methods for booting with XDELTA. See your system’s installation and operations guides for additional information.

Table 16–2 Recommended Methods for Bootstrapping with XDELTA

| Boot Commands | Explanation |
|--|--|
| MicroVAX 3600-Series, MicroVAX II, MicroVAX I, and VAX–11/750¹ Systems | |
| B[/f] devname | <p>B is the console BOOT command. The flags (f) parameter is a 32-bit hexadecimal integer loaded into R5 as input to VMB.EXE, the primary bootstrap program. See Table 16–1 for a list of its possible values.</p> <p>Using the format <i>ddcu</i>, specify the name of the device that contains the volume to be bootstrapped. You must supply both controller (c) and unit (u) identifiers; there are no defaults. If you omit <i>devname</i>, the f parameter is ignored.</p> <p>The following example bootstraps a MicroVAX II system from DUA0.²</p> <pre>>>>B/7 DUA0 SYSBOOT> SYSBOOT>CONTINUE</pre> |

¹The console TU58 of the VAX–11/750 system contains command files (DMAXDT.COM and DBAXDT.COM) analogous to those supplied for the VAX–11/780. See your system’s installation and operations guides for additional information.

²At the SYSBOOT prompt enter other required SYSBOOT commands and conclude the boot operation with a CONTINUE command. If you do not set or load system parameters with a USE command, the system uses parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, clear the BUGREBOOT system parameter.

Debugging a Device Driver

16.1 Bootstrapping the System with XDELTA

Table 16–2 (Cont.) Recommended Methods for Bootstrapping with XDELTA

| Boot Commands | Explanation |
|---|--|
| VAX 6200-Series Systems | |
| B[/R5:f] [/XMI:xmi_node_id] [/BI:vaxbi_node_id] devname | <p>B is the console BOOT command. The flags (<i>f</i>) parameter is a 32-bit hexadecimal integer loaded into R5 as input to VMB.EXE, the primary bootstrap program. See Table 16–1 for a list of its possible values.</p> <p>Specify <i>devname</i> in the format <i>dduuu</i>, identify the location of the memory-interconnect-to-VAXBI adapter (XBIA) in /XMI qualifier and the VAXBI node id of the device in the /BI qualifier. (You can substitute a symbolic name for these qualifiers as discussed in your processor-specific operations guide.)</p> <p>The console places the specified unit number (<i>uu</i>) in R3 and executes the procedure <i>dddBOO.COM</i>. If you do not specify <i>devname</i>, the console executes DEFBOO.COM. To use the /R5 qualifier to the BOOT command, you must have previously removed or commented out the DEPOSIT R5 command in the procedure to be executed.</p> <p>The following example bootstraps a VAX 6240 system from the boot disk at node 3 of the VAXBI bus at node 15 of the XMI.²</p> <pre>>>>B/R5:7/XMI:E/BI:3 DU0 SYSBOOT> SYSBOOT>CONTINUE</pre> |
| VAX 8200/8250/8300/8350 and VAX 8530/8550/8700/8800/8830/8840³Systems | |
| B[/R5:f] <i>devname</i> | <p>B is the console BOOT command. The flags (<i>f</i>) parameter is a 32-bit hexadecimal integer loaded into R5 as input to VMB.EXE, the primary bootstrap program. See Table 16–1 for a list of its possible values.</p> <p>For the VAX 8530/8550/8700/8800/8830/8840, specify <i>devname</i> in the format <i>dduuu</i>. The console places the specified unit number (<i>uuu</i>) in R3 and executes the procedure <i>dddBOO.COM</i>. If you do not specify <i>devname</i>, the console executes DEFBOO.COM.</p> <p>To use the /R5 qualifier to the BOOT command for VAX 8530/8550/8700/8800 systems, you must have previously removed or commented out the DEPOSIT R5 command in the procedure to be executed. For VAX 8830/8840 systems, data specified in the /R5 qualifier to the BOOT command overrides data specified in the DEPOSIT R5 command in the boot command procedure.</p> <p>For the VAX 8200/8250/8300/8350, specify <i>devname</i> in the format <i>ddxu</i>, where <i>x</i> represents the number of the VAXBI node to which the boot device unit is attached. If you do not specify <i>devname</i>, the console boots from the default boot device.</p> <p>The following example bootstraps a VAX 8200 system from the boot disk at VAXBI node 4.²</p> <pre>>>>B/R5:7 DU40 SYSBOOT> SYSBOOT>CONTINUE</pre> |

²At the SYSBOOT prompt enter other required SYSBOOT commands and conclude the boot operation with a CONTINUE command. If you do not set or load system parameters with a USE command, the system uses parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, clear the BUGREBOOT system parameter.

³Note that the console prompt for the VAX 8830 and 8840 systems is *PS-CIO-0>* and not *>>>*.

Debugging a Device Driver

16.1 Bootstrapping the System with XDELTA

Table 16–2 (Cont.) Recommended Methods for Bootstrapping with XDELTA

| Boot Commands | Explanation |
|--|--|
| VAX–11/780 and VAX–11/785 Systems | |
| @DMAXDT @DBAXDT | Use either DMAXDT.COM or DBAXDT.COM, depending upon the boot device. The following example boots from DMA0, first depositing the value 0 in R3. ² >>>DEPOSIT R3 0 >>>@DMAXDT SYSBOOT> SYSBOOT>CONTINUE |
| VAX–11/730 and VAX–11/725 Systems | |
| @DQAXDT @DQOXDT | Use either DQAXDT.COM or DQOXDT.COM, depending upon the boot device. The following example boots from DQA1, first depositing the value 1 in R3. When the boot device is DQA0, you can omit this step and execute DQOXDT.COM. ² >>>D/G/L 3 1 >>>@DQAXDT SYSBOOT> SYSBOOT>CONTINUE |
| VAX 8600/8650/8670 Systems | |
| @DUOXDT | Use DUOXDT.COM, if available on the console media, according to the method described for the VAX–11/780. Otherwise, perform a normal bootstrap using the available <i>dduGEN.COM</i> or <i>dduBOO.COM</i> according to the following method: Use the <i>/NOSTART</i> qualifier in the BOOT command to cause the processor to pause and await console commands after it boots. After a variety of progress messages are displayed, the console prompt reappears. First, determine a value for the flag that controls XDELTA loading (see Table 16–1). Then, examine the current value of R5; if it is nonzero (for instance, it is the system root number), perform an inclusive-OR operation upon it and your selected XDELTA flag value. ² >>>BOOT/NOSTART SYSBOOT>EXAMINE R5 SYSBOOT>DEPOSIT R5 7 SYSBOOT> SYSBOOT>CONTINUE |

²At the SYSBOOT prompt enter other required SYSBOOT commands and conclude the boot operation with a CONTINUE command. If you do not set or load system parameters with a USE command, the system uses parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, clear the BUGREBOOT system parameter.

Debugging a Device Driver

16.2 Proceeding from the Initial Breakpoints

16.2 Proceeding from the Initial Breakpoints

Before stopping at any breakpoints that may be defined in driver code, the VAX processor can stop at either or both of two breakpoints in system initialization.

The breakpoint at the end of system initialization is enabled by the default setting of the BREAKPOINTS system parameter. The breakpoint at the beginning is enabled by the appropriate value of the boot flag as described in Table 16-1.

After being bootstrapped, the system displays its welcoming message and halts in XDELTA, as follows:

```
1 BRK AT nnnnnnnn  
address/NOP
```

XDELTA is waiting for input. (XDELTA never issues explicit prompts.) Usually, you proceed from this point with the following command:

```
;P RET
```

All of the XDELTA commands are described in Section 16.10 and in the *VMS Delta/XDelta Utility Manual*.

If the operating system halts with a fatal bugcheck, the system prints the bugcheck information on the console terminal. Then, because the BUGREBOOT system parameter is clear, 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.

16.3 Loading the Driver

Once the system is running, you can log in to the system as the system manager and load the experimental driver.

To load the driver, run SYSGEN and issue the appropriate LOAD and CONNECT commands. Example 16-1 provides a sample dialogue.

The first SHOW command in Example 16-1 causes SYSGEN to display the location of the device driver in system memory. You then define the device to the operating system. The second SHOW command causes SYSGEN to display the driver's location and the addresses of the device's DDB, CRB, IDB, and UCB.

Debugging a Device Driver

16.4 Inserting Breakpoints in Driver Source Code

Example 16-1 Loading a Driver

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> LOAD DMA0:[YOUR.DIRECTORY]YRDRIVER.EXE
SYSGEN> SHOW /DEVICE=YRDRIVER

__Driver__Start__End__Dev__DDB__CRB__IDB__Unit__UCB__
YRDRIVER  80060E50 80061070
SYSGEN> CONNECT YR /ADAP=3/VEC=%0274/CSR=%0776240
SYSGEN> SHOW /DEVICE=YRDRIVER

__Driver__Start__End__Dev__DDB__CRB__IDB__Unit__UCB__
YRDRIVER  80060E50 80061070
                YRA 8005FDC0 80060B70 8005FE00
                                O 80060BBO

SYSGEN> EXIT
```

16.4 Inserting Breakpoints in Driver Source Code

The SYSGEN command CONNECT calls controller initialization and unit initialization routines. To begin debugging the driver, you should ensure that the kernel-mode debugging utility XDELTA gains control of the driver before these routines execute. This is accomplished by placing one or more calls to the special system routine INI\$BRK within the source code of either the controller or unit initialization routine. To call INI\$BRK, use the following instruction:

```
JSB    G^INI$BRK
```

The INI\$BRK routine contains two instructions:

```
BPT
RSB
```

When the processor executes the BPT instruction, XDELTA gains control and reports the address of the breakpoint:

```
1 BRK AT nnnnnnnn
```

You can use INI\$BRK as a debugging tool and place calls to it within any part of the driver source code.

To determine the last driver PC before the breakpoint, examine the kernel stack. The stack register is register RE (hexadecimal format):

```
RE/address /address
```

Display RE to find the address of the top of the stack. Another display command (/) reveals the contents of the top of the stack, which should be the return address to the driver that called INI\$BRK.

Debugging a Device Driver

16.5 Calculating the Base of Driver Code

16.5 Calculating the Base of Driver Code

Before you debug the driver, it is a good idea to calculate the base address of driver code, as follows:

- 1 Run SYSGEN and issue the SHOW/DEVICE command. The resulting display lists the location in nonpaged pool at which SYSGEN loaded the driver.
- 2 Consult the loadmap for the driver (obtained at driver link time). Driver code 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 loadmap, consult the driver prologue table in the driver listing. Look for the address of DPT_STORE_END, which generates a 2-byte entry that terminates the DPT. To get the base address of driver code, add the address of DPT_STORE_END + 2 to the address at which the driver was loaded. You can set an XDELTA base register to the base of driver code; Section 16.8 describes this procedure.

16.6 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 16.4.

Note that, in a VMS multiprocessing system, only one processor can be in XDELTA at a time. If a processor encounters a breakpoint while another processor is in XDELTA, it too must wait until the current processor exits from XDELTA. When it does, this processor again executes the instruction that caused it to attempt to enter XDELTA. If the processor previously in XDELTA did not delete the breakpoint, this processor now enters XDELTA. If the processor previously in XDELTA did remove the breakpoint, this processor does not enter XDELTA.

You can also invoke XDELTA to set breakpoints interactively by requesting an XDELTA software interrupt.

The procedures described in Table 16-3 issue a software interrupt to a single processor at IPL 14.

On the processor requesting the XDELTA interrupt, the interrupt service routine at IPL 14 handles the interrupt by calling the routine INI\$BRK, which in turn executes the first XDELTA breakpoint. XDELTA then issues this message:

```
1 BRK AT nnnnnnnn
address/NOP
```

Debugging a Device Driver

16.6 Requesting an XDELTA Software Interrupt

In a VMS multiprocessing system, if another processor attempts to enter XDELTA at this time, it must wait until the processor currently in XDELTA exits.

Table 16-3 Requesting an XDELTA Software Interrupt

| System | Boot Commands |
|--|--|
| VAX 8530/8550/8700/8800/8830/8840 ¹ VAX 6200 Series ² | \$ <input type="checkbox"/> CTRL/P >>>HALT >>>D/I 14 E >>>C |
| VAX 8600/8650/8670 ² | \$ <input type="checkbox"/> CTRL/P >>>HALT >>>D/I 14 E >>>C |
| VAX 8200/8250/8300/8350 VAX-11/750 VAX-11/730 VAX-11/725 ² | \$ <input type="checkbox"/> CTRL/P >>>D/I 14 E >>>C |
| VAX-11/780 VAX-11/785 | \$ <input type="checkbox"/> CTRL/P >>>HALT >>>DEPOSIT/I 14 E >>>CONTINUE |
| MicroVAX 3600 Series MicroVAX II MicroVAX I ² | Press and release the HALT button on the CPU control panel, or press the BREAK key (if enabled) on the console terminal. Then issue these commands: >>>D/I 14 E >>>C |

¹Note that the console prompt for the VAX 8830 and 8840 systems is *PS-CIO-0>* and not *>>>*.

²These VAX systems accept only 1-character console commands.

16.7 Examining the Vector-Jump Table

To gain familiarity with the I/O database, you might wish to look for the address of the location in the channel request block that contains a JSB instruction to the driver's interrupt service routine. You can do this at a controller initialization breakpoint because one of the inputs is the IDB address. The procedures for locating the driver interrupt service routine on non-direct-vector and direct-vector adapters follow.

Debugging a Device Driver

16.7 Examining the Vector-Jump Table

Non-Direct-Vector Procedure

```
R5/IDB-address Q+10/ADP-address
Q+10/vector-table-address
Q+vector-address-in-hex/address-of-JSB-instruction-in-CRB
Q!JSB-instruction
```

Direct-Vector Procedure

```
R5/IDB-address Q+10/ADP-address
Q+10/vector-table-address
Q+vector-address-in-hex+2/address-of-JSB-instruction-in-CRB
Q!JSB-instruction
```

Finding the address of the driver's interrupt service routine 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 can dispatch to some other interrupt service routine, or dispatch to an unassigned vector.

See the SYSGEN device table shown in Table 15-2 for a list of vectors. Consult DIGITAL field service for help with any problem similar to the one described above.

16.8 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 16.5 to determine the base address of driver code, then set a relocation register by issuing the following command:

```
driver-base-address,0;X RET
```

This command sets relocation register X0 to the base of driver code. Now you can examine offsets into the code using X0 as a base:

```
X0 + offset/nnnnnnnn
```

or

```
X0 + offset!instruction
```

XDELTA also uses the base register to display address values in the base register plus offset format. Suppose, for example, that your driver contains the following code:

| | | | | | | | | | |
|----|----|----|----|------|-----|-------|--|------|-----------|
| | 50 | 81 | 90 | 00D3 | 132 | 10\$: | | MOVB | (R1)+,R0 |
| | | 10 | 13 | 00D6 | 133 | | | BEQL | 20\$ |
| | 20 | 50 | 91 | 00D8 | 134 | | | CMPB | R0,#^A/ / |
| | | F6 | 19 | 00DB | 135 | | | BLSS | 10\$ |
| 7A | 8F | 50 | 91 | 00DD | 136 | | | CMPB | R0,#^A/Z/ |
| | | F0 | 14 | 00E1 | 137 | | | BGTR | 10\$ |
| | 82 | 50 | 90 | 00E3 | 138 | | | MOVB | R0,(R2)+ |
| | | EB | 11 | 00E6 | 139 | | | BRB | 10\$ |

Debugging a Device Driver

16.8 Setting an XDELTA Base Register

If base register 0 contains the base address of your driver, the following XDELTA dialogue is possible:

```
X0+D3,X0+E6!X0+D3/MOVB (R1)+,R0
X0+D6/BEQL      X0+E8
X0+D8/CMPB      R0,#20
X0+DB/BLSS      X0+D3
X0+DD/CMPB      R0,#7A
X0+E1/BGTR      X0+D3
X0+E3/MOVB      R0,(R2)+
X0+E6/BRB       X0+D3
```

To set breakpoints in driver code, use the following command:

```
X0 + offset;B [RET]
```

To display a driver instruction and set a breakpoint, add the instruction's offset to the base register. For example:

```
X0+1C!instruction .;B [RET]
```

The last XDELTA command sets a breakpoint at the displayed location. See Section 16.10 or the *VMS Delta/XDelta Utility Manual* for a detailed discussion of XDELTA commands.

16.9 Examining the UCB, IRP, or PSL

In addition to using XDELTA to debug drivers, you can also examine the contents of the UCB and the associated IRP.

It is also useful to examine the contents of the PSL at the time of a system failure. The PSL, for example, indicates the IPL at the time. When the system fails it prints the PSL and other register contents on the console terminal.

While the system is running, the following command can be used to examine the PSL in XDELTA:

```
RF+4/
```

The PSL location is stored in the longword following the PC.

16.10 XDELTA Commands

Table 16-4 summarizes XDELTA commands. The sections that follow this table describe the commands.

Table 16-4 XDELTA Command Summary

| Command | Function |
|-------------------------|-------------------|
| Set Display Mode | |
| [B | Set byte mode |
| [W | Set word mode |
| [L | Set longword mode |

Debugging a Device Driver

16.10 XDELTA Commands

Table 16–4 (Cont.) XDELTA Command Summary

| Command | Function |
|---|--|
| Set Display Mode | |
| [I | Set instruction mode |
| " | Set ASCII mode |
| Set and Proceed from Breakpoint | |
| ;P | Proceed from breakpoint |
| ;B | Set/clear/display breakpoint |
| Open, Examine, and Close Location | |
| / | Open location (display contents in current mode) |
| ! | Open location (display contents as instructions) |
| RET | Close current location |
| LF | Close current location; open next |
| TAB | Open location specified by current value |
| ESC | Display previous location |
| Deposit in Location | |
| 'string' | Deposit string at current location, autoincrementing the current location symbol (.). Every carriage-return and line-feed character typed will be stored. An apostrophe terminates the string. |
| Step, Set Location, and Execute Code | |
| S | Execute one instruction, step into subroutine call |
| O | Execute one instruction, step over subroutine call (on CALLx, JSB, or BSBx instruction) |
| ;G | Go to location and proceed |
| ;E | Execute command string at location |
| Special Symbols | |
| , | Field separator |
| Q | Last quantity displayed |
| = | Display value of expression; set Q |
| Xn | Base register n |
| ;X | Set base register |
| Rn | Register n |
| Pn | Processor register n |

Debugging a Device Driver

16.10 XDELTA Commands

Table 16–4 (Cont.) XDELTA Command Summary

| Command | Function |
|------------------------|---|
| Special Symbols | |
| G | Add ^X80000000 to subsequent or preceding value |
| H | Add ^X7FFE0000 to subsequent or preceding value Current location |
| Operators | |
| + | Add |
| - | Subtract |
| space | Add |
| * | Multiply |
| @ | Shift |
| % | Divide |
| Miscellaneous | |
| :L | List names and locations of loaded executive images |

16.10.1 Values and Expressions

All numeric values are interpreted in hexadecimal radix. Expressions are strings of alternating values and binary operators, where the first and last items in the string are always values, as in the following example:

G4A32 + 24 - .

XDELTA evaluates expressions from left to right with no precedence, and ignores trailing operators. To display the value of an expression, use the XDELTA Show Value (=) command, as follows:

Syntax

expression=value-of-expression

Type an expression followed by an equal sign (=). The expression can be composed of a series of values and operators from the set of operators listed in the command summary. XDELTA shows the value of the expression according to the current display data type. The last quantity (Q) is set to the value of the computed expression.

Debugging a Device Driver

16.10 XDELTA Commands

16.10.2 Special Symbols

XDELTA defines the following special symbols:

| | |
|--------|--|
| . | Current location; set by slash (/), exclamation point (!) and TAB operations. |
| Q | Last quantity displayed; you can also change this value by using the Show Value (=) command described in Section 16.10.1. |
| X0–XF | Base registers; used for remembering values. Set base registers by means of the Set Base Register command (;X) described in Sections 16.8 and 16.10.2.3. XDELTA, by default, stores special values in base registers X4 and X5 that help reference the process control block of the current process (see Section 16.10.2.1). Also, XDELTA initializes XE and XF with special commands that help reference page-frame numbers, as described in Section 16.10.2.2. |
| R0–RF | General register names. |
| PO–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. |

16.10.2.1 Stored Base Registers

XDELTA defines two base registers useful in system debugging: X4 and X5. Base register X4 contains the address of the location that contains the address of the PCB of the current process on the current processor. Base register X5 corresponds to the global symbol SCH\$GL_PCBVEC, which contains the starting address of the list of PCB slots.

16.10.2.2 Stored Command Strings

XDELTA contains two predefined command strings whose addresses are contained in base registers XE and XF. You can use these commands during general system debugging as well as driver debugging; they perform the following functions:

| | |
|----|---|
| XE | Use the value of base register X0 as a page-frame number and display the PFN database for that page |
| XF | Set base register X0 to the value (PFN) in R0 and perform the same function as XE |

You must initialize the stored commands to set the relocation registers they use (X6 through XD). Issue the following commands:

```
XE;E RET
XF;E RET
```

After executing these commands, you can use the commands stored in XE and XF to obtain the following information about a page-frame number:

- Specified physical page number (PFN)
- PFN state
- PFN type
- PFN reference count

Debugging a Device Driver

16.10 XDELTA Commands

- 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

16.10.2.3 Setting Base Registers

Syntax

address-expression,n;X RET

Type an expression followed by a comma (,), a single digit between 0 and D (hexadecimal), a semicolon (;), and the letter X. XDELTA assigns the specified expression to the base register selected by *n*. XDELTA confirms that the base register is set by displaying the value deposited in the base register.

Whenever XDELTA displays an address located close to an address stored in a base register, XDELTA displays the base register identifier (Xn), followed by an offset that gives the address's location in relation to the address stored in the base register. For example, if base register 2 (X2) contains 800D046A and the address XDELTA needs to display is 800D052E, XDELTA displays X2+C4. XDELTA computes relative addresses for opened or displayed locations and addresses that are instruction operands.

XDELTA displays an address in base register plus offset format to a distance of 800_{16} from the base register. If the address falls outside this range, XDELTA displays it as a hexadecimal value.

16.10.3 Display Names and Locations of Loaded Executive Images

Syntax

;L

Use the ;L command to list the names and locations of the loaded modules of the VMS executive. If you issue the ;L command before all the executive images are loaded (for example, at an XDELTA initial breakpoint), only those images that have been loaded will be displayed.

16.10.4 Set Display Mode

Syntax

- [B Byte width
- [W Word width
- [L Longword width
- [I Instruction display (using longword width)
- " ASCII display (using current width)

Debugging a Device Driver

16.10 XDELTA Commands

Type a left square bracket ([) followed by one of the letters B, W, or L to change the current display width to byte, word, or longword respectively. The default value is longword. The setting remains in effect until another display mode control command is given. For example, the following command displays the least significant byte contained at the specified address and deposits the new value to that byte only.

```
address-expression [B/ old-value new-value
```

Type a left square bracket ([) followed by the letter I to change the current display mode to instruction format. This command is equivalent to the exclamation point (!) command and, similarly, is canceled by typing a slash (/) or a quotation mark ("). Instruction mode sets display mode storage units to longword values. For an example of an instruction display, see Section 16.8.

You can display contents of memory locations in ASCII characters by typing an address expression followed by a quotation mark (").

```
address-expression" old-value-in-ASCII
```

Pressing LINE FEED displays the next location in ASCII.

The display mode remains set to ASCII until the next slash (/) or exclamation point (!) command. At this point, the display mode reverts to hexadecimal. The width remains unchanged.

16.10.5 Open, Examine, and Close Location

XDELTA provides the commands described in the following sections to open, examine, and close the specified memory locations.

16.10.5.1 Open and Display Value Command Syntax

```
address-expression/old-value [new-value-expression]
```

Type an address expression followed by a slash (/) character. XDELTA displays the contents of the location (**old-value** above), followed by a space character. You can change the value at the location by typing a new value and then pressing RETURN. If you press RETURN without preceding it with a value, the old contents remain unchanged.

The display and the value deposited default to longword hexadecimal values. The length can be changed to byte or word with the set mode commands.

A slash preceded by a null address expression uses the displayed value (Q) as the address value. This feature is convenient for following address linked chains, as follows:

```
address-expression/old-value /old-value /old-value
```

Debugging a Device Driver

16.10 XDELTA Commands

16.10.5.2 Display Instruction Command Syntax

address-expression!decoded-instruction

Type an address expression followed by an exclamation point (!). XDELTA displays the contents of memory as a VAX MACRO instruction starting with the address you specify.

XDELTA does not make any distinction between reasonable and unreasonable instructions or instruction streams; the decoding always begins at the specified address. The display instruction command does not allow you to modify the displayed location. The command sets a flag that causes subsequent close and display next or indirect location commands to perform instruction decoding. You can reset the flag with the open and display value command.

Whenever an address appears as an instruction operand, XDELTA sets the last quantity displayed (Q) to that address. XDELTA changes Q only for operands that use program counter or branch displacement addressing modes; Q is not altered for literal and register addressing modes. This feature is useful for following branches, as follows:

address-expression!BRW address-2 !instruction-at-address-2

16.10.5.3 Close and Display Next Location Command Syntax

LF
address/old-value

Press LINE FEED. XDELTA closes the current open location, then opens and displays the value in the next location, according to the current display mode.

If instruction display is the current mode, XDELTA does not deposit a value in the open location. The next location is the first location after the instruction currently displayed. If value display is the current mode, you can deposit a value into the open location. In this case, the next location is the current location, incremented by the current data width (byte, word, or longword).

16.10.5.4 Display Range Command Syntax

start-addr-expression,end-addr-expression/contents-of-start

or

start-addr-expression,end-addr-expression!contents-of-start

Type two address expressions separated by a comma and followed by a slash (/) or exclamation point (!) character. XDELTA displays the range of addresses, using the specified display mode (value or instruction). If you specify instruction display, XDELTA decodes one or more instructions. Otherwise, XDELTA displays the contents of each location in the current data type (byte, word, or longword).

16.10.5.5 Indirect Command Syntax

`TAB`
address/old-value

Press TAB. XDELTA uses the last quantity displayed (Q) as an address and displays that address and its contents using the current display mode. This command opens locations in the same way as the slash (/) and exclamation point (!) commands, but prints the information on a new line and displays the address value before showing the address's contents.

16.10.5.6 Display Previous Location Command Syntax

`ESC`
address/old-value

Press ESC. Unless the current display mode is instruction, XDELTA decreases the location counter by the current data width, and displays the contents of the resulting location using the current data width and type. This command is ignored in instruction display mode.

16.10.6 Breakpoints

XDELTA uses the following commands to set and clear breakpoints, display a list of set breakpoints, continue from a breakpoint, and set a complex breakpoint.

16.10.6.1 Setting Breakpoints Syntax

address-expression;B `RET`

Type an address followed by a semicolon (;) and the letter B, then press RETURN. XDELTA sets a breakpoint at the specified location and assigns it the first available breakpoint number.

Alternate syntax:

address-expression,n;B `RET`

Type an address followed by a comma, a single digit between 2 and 8, a semicolon (;), the letter B, and then press RETURN. XDELTA sets a breakpoint at the specified location and assigns it the specified breakpoint number. Breakpoint 1 is reserved for INI\$BRK.

Before XDELTA executes the instruction as a breakpoint, it suspends normal instruction processing, sets a flag that causes subsequent close and display next or indirect location commands to perform instruction decoding, and displays the following message:

```
n BRK at address  
address/decoded-instruction
```

You can now enter XDELTA commands. You can reset the flag that controls instruction display mode by issuing the open and display value command.

Debugging a Device Driver

16.10 XDELTA Commands

16.10.6.2 Clearing Breakpoints

Syntax

0,n;B

Type zero (0) followed by a comma, a single digit between 2 and 8, a semicolon (;), the letter B, and then press RETURN. XDELTA clears the specified breakpoint. Never clear breakpoint 1.

16.10.6.3 Displaying Breakpoint List

Syntax

;B

Type a semicolon (;) followed by the letter B. XDELTA shows the current settings 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 16.10.6.5)
- Command string address (for complex breakpoints)

16.10.6.4 Proceeding from Breakpoints

Syntax

;P

Type a semicolon (;) followed by the letter P, and then press RETURN. XDELTA continues executing at the current PC.

16.10.6.5 Setting Complex Breakpoints

Syntax

address-expression,n,display-addr-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.

16.10.7 Step, Set Location, and Execute Instruction Commands

The following XDELTA commands enable you to step through and execute driver code.

16.10.7.1 Loading PC and Continuing

Syntax

address-expression;G

Type an address, a semicolon, and G, then press RETURN. XDELTA loads the address into PC and continues executing at the new PC.

16.10.7.2 Execute Instruction and Step Command

Syntax

S

Type an S. XDELTA causes one instruction to be executed, then displays the address of the next instruction and decodes that instruction.

This command also sets a flag that causes subsequent close and display next or indirect location commands to perform instruction decoding. The open and display value command resets the flag.

If the next instruction is BSBB, BSBW, JSB, CALLG, or CALLS, this command steps into the subroutine and displays the first instruction within the routine.

16.10.7.3 Step Instruction Over Subroutine Command

Syntax

O

Type an O. XDELTA causes one instruction to be executed, then displays the address of the next instruction and decodes that instruction.

This command also sets a flag that causes subsequent close and display next or indirect location commands to perform instruction decoding. The open and display value command resets the flag.

If the next instruction is BSBB, BSBW, JSB, CALLG or CALLS, XDELTA executes the entire subroutine and displays the instruction that immediately follows the subroutine call; this command steps over subroutines.

16.10.8 Execute String Command

Syntax

```
address-expression;E RET
```

Type an address expression followed by a semicolon, the letter E, then press RETURN. This command executes the ASCII commands found at the specified address expression. If you terminate the ASCII commands with a semicolon followed by the letter P, XDELTA will proceed with program execution. If you terminate the string with null (1 byte of 0), XDELTA waits for a new command.

To create command strings, open the address of the start of the string and deposit ASCII text as follows:

```
address/old-contents 'XDELTA-command' RET
```

You can use any XDELTA command, including RETURN, LINE FEED, and TAB.

To terminate the string with a null, follow the above command with

```
./old-contents 0 RET
```

You can deposit command strings into nonpaged system patch space. To determine the size of patch space and its starting address, locate the symbol PAT\$A_NONPGD in the system map file (SYS\$SYSTEM:SYS.MAP). This

Debugging a Device Driver

16.10 XDELTA Commands

symbol contains a descriptor of the address and size of patch space remaining in the system, as follows:

```
PAT$A_NONPGD::  
    .LONG    size-in-bytes  
    .LONG    patch-space-start-address
```

You can also preassemble command strings with your experimental driver. Locate the addresses of these strings as you would any other address within your driver.

16.11 Guidelines for Debugging Device Drivers

The following sections discuss errors commonly made during debugging sessions and describe additional debugging techniques.

16.11.1 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
```

16.11.2 Adjusting the Device Timeout Value

When single-stepping through driver code using XDELTA, it may be necessary to adjust the device's timeout value (as specified in the WFIKPCH or WFIRLCH macro) so that it is large enough to keep the device from timing out. When the driver debugging is complete, this value should be reset to a reasonable length of time.

16.11.3 XDELTA and System Failures

Driver errors can cause the operating system to suspend activity in such a way that you cannot invoke XDELTA. In this case, the only recourse is to induce a system failure. Follow the procedure described in the *VMS System Dump Analyzer Utility Manual*; the system will signal a fatal bugcheck.

To gain control in XDELTA following a fatal bugcheck, stop in SYSBOOT while initializing the system and clear the BUGREBOOT system parameter. The system will stop in XDELTA, thereby allowing you to examine the device UCB 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 system parameter set, allow the system to reboot, and then invoke the System Dump Analyzer (SDA) Utility to examine the condition of the I/O data structures at the time of the fatal bugcheck. The *VMS System Dump Analyzer Utility Manual* provides detailed information on fatal bugcheck stack format and how SDA can help debug a device driver.

16.12 Common Driver Errors

This section describes errors commonly made in drivers.

16.12.1 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
```

16.12.2 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 using incorrect length. On VAX systems that use direct-vector interrupts, these references cause a fatal machine check exception. On VAX systems using non-direct-vector interrupts, these references cause a UNIBUS adapter error interrupt. The system logs the adapter error and continues.

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.

16.12.3 Destroying Register Contents

Because the driver frequently calls VMS I/O routines, you must be careful to anticipate the register usage of these routines. Most VMS common I/O support routines use R0 through R3 freely. A frequent driver bug is to load a value into R3 and expect to find it intact after a call to allocate or load adapter resources.

Other VMS I/O routines write into R4. In some cases, the use of R4 is obvious; for example, IOC\$REQSCHANL writes the device's CRB address into R4. In other cases, you might not anticipate the use of R4.

For example, EXE\$IIOFORK saves the calling code's R4 in a fork block, and then writes the device's IPL into R4. Because the normal flow of events is that an interrupt service routine restores a driver with a JSB instruction and the driver then calls EXE\$IIOFORK which returns to the interrupt service routine, the instructions following the JSB in the interrupt service routine can only assume R5 is still untouched. The coding sequence is as follows:

```
MOVQ    UCB$L_FR3(R5),R3      ; Restore R3-R4.
JSB     @UCB$L_FPC(R5)       ; Restore the driver process.
```

```
.;
;Between these instructions, the interrupt service routine
;can make no assumptions about the contents of R0 through R4.
.;
```

```
POPR    #^M<R0,R1,R2,R3,R4,R5> ; Restore interrupt registers.
REI     ; Return from the interrupt.
```

Debugging a Device Driver

16.13 Pool Checking Mechanism

16.13 Pool Checking Mechanism

Certain system failures cannot easily be traced to a single instruction or to a single piece of kernel-mode code. If a device driver, for example, accesses memory that it has not properly allocated or continues to use memory that it has deallocated, a system failure can occur long after the driver has completed its activity. The system may crash when another operating system thread executes and attempts to use the corrupted data.

Special pool checking code in the VMS memory allocation and deallocation modules can help isolate problems of this sort reliably and quickly. In a normal VMS system, this code is disabled. For a system experiencing frequent and inexplicable failures, you can enable pool checking by setting the POOLCHECK system parameter.

When enabled, pool checking routines execute whenever pool is deallocated or allocated.¹

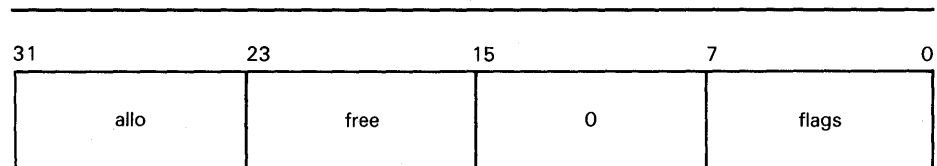
On any deallocation of pool, the routine fills the deallocated packet with a "free" pattern specified in the POOLCHECK system parameter. The first five longwords of the packet are not filled, but instead contain the following information:

- Forward and backward links into the free list
- Size, type, and subtype fields
- Address of the code that deallocated the packet
- Checksum

On any allocation from pool, the routine verifies the checksum and ensures that the packet still contains the "free" pattern. If the pattern is still intact, the routine replaces the "free" pattern with an "allo" pattern, also specified in the POOLCHECK system parameter. The two patterns allow allocated, uninitialized pool to be distinguished from nonallocated pool. If the "free" pattern is not intact, the pool checking routine induces a POOLCHECK bugcheck, assuming that some code has modified the packet while it was on the free list.

Figure 16-1 illustrates the format of the POOLCHECK system parameter.

Figure 16-1 Format of the POOLCHECK System Parameter



ZK-6618-HC

¹ The pool allocation routines (EXE\$ALLOCBUF, EXE\$ALLOCIRP, EXE\$ALONONPAGED, COM\$DEANONPAGED, EXE\$DEANONPAGED, and so on) are discussed in Appendix C. These routines are the only means recommended by DIGITAL for allocating pool.

Debugging a Device Driver

16.13 Pool Checking Mechanism

The **flags** byte indicates the actions that the pool checking code should take whenever pool is allocated or deallocated. It also indicates the type of pool to be subject to checking. The following bit masks are defined:

| Bit Mask (hex) | Action |
|----------------|---|
| 1 | At deallocation, fill variable pool packets with free pattern. |
| 2 | At allocation, check packets for free pattern and, if the pattern is intact, fill with allo pattern. If not, induce POOLCHECK bugcheck. |
| 4 | At deallocation, fill SRPs with free pattern. |
| 8 | At deallocation, fill IRPs with free pattern. |
| 10 | At deallocation, fill LRP's with free pattern. |
| 80 | At deallocation, fill P1-space addresses with free pattern. |

The **free** byte indicates the character to be inserted in a packet (except for its header) when it is deallocated to free pool.

The **allo** byte indicates the character to be inserted in a packet (if bit 2 of the **flags** byte is set) when it is allocated.

It is possible that, when first enabled on a system, the pool checking mechanism will discover specific violations of pool allocation and deallocation protocol. In investigating subsequent crashes using SDA, you should first check the value of the global longword EXE\$GL_POOLCHECK to determine whether pool checking has been enabled and, if so, which packets it has been enabled for and which patterns it is using.

One of the results of the pool checking mechanism is the occurrence of a fatal system bugcheck such as INVEXCEPTN, SSRVEXCEPT, or FATALEXCPTN whenever kernel-mode code attempts to use an address in free pool. When these exceptions signal an access violation and the **free** pattern appears as the violating address in the exception's signal array, the exception PC has been caught in the process of using deallocated pool.

The POOLCHECK bugcheck is explicitly generated by the pool checking mechanism whenever it determines that a pool packet has been corrupted while on the free queue. When a POOLCHECK bugcheck occurs, you can obtain information about the crash from the contents of general registers as well as from the pool packet itself. At the time of the crash, the following registers contain relevant information:

| Register | Contents |
|----------|---|
| R0 | Allocation (allo) pattern |
| R1 | Deallocation (free) pattern |
| R2 | Address of packet being allocated |
| R3 | Number of longwords remaining in packet to be checked |
| R4 | Address in packet where the pool checking code discovered corrupted pattern |

Debugging a Device Driver

16.13 Pool Checking Mechanism

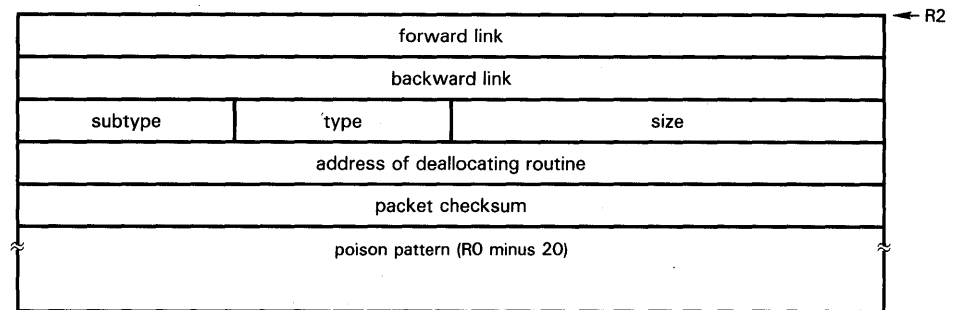
| Register | Contents |
|----------|--|
| R5 | Checksum, computed as the sum of the address of the packet, the deallocation pattern, the third and fourth longwords of the packet, and a longword within the system boot time quadword (EXE\$GQ_ BOOTIME) |

Because the address of the packet is in R2, you can attempt to format R2 to see what type of structure the pool is being allocated for. The following SDA commands accomplish this:

```
SDA> READ SYS$SYSTEM:SYSDEF.STB  
SDA> FORMAT @R2
```

If this does not identify the structure, you may obtain some information from the packet itself, as pictured in Figure 16-2.

Figure 16-2 Poisoned Pool Packet



ZK-6619-HC

To determine what code may have corrupted the packet, it may be helpful to examine the contents of R4 (the address at which the pool checking routine found a corrupted pattern). If this address contains an address, it is a fair assumption that code that uses that address placed it there.

In addition, the routine that deallocated the packet may be a likely suspect. Frequently, pool is corrupted by a device driver that deallocates pool and later attempts to use the pool that it has deallocated.

16.14 Detecting Driver Problems in a Multiprocessing System

When testing a new driver that has been designed to run in a VMS multiprocessing environment, it is a good idea to ensure that the system in which the driver is being tested is running the full-checking synchronization image. You can cause the full-checking synchronization image to be loaded at boot time on either a VAX uniprocessing system or multiprocessing system by the appropriate setting of the MULTIPROCESSING system parameter, as listed in Table 16-5.

Debugging a Device Driver

16.14 Detecting Driver Problems in a Multiprocessing System

Table 16–5 Settings of MULTIPROCESSING System Parameter

| Value | Result |
|-------|---|
| 0 | Loads uniprocessing synchronization image for any hardware configuration. |
| 1 | Loads full-checking synchronization image and sets multiprocessing-enabled bit (SMP\$V_ENABLED in SMP\$GL_FLAGS) if the hardware configuration is capable of multiprocessing and two or more processors are available; otherwise, loads uniprocessing synchronization image. This is the default value. |
| 2 | Loads full-checking synchronization image and sets multiprocessing-enabled bit regardless of the hardware configuration. |
| 3 | Loads streamlined synchronization image and sets multiprocessing-enabled bit if the hardware configuration is capable of multiprocessing and two or more processors are available; otherwise, loads uniprocessing synchronization image. |

In a processing environment with the full-checking synchronization image loaded, violation of spin lock synchronization by a device driver will produce the bugchecks described in Table 16–6.

Table 16–6 Bugchecks Produced by Full-Checking Multiprocessing

| | |
|------------|--|
| SPLIPLHIGH | <p>A processor has attempted to acquire a spin lock at an IPL higher than the IPL associated with spin lock synchronization (SPL\$_IPL). SMP\$ACQUIRE (called by the LOCK and FORKLOCK macros with condition=NOSETIPL not specified) signals this bugcheck.</p> <p>A processor has attempted to acquire a device lock—not already owned by the acquiring processor—at an IPL higher than the IPL associated with device lock synchronization (SPL\$_IPL). SMP\$ACQUIREL (called by the DEVICELOCK macro with condition=NOSETIPL <i>not</i> set) signals this bugcheck.</p> |
| SPLIPLLOW | <p>A processor has attempted to conditionally or unconditionally release a spin lock or device lock at an IPL lower than the IPL at which it originally acquired it. SMP\$RELEASE and SMP\$RESTORE (called by the UNLOCK and FORKUNLOCK macros) and SMP\$RELEASEL or SMP\$RESTOREL (called by the DEVICEUNLOCK macro) signal this bugcheck.</p> |
| SPLACQERR | <p>A processor has attempted to acquire a spin lock while holding a higher ranked spin lock. SMP\$ACQUIRE, SMP\$ACQUIREL, and SMP\$ACQNOIPL (called by the LOCK, FORKLOCK, and DEVICELOCK macros) signal this bugcheck.</p> |
| SPLRELERR | <p>An attempt has been made to completely release a spin lock not owned by the releasing processor. SMP\$RELEASE and SMP\$RELEASEL (called by the UNLOCK, FORKUNLOCK, and DEVICEUNLOCK macros) signal this bugcheck.</p> |

Debugging a Device Driver

16.14 Detecting Driver Problems in a Multiprocessing System

Table 16–6 (Cont.) Bugchecks Produced by Full-Checking Multiprocessing

| | |
|-----------|---|
| SPLRSTERR | An attempt has been made to conditionally release a spin lock not owned by the releasing processor. SMP\$RESTORE and SMP\$RESTOREL (called by the UNLOCK, FORKUNLOCK, and DEVICEUNLOCK macros when condition=RESTORE is specified) signal this bugcheck. |
|-----------|---|

An examination of the crash dump resulting from any of these bugchecks can help locate the cause of the crash. Enter the System Dump Analyzer (SDA) and perform the following steps:

- 1 Issue the following command:

```
SDA> READ/EXECUTIVE SYS$LOADABLE_IMAGES
```

This command generates the symbols that correspond to locations in the loadable images that are part of the VMS executive. These symbols facilitate the interpretation of addresses that appear in the stacks and other SDA displays.

- 2 Issue the following command:

```
SDA> SHOW STACK
```

Trace through the current stack to determine what activities on the processor led to the acquisition or release of the spin lock. Start at high stack addresses and work towards low addresses, identifying everything on the stack, or as much of it as required to decipher what is going on.

- 3 Issue the following command:

```
SDA> SHOW SPINLOCK/FULL/ADDR=@R0
```

This command produces a display of information about the spin lock the executing code was trying to acquire or release, a list of PCs indicating the addresses of the latest eight acquirers or releasers of the lock, plus the PC of the last unconditional release of a set of multiply nested spin lock acquisitions. Note the acquisition IPL and the rank of the spin lock.

- 4 To decipher SPLIPLLO and SPLIPLHI bugchecks, compare the IPL at which the system was running at the time of the crash to that shown in the SHOW SPINLOCK display as required for acquisition of the spin lock.

- 5 To decipher SPLACQERR, SPLRSTERR, and SPLRELERR bugchecks, issue the following command:

```
SDA> SHOW SPINLOCK/OWNED
```

In the case of SPLACQERR bugchecks, compare the rank of the spin lock being sought with that of the currently owned spin locks.

In the case of SPLRSTERR and SPLRELERR bugchecks, determine whether the releasing processing in fact did not own the spin lock it is attempting to release.

Standard drivers seldom release spin locks and fork locks. When they do, they should be careful to use the **condition=RESTORE** argument to the UNLOCK and FORKUNLOCK macros when it is likely that the driver code is executing at the behest of other code interested in retaining the lock.

Debugging a Device Driver

16.14 Detecting Driver Problems in a Multiprocessing System

One error of which driver writers should be wary concerns the unconditional release of a spin lock or fork lock for which there exist multiple, nested acquisitions. When multiple acquisitions of a spin lock accumulate for a processor, and one of the intermediate acquirers performs an explicit release of that lock, all ownership of the lock by that processor is entirely and immediately relinquished. If at least one of the original acquisition threads still expects the lock to be held when that thread regains control, system synchronization is broken. Moreover, when the original thread that acquired the lock itself attempts to release the lock, the system crashes with an SPLRELERR because the processor no longer owns the lock being released.

If few attempts have subsequently been made by the processor to obtain or release spin locks, you should be able to find the PC of the code that last unconditionally released the spin lock in question in SHOW SPINLOCKS /FULL/ADDRESS=@R0 display. Issue the SDA command EXAMINE/INST for each of the PCs in the display, from the top to the bottom, to determine the recent history of the lock.

17 Terminal Class and Port Drivers

This chapter describes details of the implementation of the VMS terminal driver. The VMS terminal driver consists of two pieces: the terminal port driver and the terminal class driver. These two pieces of code, when bound together within the unit control block (UCB), form a single device-dependent driver that implements the VMS terminal services.

TTDRIVER.EXE, the VMS terminal class driver, handles the device-independent functions and tasks. For example, it contains code that enables command line editing on many different types of terminals. The port drivers manage those functions and tasks that depend on the device's hardware configuration. For example, the port driver for a particular type of terminal controller performs the actual transmission and reception of characters to and from that terminal controller. The port driver reserves all manipulation and interpretation of those characters to the class driver. Because class driver code supports the functions common to terminal devices, a terminal port driver can contain only that code needed to control a specific interface.

There are several reasons why a new port driver may be required:

- To support a new terminal controller
- To implement a terminal server such as LAT11
- To provide a pseudoterminal

Both class drivers and port drivers adhere to the same rules as other VMS device drivers. They consist of the same routines and tables as standard drivers, and reference the same data structures. However, because a class driver and its port drivers must intercommunicate, they must employ a few additional structures and routines not required for standard VMS device drivers.

The structure of the VMS terminal driver illustrates one specific approach to the class/port concept. The discussions that appear in this chapter relate only to TTDRIVER.EXE and its ports. Note that there are no supported methods for implementing the class/port design in a non-DIGITAL-supplied device driver. Moreover, the System Generation Utility (SYSGEN) provides no support for alternate class drivers and can only connect port drivers to TTDRIVER.EXE.

The remainder of this chapter describes how the VMS terminal class and port drivers are structured and how they interact. A full description of the functions of the VMS terminal driver appears in the *VMS I/O User's Reference Manual: Part I*.

Terminal Class and Port Drivers

17.1 Overview

17.1 Overview

The terminal class driver is the device-independent part of the VMS terminal driver. It contains the driver's function decision table (FDT) routines, start I/O routine, fork process routines, code that implements the features of the VMS terminal services, and the class driver service routines.

The terminal port driver is the device-dependent piece of the VMS terminal driver. It contains the driver prologue table (DPT); data structure initialization; device, unit, and controller initialization routines; port service routines; interrupt service routine; and any additional device-dependent code. Among the port drivers included in VMS are DZDRIVER for the DZ-32 and DZ-11, YCDRIVER for the DMF-32 and DMZ-32, and YFDRIVER for the DHU-11 and DHV-11, YIDRIVER for the DMB32, and YEDRIVER for the MicroVAX 2000. (See the *VMS I/O User's Reference Manual: Part I* for a complete list of supported terminal controllers.)

17.2 Data Structures

There are three major data structures that define the communication between the port and the class drivers. These data structures are the UCB, the port driver vector table and the class driver vector table. To reference these structures, a driver must include an invocation of the \$TTYDEFS macro (from SYS\$LIBRARY:LIB.MLB). The \$TTYDEFS macro defines symbolic offsets for the following structures:

- Unit control block (UCB)
- Terminal UCB extension
- Channel request block (CRB)
- Interrupt dispatch block (IDB)
- Port and class driver vector tables
- Read buffer
- Input stack
- Item list descriptor
- Type-ahead buffer

17.2.1 Terminal UCB

A terminal UCB, as depicted in Figure 17-1, contains four sections: the system section (base UCB), the class driver required section, the port driver required section, and the port extension region.

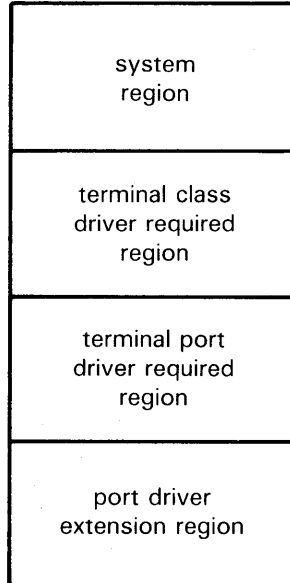
The system section of the terminal driver UCB contains the pieces of the UCB that are present in all of the UCBs on the system.

The class driver required section of the UCB contains fields that are needed by the class driver. These fields have names of the form `UCB$x_TT_fieldname`, where *x* denotes the field size and *fieldname* is the name of the field.

Terminal Class and Port Drivers

17.2 Data Structures

Figure 17–1 UCB Structure for Terminal Class/Port Drivers



ZK-6540-HC

The port driver required section of the UCB contains fields that both the class and port driver must access. These fields have names of the form `UCB$x_TP_fieldname`, where *x* denotes the field size and *fieldname* is the name of the field. Although a port driver may not actually use all these fields, their presence is required.

The terminal port extension region is defined by the terminal port driver. It can be any length and contain any context that the port driver needs to perform its duties.

Tables A–16 and A–20 describe the fields defined within a terminal UCB, and Figures A–17 and A–21 illustrate their configuration.

Terminal Class and Port Drivers

17.2 Data Structures

17.2.2 Port Driver Vector Table

The port driver vector table, as depicted in Figure 17-2, is the data structure that allows the terminal class driver to find the port service routines. The vector table contains the address, relative to the beginning of the port driver, of each port service routine. The port driver's controller initialization routine invokes the CLASS_CTRL_INIT macro, as described in Section 17.4.1.1, to relocate this vector table.

The port driver vector table is contained within the port driver itself, usually after the port driver's DPT. The port driver builds its vector table using the \$VECINI, \$VEC, and \$VECEND macros, as described in Section 17.2.4. A field in the UCB, UCB\$L_TT_PORT, contains the address of the port driver vector table.

Port and class drivers refer to fields within the port driver vector table using the symbolic offsets represented in Figure 17-2. To use these offsets, they include an invocation of the macro \$TTYDEFS (in SYS\$LIBRARY:LIB.MLB).

Figure 17-2 Port Driver Vector Table

| | |
|-----------------|----|
| PORT_STARTIO | 00 |
| PORT_DISCONNECT | 04 |
| PORT_SET_LINE | 08 |
| PORT_DS_SET | 12 |
| PORT_XON | 16 |
| PORT_XOFF | 20 |
| PORT_STOP | 24 |
| PORT_STOP2 | 28 |
| PORT_ABORT | 32 |
| PORT_RESUME | 36 |
| PORT_SET_MODEM | 40 |
| PORT_DMA | 44 |
| PORT_MAINT | 48 |
| PORT_FORKRET | 52 |
| PORT_FDT | 56 |
| PORT_CANCEL | 72 |

ZK-6625-HC

17.2.3 Class Driver Vector Table

The class driver vector table, as depicted in Figure 17-3, contains the address, relative to the beginning of the class driver, of each class service routine. The list is terminated by a longword containing zeros that indicates to the relocation routine where the list ends.

At driver load time, the relative offsets are relocated to actual virtual addresses. The port driver's controller initialization routine invokes the CLASS_CTRL_INIT macro, as described in Section 17.4.1.1, to relocate this vector table. The VMS terminal class driver is loaded by SYSINIT at boot time to allow the console terminal port driver to run.

Figure 17-3 Class Driver Vector Table

| | |
|------------------|----|
| CLASS_GETNXT | 00 |
| CLASS_PUTNXT | 04 |
| CLASS_SETUP_UCB | 08 |
| CLASS_DS_TRAN | 12 |
| CLASS_DDT | 16 |
| CLASS_READERROR | 20 |
| CLASS_DISCONNECT | 24 |
| CLASS_FORK | 28 |
| CLASS_POWERFAIL | 32 |
| CLASS_TABLES | 36 |

ZK-6624-HC

The class driver vector table is contained within the class driver itself, usually after the class driver's DPT. The class driver builds its vector table using the \$VECINI, \$VEC, and \$VECEND macros, as described in Section 17.2.4. A field in the UCB, UCB\$L_TT_CLASS, contains the address of the class driver vector table.

17.2.4 Vector Table Generation Macros

Port drivers use three VMS-supplied macros to build the port driver vector table: \$VECINI, \$VEC, and \$VECEND. Class drivers build the class driver vector table using the same macros. To obtain the definitions for these macros, a driver must invoke the \$TTYMACS macro (in SYS\$LIBRARY:LIB.MLB). This section briefly discusses the functions of each of these macros. An example of their use appears in Figure 17-4 and a full discussion of their syntax appears in Appendix B.

Terminal Class and Port Drivers

17.2 Data Structures

17.2.4.1 \$VECINI Macro

The \$VECINI macro creates a vector table and initializes each entry with the address of the driver's null entry point. Subsequent calls to the \$VEC macro fill in selected table entries with the addresses of real entry points.

The driver must specify the **drivername** and **null_routine** arguments to the \$VECINI macro. The **drivername** argument generally contains a 2-letter prefix to the driver name, such as DZ or YE. The **null_routine** argument contains the address of a routine within the driver (for example DZ\$NULL) that contains an RSB. When the class driver attempts to call the port driver at an entry point corresponding to an unsupported function, the port driver's null routine simply returns control to the class driver. The class driver can then proceed to service the error.

17.2.4.2 \$VEC Macro

The \$VEC macro validates and generates a vector table entry.

Each invocation of the \$VEC macro specifies the **entry** argument and the **routine** argument. However, a driver need not supply the address of a routine for each **entry** in the table. The \$VEC macro will construct a valid table regardless of how many entries are supplied. The \$VEC macro accepts the entry names (minus the PORT_ or CLASS_ prefix) shown in Table 17-1, for port drivers, and Table 17-2, for class drivers. Note that a driver accesses the table using the symbolic offsets indicated in Figures 17-2 and 17-3. The \$VECINI macro defines the prefix applied to the entries, which is PORT_ for the port vector table and CLASS_ for the class vector table.

17.2.4.3 \$VECEND Macro

The \$VECEND macro generates the longword of zeros that terminates the vector table and positions the location counter at label **drivername\$VECEND**. It has no required arguments.

17.3 Structure of Port and Class Drivers

Class and port drivers share a similar organization, as can be seen in Figures 17-4 and 17-5.

The vector table of each follows the driver prologue table (DPT). The driver specifies the address of the vector table in the **vector** argument to the DPTAB macro, which places its relative address in DPT\$W_VECTOR.

Following the vector table, and linked to the vectors by invocations of the \$VEC macro, are a set of service routines. The balance of the driver includes standard driver routines and tables and driver-specific routines.

Terminal Class and Port Drivers

17.3 Structure of Port and Class Drivers

Figure 17-4 Port Driver Structure

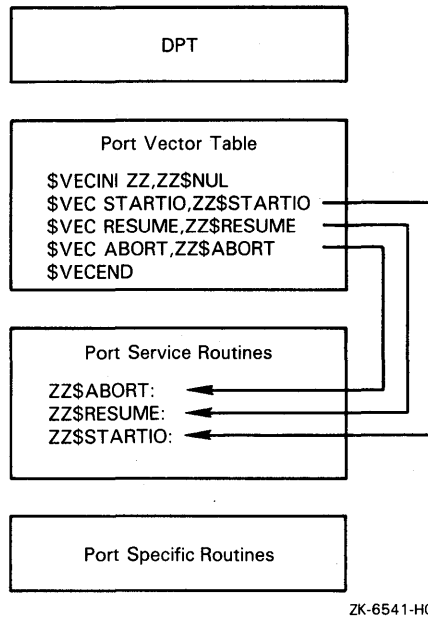
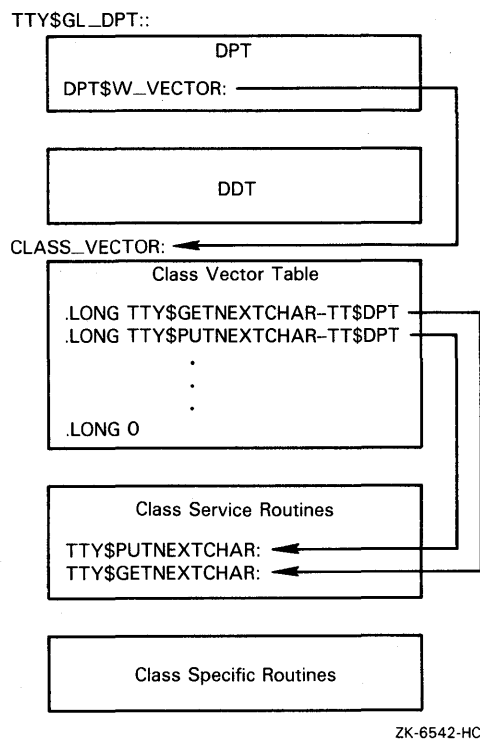


Figure 17-5 Class Driver Structure



Terminal Class and Port Drivers

17.3 Structure of Port and Class Drivers

17.3.1 Binding Class and Port Drivers

The terminal class and port drivers are bound together to form a single, complete driver in the manner represented in Figure 17-6.

The port driver's unit initialization routine performs the binding process by calling the `CLASS_UNIT_INIT` macro. The `CLASS_UNIT_INIT` macro fills in the following UCB fields as indicated:

| Field | Contents |
|------------------------------|--|
| <code>UCB\$_TT_CLASS</code> | Terminal class driver's vector table address |
| <code>UCB\$_TT_PORT</code> | Terminal port driver vector table address |
| <code>UCB\$_TT_GETNXT</code> | Address of the class driver's get-next-character routine (<code>CLASS_GETNXT</code>) |
| <code>UCB\$_TT_PUTNXT</code> | Address of the class driver's put-next-character routine (<code>CLASS_PUTNXT</code>) |
| <code>UCB\$_DDT</code> | Address of the terminal class driver's driver dispatch table |

Note that, because the get-next-character and put-next-character routines are the most heavily used class driver routines, their addresses are stored in the UCB. It is therefore possible for code to issue the instruction `JSB @address(R5)` to call either routine (presuming that R5 contains the address of the UCB). To call other routines, the driver must first move the address of the vector table to a general register and issue an instruction of the form `JSB @offset(Rn)`. Although a saving of one instruction does not seem significant, it can save one instruction per character when a driver is receiving data.

When the port driver's unit initialization routine completes the binding, the terminal class and port drivers have become one complete driver, and the device units are ready for I/O.

17.4 Port Driver Routines

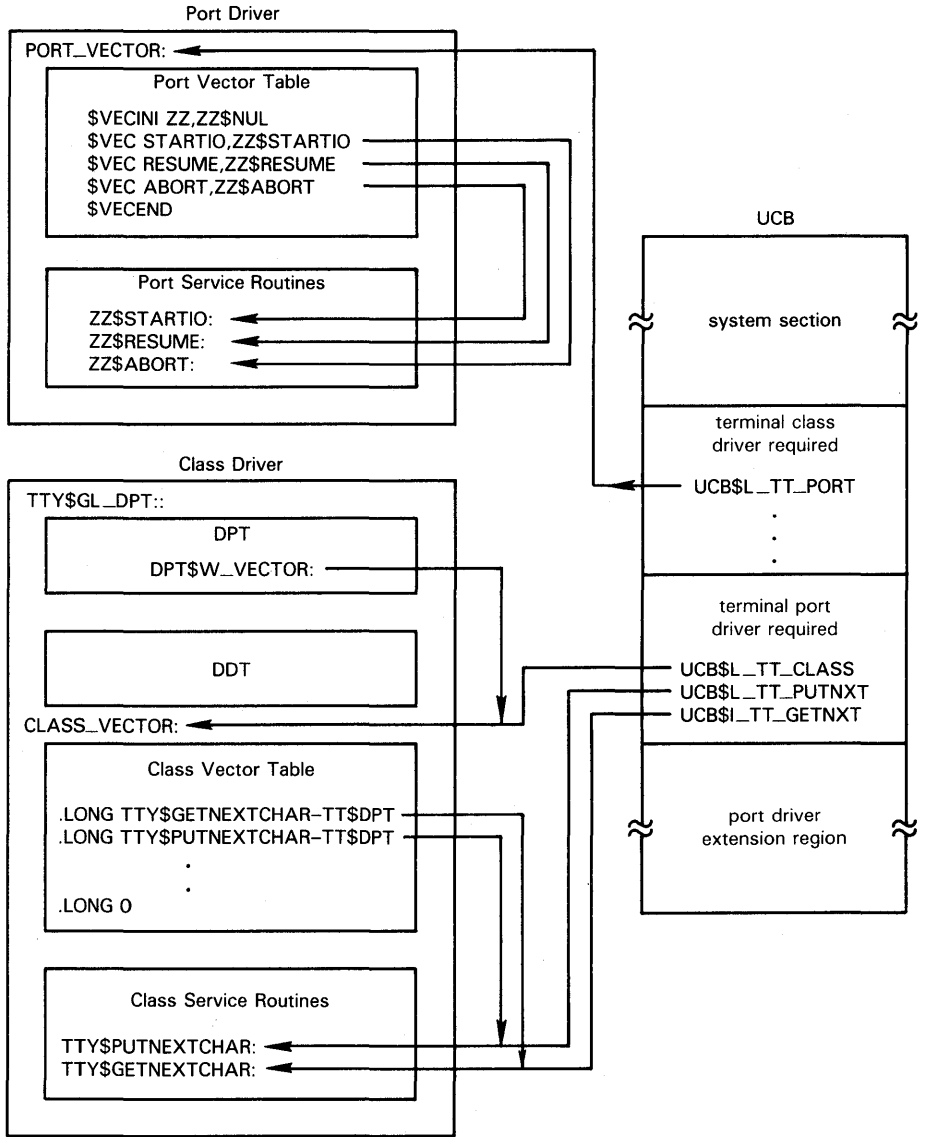
When the terminal class driver has completed a segment of device-independent processing of an I/O request, it calls a port routine to complete the device-dependent processing. The port driver contains three types of routines: port startup routines, port initiate routines, and port service routines.

Table 17-1 lists the port driver routines that are part of the class/port interface. This section describes the functions and context of each listed routine.

Terminal Class and Port Drivers

17.4 Port Driver Routines

Figure 17-6 Terminal Class/Port Driver Binding



ZK-6543-HC

Terminal Class and Port Drivers

17.4 Port Driver Routines

Table 17–1 Port Driver Routines

| Routine | Function |
|-----------------------------------|---|
| Port Startup Routines | |
| Controller initialization routine | Resets the controller and relocates the port and class driver vector tables |
| Unit initialization routine | Sets up each device unit controlled by the driver |
| Port Initiate Routines | |
| PORT_DISCONNECT | Notifies the port driver of the last deassign for the UCB |
| PORT_DS_SET | Outputs modem signals to a specified unit |
| PORT_FDT | Performs FDT processing for device-specific function modifiers |
| PORT_FORKRET | Return address in the port driver to which CLASS_FORK transfers control when servicing the port driver's request for a fork process |
| PORT_MAINT | Services \$QIO requests for IO\$_SETMODE function with the IO\$_M_MAINT modifier |
| PORT_SET_LINE | Changes terminal line parameters |
| PORT_SET_MODEM | Informs the port that a line has been enabled for modem signal input transitions |
| PORT_STARTIO | Starts output on an inactive line |
| Port Service Routines | |
| PORT_ABORT | Aborts any currently active output |
| PORT_CANCEL | Cancels internally queued operations in response to a \$CANCEL request |
| PORT_DMA | Not used; reserved to DIGITAL |
| PORT_RESUME | Resumes any previously stopped output |
| PORT_STOP | Halts the output data stream |
| PORT_STOP2 | Not used; reserved to DIGITAL |
| PORT_XOFF | Takes steps to halt an input data stream that is approaching its limit |
| PORT_XON | Resumes the acceptance of input data |

17.4.1 Port Startup Routines

Port startup routines include the port driver's controller and unit initialization routines. Note that, although these routines are not included in the port vector table, they must make calls to several class routines. They additionally fill the role of the equivalent initialization routines in a standard device driver, as discussed in Section 11.1.

Terminal Class and Port Drivers

17.4 Port Driver Routines

17.4.1.1 Controller Initialization Routine

The controller initialization routine is responsible for resetting the controller and relocating the port and class driver's vector tables. To perform the last-mentioned task, the routine should invoke the `CLASS_CTRL_INIT` macro, supplying the symbolic name of the driver prologue table (for instance, `DZ$DPT`) in the `dpt` argument and the address of the port driver's vector table in the `vector` argument. To use the `CLASS_CTRL_INIT` macro, the driver must include an invocation of the `$TTYMACS` definition macro (from `SYS$LIBRARY:LIB.MLB`).

17.4.1.2 Unit Initialization Routine

The unit initialization routine is responsible for setting up each individual device unit. The activities of a standard unit initialization routine include loading certain locations in the UCB with controller-specific data, preparing the hardware for input and output, and taking any action necessary to service a power failure.

The unit initialization routine of a terminal port driver must additionally perform the following tasks:

- 1 Invoke the `CLASS_UNIT_INIT` macro to generate the common code that must be executed by all terminal port driver unit initialization routines. This code includes the logic that binds the class and port drivers in the manner discussed in Section 17.3.1. Before it invokes the `CLASS_UNIT_INIT` macro, the unit initialization routine must place the address of the port driver vector table in `R0`.

To use the `CLASS_UNIT_INIT` macro, the driver must include an invocation of the `$TTYMACS` definition macro (from `SYS$LIBRARY:LIB.MLB`).

- 2 Call the class service routine, `CLASS_SETUP_UCB`, to allow the class driver to reset fields in the UCB.
- 3 Call the class service routine, `CLASS_SET_LINE`, to allow the class driver to reset the speed, parity, and the device-dependent bits, if necessary.
- 4 If the line can run modem protocol, call the class service routine, `CLASS_DS_TRANS`, with the transition type `MODEM$C_INIT` in `R1`. Note that, to use modem symbols, the port driver must invoke the `$TTYMDMDEF` macro (from `SYS$LIBRARY:LIB.MLB`).
- 5 When a power failure occurs (`UCB$V_POWER` set in `UCB$W_STS`), call the class service routine, `CLASS_POWERFAIL`.
- 6 Perform other hardware-specific functions.

17.4.2 Port Initiate Routines

The terminal class driver calls port initiate routines when it must initiate device activity and the port driver is not active. Port initiate routines can issue callbacks to the class driver.

A call to a port initiate routine uses the following instruction format:

```
MOVL   UCB$L_TT_PORT(R5),R0   ;get pointer to port vector table
JSB    @PORT_DISCONNECT(R0)   ;call port disconnect routine
```

Note that a port initiate routine must preserve the contents of all registers.

Terminal Class and Port Drivers

17.4 Port Driver Routines

17.4.2.1 PORT_DISCONNECT

A call to the PORT_DISCONNECT routine indicates that there are no longer channels associated with the device, thus notifying the port driver of the last deassignment for the device's UCB. If the delete bit (UCB\$V_DELMBX) is set in UCB\$W_DEVSTS, VMS will delete the UCB.

Note: As long as the device name is known to the system, broadcasts and assign channel requests may occur on this device. (Broadcasts, however, will not occur if the DEV\$V_NET bit is set in UCB\$L_DEVCHAR.)

Input to the PORT_DISCONNECT routine is as follows:

- R0 Flags. If bit 0 is set, the user requested that the UCB not be deleted (NOHANGUP).
- R5 Address of UCB.

17.4.2.2 PORT_DS_SET

The PORT_DS_SET routine sends modem signals to the specified unit. Masks representing modem signals are defined in \$TTDEF (in SYS\$LIBRARY:STARLET.MLB). They include the following:

| | |
|------------------|--|
| TT\$M_DS_CARRIER | Data channel received line signal detector |
| TT\$M_DS_CTS | Clear to send |
| TT\$M_DS_DSR | Data set ready |
| TT\$M_DS_DTR | Data terminal ready |
| TT\$M_DS_RING | Calling indicator |
| TT\$M_DS_RTS | Request to send |
| TT\$M_DS_SECRC | Secondary receive |
| TT\$M_DS_SECTX | Secondary transmit |

See the *VMS I/O User's Reference Manual: Part I* for an explanation of modem protocol.

Input to the PORT_DS_SET routine is as follows:

- R2 Low byte indicates signals to be activated; high byte indicates signals to be deactivated.
- R5 Address of UCB.

17.4.2.3 PORT_FDT

The terminal class driver calls the PORT_FDT routine when servicing a \$QIO request for an IO\$_TTY_PORT function. The PORT_FDT routine performs whatever tasks the class driver's FDT routine would normally do to service the request. These tasks include checking the function-dependent parameters (**p1** through **p6**), verifying access to buffers, and terminating with a call to EXE\$QIORETURN, EXE\$ABORTIO, or EXE\$FINISHIO.

The PORT_FDT routine thus allows a port driver to implement support for device-specific function modifiers without requiring an extension to the class/port interface.

If there is no PORT_FDT routine, control will pass to the port driver's null routine, which returns control to the class driver. The VMS terminal class driver, TTDRIVER.EXE, thereupon issues an illegal I/O function error.

Terminal Class and Port Drivers

17.4 Port Driver Routines

Input to the PORT_FDT routine is as follows:

| | |
|--------|---|
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| R6 | Address of CCB |
| R7 | Bit number of the I/O function code |
| 00(AP) | Address of the first function-dependent QIO parameter (p1) |

The routine destroys the contents of R2.

Note that a port driver must set TTY\$V_PC_PORTFDT in UCB\$W_TT_PRTCTL if it contains a PORT_FDT routine.

17.4.2.4 PORT_FORKRET

The terminal class driver's service routine, CLASS_FORK, returns control to the port driver's PORT_FORKRET entry point after servicing the port driver's request to create a fork process. (See the description of CLASS_FORK in Section 17.5.4.) The only context returned from the servicing of the fork request is the address of the UCB in R5.

The terminal class driver issues a JMP instruction (rather than a JSB instruction) to the PORT_FORKRET routine.

17.4.2.5 PORT_MAINT

The class driver calls the PORT_MAINT routine whenever a \$QIO request is issued for an IO\$_SETMODE function with the IO\$_M_MAINT modifier. The *VMS I/O User's Reference Manual: Part I* lists all possible maintenance functions; each port driver must decide which of these functions it must support.

Input to the PORT_MAINT routine is as follows:

| | |
|-----------------|---|
| R5 | Address of UCB |
| UCB\$B_TT_MAINT | Parameters to the IO\$_M_MAINT function |

17.4.2.6 PORT_SET_LINE

The PORT_SET_LINE routine changes terminal line parameters. The terminal class driver calls the PORT_SET_LINE routine whenever any terminal characteristic in UCB\$L_DEVDEPEND or UCB\$L_DEVDEPN2 is changed. It also calls this routine when speed, parity, and the enabling or disabling of DMA and automatic flow control are affected.

The PORT_SET_LINE routine is the only port routine that is allowed to write the fields UCB\$L_DEVDEPEND and UCB\$L_DEVDEPN2.

Terminal Class and Port Drivers

17.4 Port Driver Routines

Input to the `PORT_SET_LINE` routine is as follows:

| | |
|------------------|---|
| R5 | Address of UCB. |
| UCB\$B_TT_MAINT | Parameters to the <code>IO\$M_MAINT</code> function. |
| UCB\$B_TT_PARITY | Parity, stop bits, and frame size. |
| UCB\$W_TT_SPEED | Low byte indicates transmit speed; high byte indicates receive speed or is zero. |
| UCB\$W_TT_PRTCTL | DMA enable flag (<code>TTY\$V_PC_DMAENA</code>) and auto XOFF enable flag (<code>TTY\$V_PC_XOFENA</code>) |
| UCB\$L_DEVDEPEND | First longword for device-dependent status. |
| UCB\$L_DEVDEPN2 | Second longword for device-dependent status. |

The `PORT_SET_LINE` routine destroys the contents of R4.

17.4.2.7 **PORT_SET_MODEM**

A call to the `PORT_SET_MODEM` routine informs the port that the line has been enabled for modem signal input transitions. A port implementing modem functions must ensure that the hardware is ready to detect changes in input modem signals. When hardware does not provide this capability (as, for instance, the DZ11 terminal controller does not), the VMS terminal class/port interface implements the equivalent capability by using timer-based polling.

At the time of the call, R5 contains the address of the UCB.

17.4.2.8 **PORT_STARTIO**

The terminal class driver calls the `PORT_STARTIO` routine to start output on a line that is currently inactive. The `PORT_STARTIO` routine is always called with either a character or a burst of data and it is never called unless the line is idle (`UCB$V_INT` is clear in `UCB$W_STS`).

The `UCB$V_INT` bit functions as an interlock, signifying that the port output logic is busy. The class driver always sets `UCB$V_INT` when it calls `PORT_STARTIO`. If the port requests that timers be set up (`TTY$V_PC_NOTIME` clear in `UCB$W_TT_PRTCTL`), then the class driver calculates and creates an output timer for the burst or character and sets `UCB$V_TIM` in `UCB$L_STS`.

Input to the `PORT_STARTIO` routine is as follows:

| | |
|-------------------|---|
| R0 | First I/O status longword (<code>IRP\$L_IOST1</code>) |
| R3 | Character to be output (if <code>UCB\$B_TT_OUTTYPE</code> is 1) |
| R5 | Address of UCB |
| UCB\$B_TT_OUTTYPE | Zero if there is no character to be output; 1 if there is one character to be output; and a negative value if there is a burst to be output |
| UCB\$L_TT_OUTADR | Address of burst to output (if <code>UCB\$B_TT_OUTTYPE</code> is negative) |
| UCB\$W_TT_OUTLEN | Length of burst (if <code>UCB\$B_TT_OUTTYPE</code> is negative) |

Terminal Class and Port Drivers

17.4 Port Driver Routines

17.4.3 Port Service Routines

The terminal class driver can call port service routines at any time.

Note: Because they must consist of reentrant code, port service routines cannot issue callbacks to the class driver.

A call to a port service routine uses the following instruction format:

```
MOVL   UCB$L_TT_PORT(R5),R0   ;get pointer to port vector table
JSB    @PORT_ABORT(R0)        ;call port abort routine
```

Note that a port service routine must preserve the contents of all registers.

17.4.3.1 PORT_ABORT

The terminal class driver calls the PORT_ABORT routine to abort any currently active output activity: for instance, the last burst of output sent to the port. The PORT_ABORT routine invalidates the contents of the address stored in UCB\$L_TT_OUTADR.

At the time of the call, R5 contains the address of the UCB.

17.4.3.2 PORT_CANCEL

The terminal class driver calls the PORT_CANCEL routine in servicing a \$CANCEL, \$DASSGN, or \$DALLOC request. The PORT_CANCEL routine cancels any internally queued operations for the port. Most commonly, a call is issued to this routine when a request to establish an outgoing connection has been stalled because the port is busy.

Input to the PORT_CANCEL routine is as follows:

| | |
|----|--|
| R2 | Channel index number |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| R8 | Reason for cancellation, one of the following: |
| | CAN\$C_CANCEL Called by \$CANCEL system service |
| | CAN\$C_DASSGN Called by \$DASSGN or \$DALLOC system service |

17.4.3.3 PORT_RESUME

The terminal class driver calls the PORT_RESUME routine to resume any previously stopped output. The port must be prepared for this routine to be called at any time (whether output is currently active or has previously been stopped). The PORT_RESUME routine should always ensure that the port hardware is enabled for output.

At the time of the call, R5 contains the address of the UCB.

17.4.3.4 PORT_STOP

The PORT_STOP routine halts the output data stream. The terminal class driver normally calls this routine in response to input flow control. When called, the PORT_STOP routine should stop the data stream as soon as possible.

At the time of the call, R5 contains the address of the UCB.

Terminal Class and Port Drivers

17.4 Port Driver Routines

17.4.3.5 PORT_XOFF

The terminal class driver calls the PORT_XOFF routine when it is approaching or has reached its input limit. The PORT_XOFF routine takes steps to stop the input data stream. For character-oriented controllers, it commands the port to insert the flow control character in the output data stream as soon as possible.

Input to the PORT_XOFF routine is as follows:

| | |
|------------|--|
| R3 | Flow control character to be inserted in the input data stream |
| R5 | Address of UCB |
| UCB\$L_STS | UCB\$V_INT may or may not be set |

The PORT_XOFF routine must set UCB\$V_INT in UCB\$L_STS.

17.4.3.6 PORT_XON

The terminal class driver calls the PORT_XON routine when it has cleared its input path and is ready to accept data. For character-oriented controllers, the PORT_XON routine commands the port to insert the flow control character in the input data stream.

Input to the PORT_XON routine is as follows:

| | |
|------------|--|
| R3 | Flow control character to be inserted in the input data stream |
| R5 | Address of UCB |
| UCB\$L_STS | UCB\$V_INT may or may not be set |

The PORT_XON routine must set UCB\$V_INT in UCB\$L_STS.

17.4.3.7 Port Interrupt Service Routines

A terminal port driver must contain code to service receiver interrupts and transmitter interrupts. The exact form of a device interrupt associated with the port driver is device dependent. For multiple-line interfaces, the port driver must also determine which line is requesting the interrupt and move its UCB address into R5.

To service *receiver interrupts*, the port driver obtains a character from the port, together with hardware error flags that signal a parity, overrun, or frame error in the transaction. It proceeds as follows:

- If an error has been detected, the port driver passes the character and error flags to the CLASS_READERROR service routine for processing (for instance, autobaud detection).
- If no error has been detected, the port driver passes the character to the CLASS_PUTNXT service routine.

If either of these service routines returns characters that must be echoed and the line is currently inactive, the port driver must start output. Before dismissing the interrupt, the port driver for a controller with multiple lines should check all lines for pending transactions and empty the silo.

Terminal Class and Port Drivers

17.4 Port Driver Routines

To service *transmitter interrupts*, the port driver first records any reported errors. It then proceeds as follows:

- If TTY\$V_TP_ABORT is set in UCB\$B_TP_STAT, the port driver calls the PORT_ABORT service routine to terminate the transaction.
- If TTY\$V_TP_ABORT is not set, the port driver sets up the next output sequence according to the following priority:

| Transaction | Use |
|-------------|--|
| Preempt | Normally used to send an XON or XOFF character |
| Hold | Normally used for single-character output |
| Burst | Used for multiple-character (DMA) output |

Note that this action can result in an XON or XOFF character appearing in the middle of an escape sequence.

17.5 Class Driver Routines

Table 17–2 lists the class driver routines that are part of the class/port interface. This section describes the functions and context of each listed routine.

A call to a class service routine uses the following instruction format:

```
MOVL  UCB$L_TT_CLASS(R5),R0 ;get pointer to class vector table
JSB   @CLASS_DISCONNECT(R0) ;call class disconnect routine
```

Table 17–2 Class Driver Routines

| Routine | Function |
|------------------|--|
| CLASS_DDT | Pointer to the driver dispatch table |
| CLASS_DISCONNECT | Disconnects a process from a terminal on a nonmodem line |
| CLASS_DS_TRANS | Manages data set transitions |
| CLASS_FORK | Services a port driver's request to create a fork process |
| CLASS_GETNXT | Delivers to the port driver the next character or burst to be output |
| CLASS_PUTNXT | Obtains input characters from the port driver |
| CLASS_SETUP_UCB | Initializes the UCB |
| CLASS_POWERFAIL | Services a power failure |
| CLASS_READERROR | Services a parity, data overrun, or framing error on a terminal line |

Terminal Class and Port Drivers

17.5 Class Driver Routines

17.5.1 CLASS_DDT

This entry in the class driver vector table points to the driver dispatch table (DDT). The CLASS_UNIT_INIT macro uses the CLASS_DDT entry point when moving the address of the DDT into the UCB.

17.5.2 CLASS_DISCONNECT

A port driver calls the CLASS_DISCONNECT routine to indicate to the terminal class driver that the terminal is no longer connected to the system. This is the preferred way of disconnecting a process from a terminal on a nonmodem line.

At the time of the call, R5 must contain the address of the UCB. The CLASS_DISCONNECT routine destroys the contents of R4.

17.5.3 CLASS_DS_TRANS

This CLASS_DS_TRANS routine manages data set state transitions. The port driver's unit initialization routine must call this routine with the transition type MODEM\$_INIT in R1 if the unit is capable of having data set transitions. (Note that, to use modem symbols, the port driver must invoke the \$TTYMDMDEF data structure definition macro (from SYS\$LIBRARY:LIB.MLB).

Input to the CLASS_DS_TRANS routine is as follows:

| | | |
|----|--|---|
| R1 | Transition type, one of the following: | |
| | MODEM\$_INIT | Initialize modem control |
| | MODEM\$_INIT_NORESET | Start modem protocol, but do not initialize signals |
| | MODEM\$_SHUTDOWN | Shut down the line and disconnect the process |
| | MODEM\$_SHUTDOWN_NOHANGUP | Stop modem protocol but do not stop the signals |
| | MODEM\$_DATASET | Data set signal changes |
| R2 | New receive modem mask (if MODEM\$_DATASET is specified in R1) | |
| R5 | Address of UCB | |

The CLASS_DS_TRANS routine destroys the contents of R0 through R4.

17.5.4 CLASS_FORK

A port driver calls the CLASS_FORK routine to create a driver fork process that uses the UCB fork block. The port driver must never initiate a fork directly—it must always call this routine.

The CLASS_FORK routine sets up the fork block in the UCB and performs the other tasks necessary to store context in the fork block, insert it in a processor-specific fork queue, and suspend driver processing. When the fork has taken place, the class driver calls the port driver at its PORT_FORKRET entry point.

Terminal Class and Port Drivers

17.5 Class Driver Routines

At the time of the call, R5 must contain the address of the UCB. The CLASS_FORK routine destroys the contents of R4.

17.5.5 CLASS_GETNXT

The port driver calls the CLASS_GETNXT routine whenever it has completed the current character or burst to obtain the next characters to be output on the unit. If CLASS_GETNXT returns data to the port driver, a timer is set up (unless explicitly disabled) and the interrupt expected bit is set.

At the time of the call, R5 must contain the address of the UCB. Output from the CLASS_GETNXT routine includes the following:

| | |
|-------------------|--|
| R1 | Destroyed |
| R2 | Number of characters (if R3 contains an address) |
| R3 | Character to be output (if UCB\$B_TT_OUTTYPE is positive); address of characters to be output (if UCB\$B_TT_OUTTYPE is negative); or no character (if UCB\$B_TT_OUTTYPE is zero) |
| R4 | Destroyed |
| R5 | Address of UCB |
| R6 through R11 | Destroyed |
| UCB\$B_TT_OUTTYPE | Zero if there is no character to be output; 1 if there is one character to be output; and a negative value if there is a burst to be output |
| UCB\$L_TT_OUTADR | Address of burst to output (if UCB\$B_TT_OUTTYPE is negative) |
| UCB\$W_TT_OUTLEN | Length of burst (if UCB\$B_TT_OUTTYPE is negative) |

17.5.6 CLASS_PUTNXT

The port driver calls the CLASS_PUTNXT routine to pass input characters to the terminal class driver. The CLASS_PUTNXT routine filters characters received from nonpassall units for immediate control sequences. If a slave mode unit (that is, generating no unsolicited input) does not have a read outstanding, the CLASS_PUTNXT routine ignores the input characters, after performing the control-character filtering.

If the input characters are to be echoed to the terminal, CLASS_PUTNXT calls CLASS_GETNXT to notify the port driver.

The CLASS_PUTNXT routine may or may not return output data to the port driver, depending upon the setting of the interrupt-expected bit (UCB\$V_INT) in UCB\$L_STS. If this bit is set, CLASS_PUTNXT does not return data. If it does return data, the terminal port driver should assume that more data may follow, and call CLASS_GETNXT after outputting the returned data.

Input to the CLASS_PUTNXT routine is as follows:

| | |
|----|-----------------|
| R3 | Input character |
| R5 | Address of UCB |

Terminal Class and Port Drivers

17.5 Class Driver Routines

Output from the CLASS_PUTNXT routine is as follows:

| | |
|------------------|---|
| R1 | Destroyed |
| R2 | Number of characters (if R3 contains an address) |
| R3 | Character to be output (if UCB\$B_TT_OUTYPE is positive); address of characters to be output (if UCB\$B_TT_OUTYPE is negative); or no character (if UCB\$B_TT_OUTYPE is zero) |
| R4 | Destroyed |
| R5 | Address of UCB |
| R6 through R11 | Destroyed |
| UCB\$B_TT_OUTYPE | Zero if there is no character to be output; one if there is one character to be output; and a negative value if there is a burst to be output |
| UCB\$L_TT_OUTADR | Address of burst to output (if UCB\$B_TT_OUTYPE is negative) |
| UCB\$W_TT_OUTLEN | Length of burst (if UCB\$B_TT_OUTYPE is negative) |

17.5.7 CLASS_SETUP_UCB

A port driver's unit initialization routine calls CLASS_SETUP_UCB when it is invoked at system startup and power failure.

The CLASS_SETUP_UCB routine initializes the unit's fork block; write queue (UCB\$L_TT_WFLINK); break, passall, and DMA device characteristics; and read timed out dispatch field (UCB\$L_TT_RTIMOU).

In addition, it initializes several UCB fields as follows:

| | |
|------------------|--------------------|
| UCB\$L_TT_LOGUCB | Address of UCB |
| UCB\$L_DEVCHAR | DEV\$V_AVL set |
| UCB\$L_DEVCHAR2 | DEV\$V_RED cleared |
| UCB\$W_TT_CURSOR | 1 |
| UCB\$W_TT_HOLD | Cleared |
| UCB\$W_TT_SPEED | UCB\$W_TT_DESPPEE |
| UCB\$B_TT_PARITY | UCB\$B_TT_DEPARI |
| UCB\$B_DEVTYPE | UCB\$B_TT_DETTYPE |

At the time of the call, R5 must contain the address of the UCB.

17.5.8 CLASS_POWERFAIL

A port driver's unit initialization routine calls the CLASS_POWERFAIL routine when it detects a power failure.

At the time of the call, R5 must contain the address of the UCB.

Terminal Class and Port Drivers

17.5 Class Driver Routines

Output from the CLASS_POWERFAIL routine includes the following:

| | |
|---------------|------------------------------------|
| UCB\$W_STS | UCB\$V_INT cleared; UCB\$V_TIM set |
| UCB\$L_DUETIM | Cleared |

17.5.9 CLASS_READERROR

A port driver calls CLASS_READERROR when it detects a parity, data overrun, or framing error on the terminal line. CLASS_READERROR completes the read operation with error status if a read is active, or simply returns if no read is active.

Input to the CLASS_READERROR routine is as follows:

R3 Character and flags. The following flags are defined:

| | |
|--------|--------------------------------------|
| Bit 12 | Parity error on the given character |
| Bit 13 | Framing error on the given character |
| Bit 14 | Data overrun |

R5 Address of UCB.

Output from the CLASS_READERROR routine is as follows:

| | |
|------------------|---|
| R0 through R3 | Destroyed |
| UCB\$B_TT_OUTYPE | Zero if there is no character to be output; 1 if there is one character to be output; and a negative value if there is a burst to be output |

18 Mapping to I/O Space and the Connect-to-Interrupt Facility

Programs written in VAX MACRO can interface with the I/O subsystem by using VMS RMS, by using the Queue I/O Request (\$QIO) system service, or by mapping to I/O address space and connecting to a device interrupt vector. Programs written in a high-level language can interface with the I/O subsystem using the same methods as a VAX MACRO program, or they can issue the I/O statements specific to that language. In the latter case, the program interfaces with the I/O subsystem by means of the VAX Common Run-Time Procedure Library.

A user program can interface with the I/O subsystem at one of several levels, depending on its requirements. At each level, the user program makes trade-offs between ease of use and execution speed. As a general rule, the closer to the VMS executive that a user program interfaces, the less overhead is involved in the I/O operation. The connect-to-interrupt capability offers the least overhead.

A process with suitable privileges can connect to a device interrupt vector or map the system's I/O address space into process virtual address space or both. Connecting to a device interrupt vector allows your process to respond to interrupts from the device with minimal overhead. Mapping system I/O address space allows your process to access device registers from the main program or from an AST procedure.

A process normally uses these features for devices that do not have VMS drivers. These devices must not be direct memory access (DMA) devices, and they must be attached to the UNIBUS or Q22 bus. Examples of such devices are the AXV11-C and the KW11-P.

18.1 I/O Address Space

In a VAX system, I/O address space is assigned physical address locations of 20000000_{16} and higher ($F20000_{16}$ and higher for VAX-11/730 and VAX-11/780 systems). I/O address space contains device registers that a driver or user process can read and write to control a device. Each device controller has an associated control and status register (CSR) in I/O address space. Device registers for each device are located at an offset from the device's CSR.

Macros of the format \$IOxxxDEF (where xxx represents a specific VAX system), contained in SYS\$LIBRARY:LIB.MLB, define symbols describing the layout of I/O address space. Table 18-1 describes these macros and the symbols they define for each VAX system.

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.1 I/O Address Space

Table 18–1 Symbols Defined by the \$IOxxxDEF Macros

| Macro | Symbol(s) | Meaning | Value (Hex) |
|--|----------------------|---|-------------|
| VAX 6200 Series | | | |
| \$IO9CCDEF | IO9CC\$AL_IOBASE | Start of I/O address space | 20000000 |
| | IO9CC\$_BIWINDOW | Offset to node 0 window space | 400000 |
| | IO9CC\$_BIWSIZ | Size of window space | 40000 |
| | IO9CC\$_PERXBI | Size of adapter address space | 2000000 |
| VAX 8530/8550/8700/8800/8830/8840 | | | |
| \$IO8NNDEF | IO8NN\$AL_NBIB_0 | Start of I/O address space for VAXBI 0 | 20000000 |
| | IO8NN\$AL_NBIB_1 | Start of I/O address space for VAXBI 1 | 22000000 |
| | IO8NN\$AL_NBIB_2 | Start of I/O address space for VAXBI 2 | 24000000 |
| | IO8NN\$AL_NBIB_3 | Start of I/O address space for VAXBI 3 | 26000000 |
| | IO8NN\$AL_NBIB_4 | Start of I/O address space for VAXBI 4 | 28000000 |
| | IO8NN\$AL_NBIB_5 | Start of I/O address space for VAXBI 5 | 2A000000 |
| | IO8NN\$AL_NODESP | Offset to node 0 window space | 400000 |
| IO8NN\$AL_NDSPER | Size of window space | 40000 | |
| VAX 8200/8250/8300/8350 | | | |
| \$IO8SSDEF | IO8SS\$AL_NODESP | Address of node 0 window space | 20400000 |
| | IO8SS\$AL_NDSPER | Size of window space | 40000 |
| VAX 8600/8650/8670 | | | |
| \$IO790DEF | IO790\$AL_IOA0 | Start of I/O address space for SBIO | 20000000 |
| | IO790\$AL_IOA1 | Start of I/O address space for SBI1 | 22000000 |
| | IO790\$AL_UB0SP | Offset to start of adapter address space for first UNIBUS | 24000000 |
| VAX–11/780 and VAX–11/785 | | | |
| \$IO780DEF | IO780\$AL_IOBASE | Start of I/O address space | 20000000 |
| | IO780\$AL_UB0SP | Start of adapter address space for first UNIBUS | 20100000 |
| VAX–11/750 | | | |
| \$IO750DEF ¹ | IO750\$AL_IOBASE | Start of I/O address space | F20000 |
| | IO750\$AL_UBBASE | Start of UBA0 adapter register space | F30000 |
| | IO750\$AL_MBBASE | Start of MBA0 adapter register space | F28000 |
| | IO750\$AL_UB0SP | Start of adapter address space for first UNIBUS | FC0000 |
| VAX–11/730 and VAX–11/725 | | | |
| \$IO730DEF | IO730\$AL_IOBASE | Start of I/O address space | F20000 |
| | IO730\$AL_UB0SP | Start of adapter address space for UNIBUS | FC0000 |

¹The VAX–11/750 system has fixed MASSBUS adapters (UBBASE, MBBASE) in contrast to the VAX–11/780 system, which has floating MASSBUS adapters, and the VAX–11/730, which does not have MASSBUS adapters.

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.1 I/O Address Space

Table 18–1 (Cont.) Symbols Defined by the \$IOxxxDEF Macros

| Macro | Symbol(s) | Meaning | Value (Hex) |
|-----------------------------|-----------------|--|-------------|
| MicroVAX 3600 Series | | | |
| \$IO650DEF | IO650\$AL_QBOSP | Start of adapter address space for Q22 bus | 20000000 |
| MicroVAX II | | | |
| \$IOUV2DEF | IOUV2\$AL_QBOSP | Start of adapter address space for Q22 bus | 20000000 |
| MicroVAX I | | | |
| \$IOUV1DEF | IOUV1\$AL_QBOSP | Start of adapter address space for Q22 bus | 20000000 |

The number of registers and their locations varies from device to device. The *PDP-11 Peripherals Handbook* provides the necessary information for DIGITAL-supplied devices. The *VAX Hardware Handbook* contains information about the layout of I/O address space.

From the symbols defined by the macros described in Table 18–1, you can derive the starting physical addresses of UNIBUS or Q22 bus adapter address space for the various VAX systems. Table 18–2 lists the starting physical addresses for UNIBUS adapters on the VAX 8600/8650/8670, VAX–11/780, VAX–11/785, VAX–11/750, VAX–11/730, and VAX–11/725 systems, as well as the starting physical addresses for MicroVAX I, MicroVAX II, and MicroVAX 3600-series Q22 bus interface address space.

Note: To access UNIBUS device CSRs you must add $3E000_{16}$ to the addresses listed in Table 18–2. This operation is not necessary when you use the values supplied for MicroVAX/Q22 bus systems.

For VAX 8530/8550/8700/8800 and VAX 8200/8250/8300/8350 systems, Example 18–1 illustrates the calculations that are necessary to determine the location of the adapter address space for a given DWBUA adapter on a VAXBI bus. For additional information on the layout of VAXBI I/O address space, see the discussion and illustrations in Section 14.2.

Table 18–2 UNIBUS and Q22 Bus Adapter Address Space

| UNIBUS Adapter Number | VAX–11/725 | VAX–11/730 | VAX–11/750 | VAX–11/780 | MicroVAX 3600 Series | VAX 8600 SBI0/SBI1 |
|-----------------------|------------|------------|------------|------------|----------------------|--------------------|
| | | | | VAX–11/785 | MicroVAX II | VAX 8650 SBI0/SBI1 |
| | | | | | MicroVAX I | VAX 8670 SBI0/SBI1 |
| 0 | 00FC0000 | 00FC0000 | 00FC0000 | 20100000 | 20000000 | 20100000/22100000 |
| 1 | – | 00F80000 | 00F80000 | 20140000 | – | 20140000/22140000 |
| 2 | – | – | – | 20180000 | – | 20180000/22180000 |
| 3 | – | – | – | 201C0000 | – | 201C0000/221C0000 |

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.1 I/O Address Space

Example 18-1 Locating the Adapter Address Space of a DWBUA Adapter on a VAXBI Bus

For VAX 8530/8550/8700/8800:

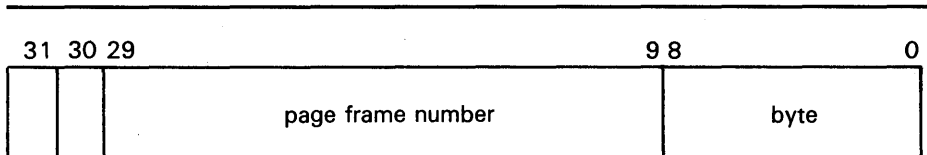
```
IO8NN$AL_NBIB_n           ;Start of I/O address space for given VAXBI bus
+ IO8NN$AL_NODESP         ;Offset to window space
+ IO8NN$AL_NDSPER * VAXBI-node-ID-of-DWBUA ;Offset to given VAXBI node window space
+ UNIBUS-address-of-device-CSR ;Offset to device CSR
```

For VAX 8200/8250/8300/8350:

```
IO8SS$AL_NODESP         ;Start of window space
+ IO8SS$AL_NDSPER * VAXBI-node-ID-of-DWBUA ;Offset to given VAXBI node window space
+ UNIBUS-address-of-device-CSR ;Offset to device CSR
```

For most VAX processors, the *page frame number* (PFN) of a physical page in memory is contained in bits 9 through 29 of its physical address (see Figure 18-1). Bit 29 of the address is clear to indicate a physical memory address and set to indicate an address in I/O address space. Bits 0 through 8 specify the byte address within the page.

Figure 18-1 Format of a Physical Address



ZK-4845-85

18.2 PFN Mapping

For a process to gain access from an outer access mode to I/O address space or to any page of physical memory, it must map that page into its virtual address space. When a VMS process maps a page by specifying its page frame number, it completely bypasses VMS memory management and creates its own window to the page. As a result, the protection functions that VMS normally performs are not performed for PFN mapping:

- No checks are performed to ensure that no other VMS processes are mapped to the page and modifying it.
- No reference count is maintained. A process can delete a global section mapped by page frame numbers when other processes are still using it; this is not the case for other types of global sections.

Modifying pages mapped by page frame numbers can have unpredictable results and can adversely affect system operation, especially if the operating system is also using these pages or accessing devices whose registers are in the same pages. Because PFN-mapped pages are not inherently protected from such modification, a process must have the PFNMAP privilege to use this capability.

When used for PFN mapping, the Create and Map Section (\$CRMPSC) system service designates the specified page(s) as a global or private section and maps the section into the requesting process's virtual address space. The pages can be located anywhere in the VAX system's local memory, in MA780 memory (if a multiport memory unit is connected to the system), or in I/O address space.

The format and conventions for PFN mapping (that is, mapping a physical page frame section) are similar to those for mapping a disk file section. The \$CRMPSC system service has the following general formats:

VAX MACRO Format

```
$CRMPSC [inadr] [,retadr] [,acmode] [,flags] [,gsdnam] [,ident] -  
        [,relpag] [,chan] [,pagcnt] [,vbn] [,prot] [,pfc]
```

High-Level Language Format

```
SY$CRMPSC (([inadr] [,retadr] [,acmode] [,flags] [,gsdnam] [,ident]  
            [,relpag] [,chan] [,pagcnt] [,vbn] [,prot] [,pfc])
```

The **relpag**, **chan**, and **pfc** arguments are not applicable to mapping by page frame number. The **inadr**, **retadr**, **acmode**, **gsdnam**, **ident**, and **prot** arguments have the same functions regardless of whether you specify page frame number mapping. The *VMS System Services Reference Manual* further describes these arguments.

The following arguments can have values specific to PFN mapping:

Arguments

[flags]

Mask defining the section type and characteristics. This mask is the logical OR of the flag bits you want to set. The \$SECDEF macro defines symbolic names for the flag bits in the mask. The SEC\$M_PFNMAP flag bit must be set to indicate mapping by page frame number. The SEC\$M_PFNMAP flag setting identifies the memory for the section as starting at the page frame

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.2 PFN Mapping

number specified in the **vbn** argument and extending for the number of pages specified in the **pagcnt** argument.

If appropriate, the following flags can also be set:

| Flag | Description |
|---------------|--|
| SEC\$M_GBL | Pages form a global section. The default is private section. |
| SEC\$M_EXPREG | Pages are mapped into the first available space. By default, pages are mapped into the range specified by the inadr argument. |
| SEC\$M_WRT | Pages form a read/write section. By default, pages form a read-only section. |
| SEC\$M_PERM | Pages are permanent. By default, pages are temporary. |
| SEC\$M_SYSGBL | Pages form a system global section. By default, pages form a group global section. |

You must not set either the SEC\$M_CRF (copy-on-reference) or the SEC\$M_DZRO (demand-zero) bit when mapping by page frame number.

[pagcnt]

Number of pages in the section; the value of this argument must not be zero.

[vbn]

Page frame number of the first page to be mapped (as opposed to this argument's normal usage identifying the starting virtual block number (vbn) within a disk file). When you are mapping more than one page with a single \$CRMPSC system service request, the pages are physically contiguous starting with the specified page.

18.2.1 Notes on PFN Mapping

The following considerations apply to PFN mapping.

- 1** An error in mapping UNIBUS or Q22 bus adapter address space or a reference to a nonexistent bus address causes a UNIBUS adapter error. However, this error does not cause a system failure (except on a VAX 8530/8550/8700/8800/8830/8840, VAX-11/750, or VAX-11/730 system, where a machine check will occur). Rather, an entry is made in the system error log file and the user program continues executing (probably with erroneous results). The process is not notified of the UNIBUS adapter error.
- 2** On systems where a UNIBUS power failure can occur without causing a system failure, a user process receives a machine check exception, if it is using perprocess space mapping when accessing UNIBUS or Q22 bus I/O address space during the failure. To survive this exception, the process must have a condition handler to deal with machine check exceptions. The *VMS System Services Reference Manual* discusses condition handlers in detail.
- 3** During recovery from a power failure, the processor spends a considerable amount of time (perhaps 10 to 60 milliseconds) at IPL 31. This action blocks user processes from executing during the recovery.

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.2 PFN Mapping

- 4 When a process requests deletion of a PFN-mapped page, VMS will wait until there is no direct I/O outstanding for the process before deleting the page. This is because no reference count is maintained for PFN-mapped pages. (For example, VMS cannot determine whether outstanding direct I/O is for the PFN-mapped page or not.) Applications using devices that have direct I/O perpetually outstanding, such as the DR32, must not delete PFN-mapped pages because this will cause the process to hang in the MWAIT state.

Once you have mapped to I/O address space, you can read data from a device data buffer register, because the device registers are now addressable as part of your process's virtual memory. The UNIBUS adapter performs the actual mapping of VAX virtual addresses to the 18-bit UNIBUS addresses that correspond to device registers. Likewise, the MicroVAX 3600-series, MicroVAX II, or MicroVAX I system performs the mapping of virtual addresses to 22-bit Q22 bus addresses that correspond to device registers.

See Section 5.2 for a list of restrictions that apply to instruction references to device register address space.

18.3 Connecting to an Interrupt Vector

You can use the \$QIO system service with an appropriate function code to connect to a device interrupt vector and to specify a user-supplied interrupt service routine that VMS executes when the designated device interrupts. Connecting to a device interrupt vector allows you to do the following:

- Respond to an interrupt within a short time
- Preempt other system processing to handle a real-time event, for example, a clock interrupt
- Buffer data from a device in real time and return the data to the process at a later time
- Set an event flag or queue an AST to your process after receiving the interrupt

An interrupt service routine specified in your process allows it to perform some of the functions normally performed by a device driver. The connect-to-interrupt facility, with its VMS-supplied driver (CONINTERR), thus allows you to avoid writing a full device driver and loading it into the operating system.

If you must access device registers from user mode (that is, from the main program or a user-mode AST procedure), you must use the Create and Map Section (\$CRMPSC) system service to map I/O address space, specifying page frame number (PFN) mapping. The service creates a global or private section that maps the specified I/O pages into your process's virtual address space with suitable protection. The process can then gain access to I/O address space using perprocess virtual addresses (see Section 18.1 for additional discussion).

You do not need to map I/O address space to access device registers from any of the following routines specified in the \$QIO call connecting to an interrupt vector:

- Unit initialization routine

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.3 Connecting to an Interrupt Vector

- Start-I/O routine
- Interrupt service routine
- Cancel-I/O routine

These routines execute in system context and thus can access UNIBUS or Q22 bus I/O address space, which is mapped as part of system address space.

18.3.1 Performing the Connect-to-Interrupt

Connecting to a device interrupt vector allows your program to receive notification of an interrupt from a designated device by any combination of the following means:

- By execution of a user-supplied interrupt service routine
- By the setting of an event flag
- By execution of an AST procedure that gains control in process context

In addition, you can specify a cancel-I/O routine that is executed when the process disconnects from the interrupt vector or is deleted.

Before your program can run, the system manager must have performed the following actions at system generation time:

- Specify the *REALTIME_SPTS* system parameter, reserving system page table entries for use by real-time processes. These system page-table entries are used to map process-specified buffers in system address space (see the **p1** argument description in Section 18.3.2). The *REALTIME_SPTS* parameter value must be greater than or equal to the number of pages in buffers specified by processes connected to interrupt vectors.
- Configure the real-time device by issuing a **CONNECT** command to the System Generation Utility. This command names the device; its vector, register, and adapter addresses; and a skeletal driver (**CONINTERR**) for the device. (See the description of the **CONNECT** command in Section 15.2.2 and in the *VMS System Generation Utility Manual*.)

At run time the process calls the **\$ASSIGN** system service to associate a channel with the device. To connect to the device interrupt vector, the process issues a **\$QIO** call specifying the **IO\$_CONINTREAD** or **IO\$_CONINTWRITE** function code and as many of the following items as are appropriate:

- An interrupt service routine to be executed when the device generates an interrupt.
- A unit initialization routine.
- A start-I/O routine.
- A cancel-I/O routine.
- A buffer containing the code to be executed in system context, data (that is, the previously-listed routines), or both.

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.3 Connecting to an Interrupt Vector

- An AST procedure to execute, an event flag to be set after the interrupt service routine (if any) completes, or both. (If an AST procedure is specified, an AST parameter may also be specified.)

A nonprivileged process (that is, lacking the CMKRNL privilege) can also connect to an interrupt vector, but it can only specify an AST procedure to be executed or an event flag to be set (or both) when an interrupt is generated. The process can also map the page in UNIBUS or Q22 bus I/O address space containing the device registers (see Section 18.2).

18.3.2 \$QIO Connect-to-Interrupt Request to Driver

The format of the \$QIO system service to connect to an interrupt vector follows. This explanation is limited to connecting to an interrupt vector. For a detailed description of the \$QIO system service, see the *VMS System Services Reference Manual*.

VAX MACRO Format

```
$QIO [efn] [,chan] ,func [,iosb] [,astadr] [,astprm] -  
      [,p1] [,p2] [,p3] [,p4] [,p5] [,p6]
```

High-Level Language Format

```
SY$QIO ([efn] [,chan] ,func [,iosb] [,astadr] [,astprm]  
        [,p1] [,p2] [,p3] [,p4] [,p5] [,p6])
```

Arguments

[efn]
[chan]
[iosb]
[astadr]
[astprm]

These arguments apply to the \$QIO system service completion, not to device interrupt actions. For an explanation of these arguments, see the description of the \$QIO system service in the *VMS System Services Reference Manual*.

func

Function code of IO\$_CONINTREAD or IO\$_CONINTWRITE. The IO\$_CONINTWRITE function code allows locations in the buffer pointed to by the p1 argument to be modified; the IO\$_CONINTREAD function code makes the buffer contents read-only.

[p1]

Address of a descriptor for the buffer containing code and/or data. The first longword records the number of bytes in the buffer; the second longword records the address of the buffer. The buffer size must not exceed 65,535 bytes.

[p2]

Address of an entry point list. The list consists of four longwords that contain offsets into the buffer (specified in the p1 argument) of the entry points

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.3 Connecting to an Interrupt Vector

of process-specified routines. These longwords and their contents¹ are as follows:

| Symbol | Meaning |
|--------------|---------------------------------------|
| CIN\$_INIDEV | Offset to unit initialization routine |
| CIN\$_START | Offset to start-I/O routine |
| CIN\$_ISR | Offset to interrupt service routine |
| CIN\$_CANCEL | Offset to cancel-I/O routine |

[p3]

Longword containing flags and an optional event flag number specification.

The low-order word contains the inclusive-OR of flags describing options to the connect-to-interrupt facility. The flags and their meanings are as follows:

| Flag | Meaning |
|--------------|---|
| CIN\$_EFN | Set event flag on interrupt. |
| CIN\$_USECAL | Use CALL interface to process-specified routines (default is JSB interface). |
| CIN\$_REPEAT | Leave process connected to the interrupt vector until the connection is canceled. |
| CIN\$_INIDEV | Process-specified unit initialization routine is in the buffer specified in the p1 argument. |
| CIN\$_START | Process-specified start-I/O routine is in buffer. |
| CIN\$_ISR | Process-specified interrupt service routine is in buffer. |
| CIN\$_CANCEL | Process-specified cancel-I/O routine is in buffer. |

The high-order word specifies the number of the event flag to be set when an interrupt occurs. This number is expressed as an offset to CIN\$_EFNUM.

For example, to specify that your interrupt service routine is in the buffer and to set event flag 4, code **p3** as follows:

```
P3 = <CIN$_ISR!CIN$_EFN!4@CIN$_EFNUM>
```

See note 3 in the following description for additional information on these flags.

[p4]

Address of the entry mask of an AST procedure to be called as the result of an interrupt (see Section 18.3.5).

[p5]

AST parameter to be passed to the AST procedure (used as the AST parameter only if the process-supplied interrupt service routine does not overwrite the value).

¹ The listed symbols are defined by the \$CINDEF macro located in the library SYS\$LIBRARY:LIB.MLB.

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.3 Connecting to an Interrupt Vector

[p6]

Number of AST control blocks to preallocate in anticipation of fast, recurrent interrupts from the device.

Condition Values Returned

| | |
|-----------------|--|
| SS\$_NORMAL | System service successfully completed. |
| SS\$_ACCVIO | The caller does not have the appropriate access to the buffer specified in the p1 argument or to the entry point list specified in the p2 argument. |
| SS\$_BADPARAM | The size of the buffer specified in the p1 argument exceeds 65,535 bytes, or the number of preallocated AST control blocks specified in the P6 argument exceeds 65,535. |
| SS\$_DISCONNECT | A connection is already outstanding for the device, or a condition described as follows in note 2b has occurred. |
| SS\$_EXQUOTA | The process has exceeded its direct I/O limit quota or its AST limit quota. |
| SS\$_ILLEFC | An illegal event flag number was specified. |
| SS\$_INSFMEM | Insufficient system dynamic memory is available to complete the system service. |
| SS\$_INSFSPTS | Insufficient system page-table entries are available to double map the process buffer. (The value of the <i>REALTIME_SPTS</i> SYSGEN parameter must be increased.) |
| SS\$_NOPRIV | The process does not have the CMKRNL privilege. This privilege is only required if the user specifies a buffer to be used by the process and the process-specified kernel-mode routines. |
| SS\$_UNASEFC | The process is not associated with the cluster containing the specified event flag. |

Privilege Restrictions

The connect-to-interrupt \$QIO call does not require privileges if no shared buffer is specified. If the request specifies a buffer descriptor argument (that is, **p1**), the process must have the CMKRNL privilege.

Resources Required/Returned

A connect-to-interrupt request updates the process quota values as follows:

- Subtracts the number of preallocated AST control blocks in the **p6** argument from the number of outstanding ASTs remaining for the process (ASTCNT)
- Subtracts 1 (for the \$QIO) from the direct I/O count (DIOCNT)

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.3 Connecting to an Interrupt Vector

Notes

- 1 After the \$QIO call is issued, the operation is not completed until the process or the connect-to-interrupt driver cancels I/O on the channel.
- 2 The connect-to-interrupt driver can cancel I/O on the channel for a number of reasons, including the following:
 - a. The driver cannot set the specified event flag, perhaps because the process disassociated from the common event flag cluster after requesting that a flag in that cluster be set.
 - b. The driver cannot reallocate AST control blocks quickly enough. This condition can occur because not enough AST control blocks (**p6** argument) were specified, not enough pool space is available for the requested AST control blocks, or the process ASTCNT quota is exhausted.
 - c. The driver cannot queue the AST to the process.
- 3 If no event flag setting was requested in the **p3** argument and if no AST procedure was specified in the **p4** argument, **p6** is ignored and no AST control blocks are preallocated. If you requested that an event flag be set or specified an AST procedure, but did not preallocate any AST control blocks (that is, **p6** is zero), one AST control block is preallocated automatically, because the system needs one control block to set any event flag or to deliver any ASTs.

If you request an event flag and/or an AST procedure and if you preallocate any AST control blocks, the CIN\$M_REPEAT bit is set automatically in the longword specified in the **p3** argument. Thus, as long as you preallocate any AST control blocks, your process will automatically remain connected to the interrupt vector to receive repeated interrupts until the process is disconnected from the interrupt vector.

If the CIN\$M_REPEAT flag is not set, the process is disconnected from the interrupt vector after the first successful interrupt, and a status code of SS\$_NORMAL is returned.

18.3.3 The Connect-to-Interrupt Driver (CONINTERR.EXE)

The VMS connect-to-interrupt driver (CONINTERR) provides a driver interface to the system on behalf of the process. CONINTERR connects the process to the device by executing the following steps:

- 1 Validates the arguments to the \$QIO system service call, such as the accessibility of the buffer specified in argument **p1** to the process, and the number of the event flag optionally specified in the **efn** argument.
- 2 Locks the physical pages of the buffer into physical memory, and maps the pages using system page-table entries allocated by the *REALTIME_SPTS* SYSGEN parameter.
- 3 Constructs argument lists and calling interfaces to the process-specified routines by storing values in the device's unit control block (UCB).
- 4 Allocates the specified number of AST control blocks to the process, and inserts each block in a queue in the device's UCB.

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.3 Connecting to an Interrupt Vector

- 5 Transfers control to VMS to queue the connect-to-interrupt I/O packet to the CONINTERR start-I/O routine.

When the CONINTERR start-I/O routine gains control, it passes control, by means of a user-specified JSB or CALLS instruction interface, to the process-specified start-I/O routine. This routine usually initializes the device and may also start device activity.

When the device generates an interrupt, the CONINTERR interrupt service routine gains control. This routine transfers control to the process-supplied interrupt service routine.

18.3.4 Process-Specified Routines

Any routines that the process specifies in the connect-to-interrupt call, with the exception of the AST procedure, are double-mapped, once in process address space and once in system address space. Each routine executes in kernel mode at an appropriate IPL:

| Routine | IPL |
|--|------------------------|
| Unit initialization routine (after power recovery) | IPL\$_POWER (IPL 31) |
| Start-I/O routine | IPL\$_QUEUEAST (IPL 6) |
| Interrupt service routine | Device IPL |
| Cancel-I/O routine | IPL\$_QUEUEAST (IPL 6) |

The process must have CMKRNL privilege. Each routine must

- Be position independent
- Follow the rules for accessing I/O address space as described in Section 5.2
- Access only data within the buffer or nonpageable locations in system address space
- Perform any necessary synchronization of access to data in the shared buffer
- Save any registers it uses (unless otherwise noted in the remaining sections of this chapter)
- Exit properly
- Not incur exceptions
- Not perform lengthy processing
- Not dispatch to code outside the buffer specified in the **p1** argument to the \$QIO system service call

Only VAX MACRO or VAX BLISS-32 should be used to code process-specified routines in system address space or any references to I/O address space. There is no assurance that the code generated by compilers for other languages will satisfy all the constraints described in this section.

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.3 Connecting to an Interrupt Vector

The following constraints apply to process-specified routines in system address space (that is, in the buffer specified in the **p1** argument to the **\$QIO** call that establishes the connection to the interrupt vector):

- The compiler must generate position-independent code for the routines.
- The generated code and data must be contiguous in virtual address space.
- No calls can be made to any procedure outside the buffer. (This restriction includes calls to routines in the VAX Run-Time Library.)
- For any references to I/O address space, the generated code must follow the rules for accessing I/O address space discussed in Section 5.2.

You can find additional help for writing a start-I/O routine, interrupt service routine, unit initialization routine, or cancel-I/O routine in Sections 8, 9, 11.1, and 11.2, respectively. Additionally, you may find useful the several program examples of connecting to an interrupt vector with which this chapter concludes.

18.3.4.1 Unit Initialization Routine

During recovery from a power failure, VMS calls the **CONINTERR** unit initialization routine. This routine marks the device as on line in the **UCB\$L_STS** field, stores the UCB address in the **IDB\$L_OWNER** field, and then transfers control to the process-specified unit initialization routine. The process-specified routine executes in system context at **IPL\$_POWER** (IPL 31).

If the process specified a **JSB** interface, the process unit initialization routine gains control with the following register settings:

| | |
|----|----------------|
| R0 | Address of UCB |
| R4 | Address of CSR |
| R5 | Address of IDB |
| R6 | Address of DDB |
| R8 | Address of CRB |

If the process specified a **CALL** interface, the process unit initialization routine gains control with an argument list pointed to by **AP**:

| | |
|--------|---------------------|
| 00(AP) | Argument count of 5 |
| 04(AP) | Address of CSR |
| 08(AP) | Address of IDB |
| 12(AP) | Address of DDB |
| 16(AP) | Address of CRB |
| 20(AP) | Address of UCB |

The process-specified unit initialization routine may initialize device registers. It must follow these conventions:

- Not lower IPL nor obtain any spin locks
- Save and restore all registers it uses, other than R0 through R3
- Restore the stack to its original state before exiting

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.3 Connecting to an Interrupt Vector

- Exit with an RSB instruction (for a JSB interface) or a RET instruction (for a CALL interface)

For additional information on writing a unit initialization routine, see Section 11.1.

18.3.4.2 Start-I/O Routine

The process-specified start-I/O routine executes in process context in system space at IPL\$_QUEUEAST (IPL 6), holding the QUEUEAST fork lock in a VMS multiprocessing environment. It is entered from the CONINTERR start-I/O routine.

If the process specified a JSB interface, the process start-I/O routine gains control with the following register settings:

| | |
|----|----------------------------------|
| R2 | Address of counted argument list |
| R3 | Address of IRP |
| R5 | Address of UCB |

If the process specified a CALL interface, the process start-I/O routine gains control with an argument list pointed to by AP:

| | |
|--------|---|
| 00(AP) | Argument count of 4 |
| 04(AP) | System-mapped address of process buffer |
| 08(AP) | Address of IRP |
| 12(AP) | System-mapped address of the device's CSR |
| 16(AP) | Address of UCB |

The process-specified start-I/O routine may set up device registers. It must follow these conventions:

- Maintain an IPL equal to or higher than IPL\$_QUEUEAST (IPL 6), and exit at IPL 6. (If it raises IPL, the routine should first save the current IPL on the stack for later use in restoring IPL.) In a VMS multiprocessing system, the process-specified start-I/O routine must suitably synchronize any access of device registers with the process-specified interrupt service routine. To do so, each routine must obtain the appropriate device lock, using the VMS-supplied macro DEVICELOCK. Before exiting, each routine releases ownership of the device lock using the DEVICEUNLOCK macro. (See the discussion of these macros in Appendix B.)
- Save and restore all registers it uses, other than R0 through R4.
- Restore the stack to its original state before exiting.
- Exit with an RSB instruction (for a JSB interface) or a RET instruction (for a CALL interface).

For additional information on writing a start-I/O routine, see Chapter 8.

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.3 Connecting to an Interrupt Vector

18.3.4.3 Interrupt Service Routine

A process-specified interrupt service routine is entered when an interrupt from the device occurs. This routine executes in system context at device IPL.

If the process specified a JSB interface, the process interrupt service routine gains control with the following register settings:

R2 Address of counted argument list
R4 Address of IDB
R5 Address of UCB

If the process specified a CALL interface, the process interrupt service routine gains control with an argument list pointed to by AP:

00(AP) Argument count of 5
04(AP) System-mapped address of process buffer
08(AP) Address of AST parameter
12(AP) System-mapped address of the device's CSR
16(AP) Address of IDB
20(AP) Address of UCB

The process-specified interrupt service routine usually performs one or more of the following steps:

- 1 Copies the contents of device registers into the shared buffer or the AST parameter
- 2 Writes to a device register to clear the interrupt condition, if such an operation is required for the device
- 3 Restarts the device, or returns an offset, a byte count, or actual data as an AST parameter
- 4 Returns an interrupt status to the VMS connect-to-interrupt driver (CONINTERR)

The process-specified interrupt service routine, like those supplied by VMS, has the following characteristics:

- It is mapped in system address space.
- It executes on the interrupt stack.
- It executes at the IPL of the device that requested the interrupt.

The routine must follow these conventions:

- Maintain an IPL equal to or higher than device IPL. (If it raises IPL, the routine should first save the current IPL on the stack for later use in restoring IPL.) In a VMS multiprocessing system, if the process-specified start-I/O routine or cancel-I/O routine accesses device registers or UCB fields also accessed by the process-specified interrupt service routine, the routines must suitably synchronize. To do so, each routine must obtain the appropriate device lock, using the VMS-supplied macro DEVICELock. Before exiting, each routine releases ownership of the device lock using the DEVICEUNLOCK macro. (See the discussion of these macros in Appendix B.)
- Save and restore all registers it uses, other than R0 through R4.

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.3 Connecting to an Interrupt Vector

- Restore the stack to its original state before exiting.
- Set or clear the low bit of R0, as a status value, before exiting. The status values are as follows:

| Bit 0 of R0 | Meaning |
|-------------|--|
| Clear | Dismiss the interrupt. The process is not notified of the interrupt. |
| Set | Set the event flag if CIN\$M_EFN bit is set in the p3 argument to the \$QIO system service call, and queue the AST if p4 specifies an AST procedure. |

- Return to the CONINTERR interrupt service routine with a RET instruction (for a CALL interface) or RSB instruction (for a JSB interface).

Depending on the interrupt status returned in R0, the CONINTERR interrupt service routine queues a fork process to run at a lower IPL (IPL\$_QUEUEAST). Then the interrupt service routine exits from the interrupt with an REI instruction. When the CONINTERR fork process gains control, it queues an AST or posts an event flag to the process (or both).

For additional information on writing an interrupt service routine, see Chapter 9.

18.3.4.4 Cancel-I/O Routine

When the user process issues a cancel-I/O request for a device connected to the process, the CONINTERR cancel-I/O routine first checks to determine whether the process can indeed cancel I/O for this device. If it can, the CONINTERR cancel-I/O routine transfers control to the process-specified cancel-I/O routine. This routine executes in system context at IPL 8 (fork IPL).

If the process specified a JSB interface, the process cancel-I/O routine gains control with the following register settings:

| | |
|----|--|
| R2 | Negated value of channel index number |
| R3 | Address of current IRP |
| R4 | Address of PCB for process canceling the I/O |
| R5 | Address of UCB |

If the process specified a CALL interface, the process cancel-I/O routine gains control with an argument list pointed to by AP:

| | |
|--------|--|
| 00(AP) | Argument list count of 4 |
| 04(AP) | Negated value of channel index number |
| 08(AP) | Address of current IRP |
| 12(AP) | Address of PCB for process canceling the I/O |
| 16(AP) | Address of UCB |

The process-specified cancel-I/O routine may clear device registers and set the UCB\$V_CANCEL bit in UCB\$L_STS. It must follow these conventions:

- Maintain an IPL equal to or higher than IPL\$_QUEUEAST (IPL 6), and exit at IPL 6. (If it raises IPL, the routine should first save the current IPL on the stack for later use in restoring IPL.) In a VMS multiprocessing

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.3 Connecting to an Interrupt Vector

system, if the process-specified cancel-I/O routine accesses device registers or UCB fields also accessed by the process-specified interrupt service routine, the routines must suitably synchronize. To do so, each routine must obtain the appropriate device lock, using the VMS-supplied macro `DEVICELock`. Before exiting, each routine releases ownership of the device lock using the `DEVICEUNLOCK` macro. (See the discussion of these macros in Appendix B.)

- Save and restore all registers it uses, other than R0 through R3.
- Place a completion status in R0 and R1. VMS places the values in these registers in the I/O status block associated with the connect-to-interrupt `$QIO` call.
- Restore the stack to its original state before exiting.
- Exit with an RSB instruction (for a JSB interface) or a RET instruction (for a CALL interface).

For additional information on writing a cancel-I/O routine, see Section 11.2.

18.3.5 AST Procedure

The AST procedure that you specify in the call to the `$QIO` system service for the connect-to-interrupt operation gains control in process context. This routine usually performs one or more of the following steps:

- 1 Reads or writes device registers if the process mapped I/O address space.
- 2 Interprets data. Use caution, however, because any processing done by the AST procedure can be interrupted by a device interrupt, which might store more data or modify the buffer's contents.
- 3 Calls the Cancel I/O on Channel (`$CANCEL`) system service to disconnect the process from the interrupt. Once the process is completely disconnected, the `CONINTERR` driver clears all interrupts for the driver.

18.4 Real-Time Applications Examples

To understand how the connect-to-interrupt facility is useful for programming real-time devices, consider devices used in three types of real-time applications:

- 1 Asynchronous event reporting without data—devices that generate an interrupt as the result of an external event not initiated by a programmed request.
- 2 Program-driven data collection—devices that generate an interrupt as the result of a programmed request, and make the result of the request available as data in a device register at the time of the interrupt.
- 3 Asynchronous event reporting with data—one device triggers another device by generating an interrupt that causes a programmed request to be sent to the other device, which in turn generates an interrupt.

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.4 Real-Time Applications Examples

Examples of these three types of real-time applications and models of programs to handle the devices follow.

Note: The configurations described in the examples in this section are not officially supported; DIGITAL does not provide device driver, UETP, or diagnostic support for certain devices mentioned. (In fact, DIGITAL has officially retired the -K series models (AD11-K and AM11-K A/D Converter). The examples are provided merely as possible models for users who wish to design real-time applications using unsupported devices or configurations.

The files in the SYS\$EXAMPLES directory whose names begin with "LABIO" illustrate an application using the connect-to-interrupt technique. Included is a program example illustrating data definitions and coding used to connect to a device interrupt vector.

18.4.1 Example 1: KW11-W Watchdog Timer

This type of device reports asynchronous external events: it generates an interrupt as a result of an external event not initiated by a programmed request. The only data of interest to be passed to the user process is the occurrence of the external event. Such devices include contact and/or solid state interrupts, and clocks or counters. The program may need to activate clock and counter devices by means of a programmed request, but any subsequent interrupts are the result of external events only.

In this example, a dual-processor system uses two KW11-W watchdog timers connected back-to-back to monitor CPU failures. Each processor must arm its timer at regular intervals to prevent the timer from operating a relay that outputs an alarm signal. The alarm output of each timer is connected to the receive input of the other watchdog. If processor A fails and its watchdog times out, the alarm output generates an interrupt on processor B by way of the second watchdog timer.

The watchdog control program on each processor simply addresses the timer at regular intervals. If the interval passes without the timer being addressed, the timer operates an output relay that generates an interrupt to the second CPU. For this example, assume that the interval is 5 seconds. (Section 18.4.3 contains an example that addresses the problem of a much smaller time interval.)

The watchdog control program on processor A executes as follows:

- 1 Assigns a channel to the device
- 2 Calls the \$CRMPSC system service to map to the I/O page in order to address the device registers
- 3 Issues a connect-to-interrupt \$QIO request to connect the program to the watchdog timer for processor B; specifies the addresses of an interrupt service routine and an AST procedure
- 4 Writes a value to a device register to start the timer
- 5 Calls the \$SETIMR system service to request that an event flag be set after a specified interval (for example, 4 seconds)
- 6 Calls the \$WAITFR system service to wait for the event flag

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.4 Real-Time Applications Examples

- 7 When the event flag is set, writes a value to a device register to reset the timer
- 8 Loops to step 5

The same control program runs on processor B except that it connects to the watchdog timer for processor A. If either processor fails, the watchdog timer generates an interrupt on the other processor.

The standby processor that receives the interrupt gains control in the VMS connect-to-interrupt driver (CONINTERR), which calls a process-supplied interrupt service routine (defined in step 3) that handles the interrupt as follows:

- 1 Sets the KW11-W switch relay register to clear the timer interrupt condition
- 2 Sets a status flag that will cause an AST to be delivered to the control program that connected to the interrupt
- 3 Returns to CONINTERR

CONINTERR completes the interrupt handling as follows:

- 1 Schedules a fork process at a lower IPL (IPL\$_QUEUEAST). This fork process, when it gains control, will queue an AST to the user program.
- 2 Executes an REI instruction to return from the interrupt.

The timer control program on the standby processor regains control in an AST procedure which responds to the other processor's failure by switching over and assuming control of the other processor's tasks (or whatever is appropriate).

18.4.2 **Example 2: AD11-K, AM11-K A/D Converter with Multiplexer Connected to the UNIBUS**

This type of device provides program-driven data collection: it generates an interrupt as the result of a programmed request to the device, and makes the result of the request available as data in a device register. Typical devices include A/D converters and digital I/O registers.

The data collection operation is usually repetitive for such applications. Therefore, the interrupt service routine must be capable of buffering data from the device in order to ensure that no data is lost because of the high-speed data transfer rate. A typical buffer size for this sampling technique might be 32 16-bit words.

In this example, a user program controls an AD11-K/AM11-K combination that accepts analog data from thermocouples. The AD11-K converts analog data to digital data and returns the data in a device register. Every 10 seconds, the program samples 16 to 32 out of 64 channels at gain settings that may vary based on the thermocouple type and previous samplings.

To collect data efficiently, the program buffers data in a process-specified interrupt service routine, and requests delivery of an AST to the user process when all the requested channels have been sampled. To perform variable sampling, the program passes parameters to the interrupt service routine.

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.4 Real-Time Applications Examples

The program establishes a protocol to communicate between the program and the interrupt service routine. The protocol defines a data area shared by the main program, the interrupt service routine, and the AST procedure. The data area contains parameters from the program and data from the AD11-K. The data area is a 98-word array used as follows:

- 1 Elements 1-2 of the data area contain an index to the next buffer location to be filled, and a count indicating the number of samplings still to be taken. The main program initializes these values before starting the device. The interrupt service routine reads and modifies these values in the process of copying data and determining when to stop sampling.
- 2 Elements 3-66 of the data area are reserved for interrupt service routine parameters. Each pair of elements contains the number of a channel and a gain value. The main program loads these parameters before starting the device.
- 3 Elements 67-98 of the data area receive the data that the interrupt service routine reads from the AD11-K data buffer register. The AST routine later reads data from this part of the buffer.

The program sets up for the sampling as follows:

- 1 Assigns a channel to the device
- 2 Calls the \$CRMPSC system service to map to the I/O page in order to address the device registers
- 3 Initializes the data area by writing a 67 (the index to the next buffer location to be filled) into element 1, and the number of samples to take into element 2 of the data area; clears elements 3 through 98 of the data area
- 4 Writes channel numbers and gain values into the parameter section of the data area
- 5 Issues a connect-to-interrupt \$QIO call to connect the process to the A/D converter; specifies the addresses of the area to be double-mapped, an offset to the interrupt service routine, and an AST procedure
- 6 Sets the start and interrupt-enable bits in the AD11-K status register to start the A/D converter
- 7 Calls the \$HIBER system service to place the process in a wait state

As soon as the AD11-K has converted the first sample, the device generates an interrupt. CONINTERR.EXE calls the process-specified interrupt service routine. This process-specified routine executes as follows:

- 1 Computes the next location to be written in the buffer by reading the first element in the data area
- 2 Reads 12 bits of data from the A/D buffer register into the next location in the buffer
- 3 Updates the buffer offset and count elements at the beginning of the data area
- 4 If all requested samples have been collected, writes the address of the data area into the AST parameter, sets a status flag that will cause an AST to be delivered to the control program, and returns to the CONINTERR routine

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.4 Real-Time Applications Examples

- 5 Otherwise, sets the start bit in a device register to restart the device and returns to the CONINTERR routine with a status flag requesting no AST delivery or event flag setting

Based on the interrupt status from the process-specified interrupt service routine, the CONINTERR routine completes the interrupt processing by queuing a fork process that will queue an AST to the user process. When the process gains control in the AST procedure, it processes the samples in the following steps:

- 1 Clears the interrupt-enable bit in the device status register
- 2 Examines the data collected in order to adjust channel selection and/or gain values for the next sampling
- 3 Copies the data to a file
- 4 Reinitializes the data area
- 5 Calls the \$SCHDWK system service to wake the process after a short interval (for example, 10 seconds)
- 6 Returns

When the time interval elapses, the process regains control. The program can then restart the sampling process by again setting the start and interrupt-enable bits in the AD11-K status register.

18.4.3 **Example 3: KW11-P Real-Time Clock and AD11-K Converter Connected to the UNIBUS**

This type of device reports asynchronous external events by collecting data: one device triggers another device by generating an interrupt that causes a programmed request to be sent to the other device, which in turn generates an interrupt. A typical example is a clock-driven A/D operation for precise time sampling as required in signal processing. This processing technique is often used in laboratories. The amount of data collected in such a timed sampling might typically be 200 to 1000 16-bit words.

In this example, the main program sets up the real-time clock to generate interrupts periodically. At regular intervals, the clock interrupt triggers a programmed request for an A/D conversion operation. The AD11-K collects a sample, and interrupts the CPU with a "done" interrupt and 12 bits of data. The AD11-K interrupt service routine buffers the data and, if the buffer is full, causes an AST to be delivered to the process. The process, gaining control in an AST procedure, copies the buffered data to another buffer or to disk.

Programming these device functions is slightly more complicated than the previous example. The main program must specify a large buffer to be used in ring fashion to guarantee that data is not lost between clock-driven samplings. In addition, the program must connect to two device interrupts—one for the clock and one for the A/D converter.

The protocol used by the main program, the interrupt service routine, and the AST procedure is similar to the previous example. The data area is larger: 4K words of buffer area follow the parameter area. The A/D converter interrupt service routine and the AST procedure treat the 4K-word buffer as four buffer sections of 1K words per section. The first element in each 1K buffer section

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.4 Real-Time Applications Examples

is a flag indicating whether the section is in use. The AST resets the flag value after copying the contents of the buffer. The interrupt service routine uses a buffer section only if the section's flag value indicates that the buffer has been emptied.

The main program starts the sampling with the following steps:

- 1 Assigns channels to the clock and to the A/D converter.
- 2 Calls the \$CRMPSC system service to map to the I/O page in order to address the device registers.
- 3 Initializes the data buffer by writing a 67 (the index to the next buffer location to be filled) into element 1, and the number of samples to take into element 2 of the data area; clears elements 3 through 4096 of the data area; flags each page of the buffer as available.
- 4 Writes channel numbers and gain values into the parameter segments of the data area.
- 5 Issues a connect-to-interrupt \$QIO call to connect the process to the clock, and specifies the address of an interrupt service routine.
- 6 Issues a connect-to-interrupt \$QIO call to connect the process to the A/D converter; and specifies the addresses of the area to be double mapped, an offset to the interrupt service routine and an AST procedure.
- 7 Sets the sampling interval by writing a 16-bit value into the KW11-P count set buffer register.
- 8 Starts the clock by setting the run, mode, rate selection, and interrupt-enable bits in the KW11-P control and status register. Setting the mode bit causes repeated interrupts generated at a rate specified in the time interval.
- 9 Calls the \$HIBER system service to place the process in a wait state.

The clock interrupts when zero (underflow) occurs during a countdown from the preset interval count. The VMS CONINTERR routine calls the process-specified clock interrupt service routine. This process-specified routine starts the A/D conversion as follows:

- 1 Starts the A/D converter by setting the start and interrupt-enable bits in the AD11-K status register
- 2 Sets interrupt status that prevents AST delivery or event flag setting as a result of this interrupt
- 3 Returns to CONINTERR

Starting the A/D converter results in an interrupt from the AD11-K, and control passes, by way of CONINTERR, to the AD11-K interrupt service routine. This routine executes as follows:

- 1 If this sample is the first sample for a new buffer (indicated by a flag in the data area), the routine moves to the next buffer section (branching to error handling if the buffer is still full), and sets up the first two elements of the data area to indicate the buffer section to be written next. Then it sets the flag at the start of the new buffer section and sets a flag in the data area to indicate that sampling is occurring.
- 2 The routine computes the next location to be written in the buffer by reading the first location in the data area.

Mapping to I/O Space and the Connect-to-Interrupt Facility

18.4 Real-Time Applications Examples

- 3 The routine reads 12 bits of data from the A/D buffer register into the next location in the buffer.
- 4 The routine updates the buffer offset and count values in the data area.
- 5 If this sample fills the data sector, the routine writes the offset of the filled sector from the start of the 4K-word buffer into the AST parameter, sets a status flag that will cause an AST to be delivered to the control program, and sets a flag indicating that a new data section is to be started.
- 6 The routine returns to CONINTERR.

The AST procedure copies and fills the next buffer section with zeros to indicate that the section is again available to the interrupt service routine. When the next clock interrupt occurs, the data can be written to the next buffer section, even if the AST routine has not yet emptied the previous buffer section.

Part IV Reference Section and Examples

A

Data Structures

This appendix provides a condensed description of those data structures referenced by driver code. It lists their fields in the order in which they appear in the structures. All data structures discussed in this appendix—with the exception of the channel control block (CCB)—exist in nonpaged system memory.

Many of these structures—including the adapter control block (ADP), channel control block (CCB), channel request block (CRB), configuration control block (ACF), device data block (DDB), driver dispatch table (DDT), driver prologue table (DPT), object rights block (ORB), I/O request packet (IRP), I/O request packet extension (IRPE), and unit control block (UCB)—are collectively known as the I/O database (see Figure A-1). The structures in the *I/O database* help the VMS operating system and device drivers monitor the status of, and control the functions of, the I/O subsystem. They provide the following types of information:

- Descriptions of each pending and in-progress I/O request
- Characteristics of each device type
- Number and type of each device unit
- Status of current activity on each device unit
- External entry points to all device drivers
- Entry points for controller and device unit initialization routines
- Code that dispatches interrupts to the appropriate servicing routines
- Addresses of device registers
- Bit maps describing the allocation of data paths and map registers

Aside from the I/O database structures, this appendix includes descriptions of those data structures VMS uses to maintain multiprocessing synchronization and record processor-specific information: the spin lock data structure (SPL) and the per-CPU database structure (CPU), respectively.

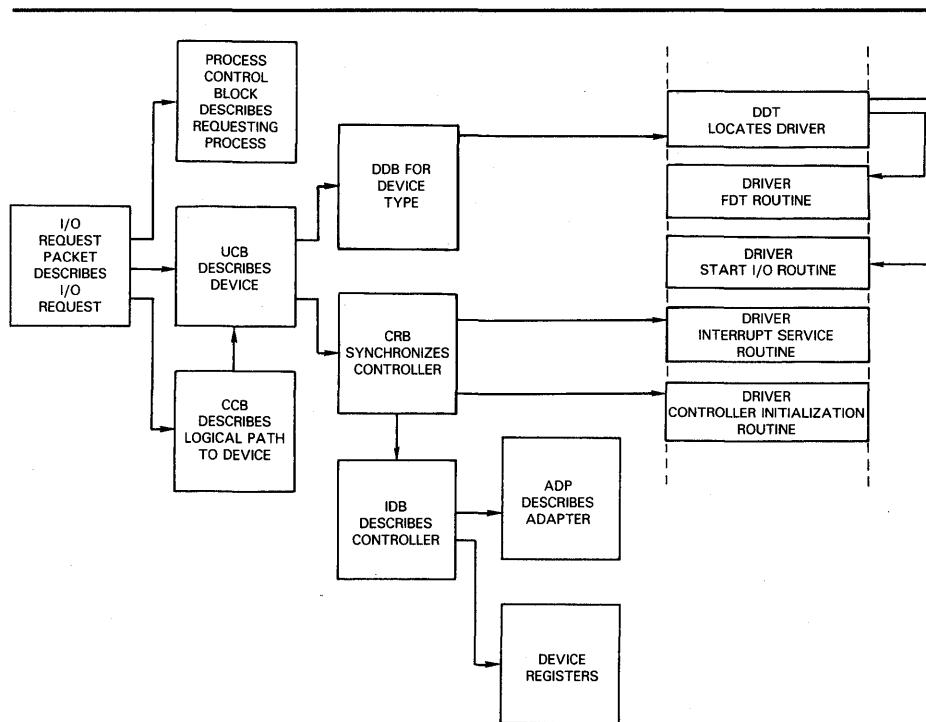
Notes: Driver code must consider fields marked by asterisks to be read-only fields. Fields marked “reserved” or “unused” are reserved for future use by DIGITAL unless otherwise specified.

When referring to locations within a data structure, a driver should use symbolic offsets from the beginning of the structure and *not* numeric offsets. Numeric offsets are likely to change with each new release of the VMS operating system. The figures in this appendix list VMS Version 5.0 numeric offsets to aid in driver debugging.

Data Structures

A.1 Configuration Control Block (ACF)

Figure A-1 The I/O Database



ZK-1766-84

A.1 Configuration Control Block (ACF)

The configuration control block (ACF) is used by the SYSGEN autoconfiguration facility to describe the device it is adding to the system. Device drivers can gain access to this data structure only if they have specified a unit delivery routine in the DPT and only when that routine is executing. Under certain conditions, the information stored in the ACF might be useful to a unit delivery routine.

The fields described in the configuration control block are illustrated in Figure A-2 and described in Table A-1.

Data Structures

A.1 Configuration Control Block (ACF)

Figure A-2 Configuration Control Block (ACF)

| | | | | |
|------------------|----------------|-----------------|------------------|----|
| ACF\$_ADAPTER* | | | | 00 |
| ACF\$_CONFIGREG* | | | | 04 |
| ACF\$_AFLAG* | ACF\$_AUNIT* | ACF\$_AVECTOR* | | 08 |
| ACF\$_CONTRLREG* | | | | 12 |
| ACF\$_CUNIT* | | ACF\$_CVECTOR* | | 16 |
| ACF\$_DEVNAME* | | | | 20 |
| ACF\$_DRVNAME* | | | | 24 |
| ACF\$_COMBO_VEC* | ACF\$_CNUMVEC* | ACF\$_MAXUNITS* | | 28 |
| unused | | ACF\$_NUMUNIT* | ACF\$_COMBO_CSR* | 32 |
| ACF\$_DLVR_SCRH | | | | 36 |

ZK-6626-HC

Data Structures

A.1 Configuration Control Block (ACF)

Table A-1 Contents of the Configuration Control Block

| Field Name | Contents |
|------------------|---|
| ACF\$_ADAPTER* | Address of ADP for adapter currently being configured. |
| ACF\$_CONFIGREG* | Address of configuration register for adapter currently being configured. |
| ACF\$_AVECTOR* | Offset from base of SCB to interrupt vector of adapter currently being configured. |
| ACF\$_AUNIT* | Adapter unit number of device or controller currently being configured. |
| ACF\$_AFLAG* | Flags associated with autoconfiguration operation. Flags defined in this field include the following: ACF\$_RELOAD Reloading driver code. ACF\$_CRBBLT CRB and IDB already built for device. ACF\$_SCBVEC CVECTOR is offset into SCB. ACF\$_NOLOAD_DB Do not load I/O database, only load driver. ACF\$_SUPPORT VMS-supported device. ACF\$_GETDONE Addresses of data structures in I/O database have been obtained. ACF\$_BVP Multiport BVP adapter. |
| ACF\$_CONTRLREG* | Address of CSR for controller currently being configured. |
| ACF\$_CVECTOR* | Offset into ADP vector table to longword that contains transfer address of interrupt vector used by controller currently being configured (if ACF\$_SCBVEC is not set). If ACF\$_SCBVEC is set, this field is the offset from the SCB base to the interrupt vector of the controller currently being configured. |
| ACF\$_CUNIT* | Unit number of device currently being configured. |
| ACF\$_DEVNAME* | Address of counted ASCII string that gives name of controller currently being configured. |
| ACF\$_DRVNAME* | Address of counted ASCII string that gives driver name for controller currently being configured. |
| ACF\$_MAXUNITS* | Maximum number of units that can be connected to controller currently being configured. |
| ACF\$_CNUMVEC* | Number of interrupt vectors to configure for controller currently being configured. |
| ACF\$_COMBO_VEC* | Offset to vectors for combo device. (The name of this field is ACF\$_COMBO_VECTOR_OFFSET.) |
| ACF\$_COMBO_CSR* | Offset to start of control registers of combo device. (The name of this field is ACF\$_COMBO_CSR_OFFSET.) |
| ACF\$_NUMUNIT* | Number of units to be configured for controller currently being configured. |
| ACF\$_DLVR_SCRH | Field available for use by unit delivery routine. SYSGEN never alters this field. |

A.2 Adapter Control Block (ADP)

Each MASSBUS adapter, UNIBUS adapter, Q22 bus, and VAXBI node configured in a VAX system is represented to VMS and driver routines by an adapter control block (ADP). The ADP stores adapter-specific static and dynamic data such as the adapter CSR address and map-register wait queues.

Data Structures

A.2 Adapter Control Block (ADP)

The ADP varies in size, depending upon what type of adapter it describes. Table A-2 defines those fields that generally appear in an ADP.

Figure A-3 Adapter Control Block (ADP)

| | | | |
|-------------------------------|--------------|-----------------------|-----|
| ADP\$L_CSR* | | | 00 |
| ADP\$L_LINK* | | | 04 |
| ADP\$B_NUMBER* | ADP\$B_TYPE* | ADP\$W_SIZE* | 08 |
| ADP\$W_ADPTYPE* | | ADP\$W_TR* | 12 |
| ADP\$L_VECTOR* | | | 16 |
| ADP\$L_DPOFL* | | | 20 |
| ADP\$L_DPQBL* | | | 24 |
| ADP\$L_AVECTOR* | | | 28 |
| ADP\$L_BI_IDR* | | | 32 |
| unused | | | 36 |
| ADP\$W_BI_VECTOR* | | ADP\$W_BI_FLAGS* | 40 |
| ADP\$L_SCB_PAGE* | | | 44 |
| ADP\$L_BIMASTER* | | | 48 |
| ADP\$B_ADDR_BITS* | unused | ADP\$W_ADPDISP_FLAGS* | 52 |
| reserved | | | 56 |
| ADP\$L_MRQFL* | | | 60 |
| ADP\$L_MRQBL* | | | 64 |
| ADP\$L_INTD* (12 bytes) | | | 68 |
| ADP\$L_UBASCB* (16 bytes) | | | 80 |
| ADP\$L_UBASPTE* | | | 96 |
| ADP\$L_MRACTMDRS* | | | 100 |
| ADP\$W_MRNFFENCE* | | ADP\$W_DPBITMAP* | 104 |
| ADP\$W_MRNREGARY* (248 bytes) | | | 108 |
| | | ADP\$W_MRFFENCE* | 356 |
| ADP\$W_MRFREGARY* (248 bytes) | | | |
| ADP\$W_UMR_DIS* | | | |
| ADP\$L_MR2QFL* | | | 608 |
| ADP\$L_MR2QBL* | | | 612 |
| ADP\$L_MR2ACTMDR* | | | 616 |
| ADP\$W_MR2NFENCE* | | unused | 620 |
| ADP\$W_MR2NREGAR* (248 bytes) | | | 624 |

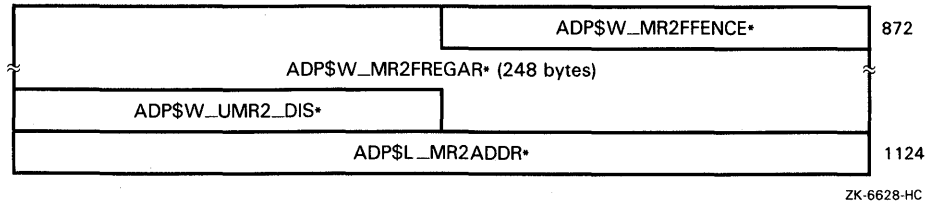
ZK-6627-HC

Figure A-3 Cont'd. on next page

Data Structures

A.2 Adapter Control Block (ADP)

Figure A-3 (Cont.) Adapter Control Block (ADP)



ZK-6628-HC

Table A-2 Contents of Adapter Control Block

| Field Name | Contents |
|-----------------|---|
| ADP\$L_CSR* | Virtual address of adapter configuration register. For a generic VAXBI adapter, this field contains the address of the base of the adapter's node space. The VMS adapter initialization routine writes this field. The configuration register marks the base of adapter register space, an area that contains data path registers, map registers, or any other registers appropriate to the implementation of the adapter. |
| ADP\$L_LINK* | Address of next ADP. The VMS adapter initialization routine writes this field. A value of 0 indicates that this is the last ADP. |
| ADP\$W_SIZE* | Size of ADP. The VMS adapter initialization routine writes this field when the routine creates the ADP. For non-direct-vector UNIBUS adapters, ADP\$W_SIZE includes the space allocated for the four UNIBUS interrupt service routines (for BR4 to BR7) and the vector jump table. |
| ADP\$B_TYPE* | Type of data structure. The VMS adapter initialization routine writes the symbolic constant DYN\$C_ADP into this field when the routine creates the ADP. |
| ADP\$B_NUMBER* | Number of this type of adapter (for example, the number for a third MASSBUS adapter is 2). The VMS adapter initialization routine writes this field when the routine creates the ADP. |
| ADP\$W_TR* | Nexus number of adapter. The VMS adapter initialization routine writes this field when the routine creates the ADP. The driver-loading procedure compares the nexus number specified in a CONNECT command with this field of each ADP in the system to determine to which adapter a device is attached. For a generic VAXBI adapter, this field contains its VAXBI node ID. |
| ADP\$W_ADPTYPE* | Type of adapter. The VMS adapter initialization routine writes the symbolic constant AT\$_UBA into this field when the routine creates an ADP for a UNIBUS adapter or Q22 bus; AT\$_MBA for a MASSBUS adapter; and AT\$_GENBI for a generic VAXBI adapter. |

Data Structures

A.2 Adapter Control Block (ADP)

Table A-2 (Cont.) Contents of Adapter Control Block

| Field Name | Contents |
|----------------|--|
| ADP\$_VECTOR* | <p>Address of adapter dispatch table. The table is 512 bytes of longword vectors that correspond to device interrupt vectors (0_8-777_8).</p> <p>On VAX processors that handle direct-vector interrupts, ADP\$_VECTOR points to the second (or subsequent) page of the SCB. The CPU uses this page when it dispatches the device interrupt to the driver interrupt service routine. Each vector entry that corresponds to a vector in use contains the address of the controller's interrupt dispatcher (CRB\$_INTD). (The actual stored value is CRB\$_INTD+1, the set low bit of the address indicating that the interrupt stack is to be used in servicing interrupts.)</p> <p>On VAX processors that handle non-direct-vector interrupts, ADP\$_VECTOR points to a page allocated from nonpaged pool called the adapter dispatch table (or vector jump table). Each longword in the page that corresponds to a vector in use contains the address of the controller's interrupt dispatcher (CRB\$_INTD+2). When the UNIBUS adapter interrupts on behalf of a UNIBUS device, the UNIBUS adapter interrupt service routine saves R0 through R5, determines the vector address of the interrupting device, indexes into the vector-jump table, and jumps to the instruction at CRB\$_INTD+2.</p> <p>For both types of VAX processor, adapter dispatch table entries that correspond to unused vectors contain the address of the adapter's unexpected-interrupt service routine.</p> |
| ADP\$_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> <p>This field is also known as ADP\$_MBASCB. For MASSBUS adapters and generic VAXBI adapters, the VMS adapter initialization routine stores the address of the adapter's interrupt vector in this field. Certain power failure recovery operations use the contents of ADP\$_MBASCB to refresh the SCB vectors. The actual stored value is CRB\$_INTD+1, the set low bit of the address indicating that the interrupt stack is to be used in servicing interrupts.</p> |
| ADP\$_DPQBL* | <p>Data path wait queue backward link. IOC\$REQDATAP and IOC\$RELDATAP read and write this field.</p> <p>This field is also known as ADP\$_MBASPTE. For generic VAXBI adapters, the VMS adapter initialization routine stores here the contents of the first of 16 SPTes that map the adapter's node space. For the MASSBUS adapter, the routine stores here the SPTe value that maps MBA address space. Certain recovery operations use the contents of ADP\$_MBASPTE to restore SPTe values and remap node space following a power failure.</p> |
| ADP\$_AVECTOR* | <p>Address of first SCB vector for adapter.</p> |

Data Structures

A.2 Adapter Control Block (ADP)

Table A-2 (Cont.) Contents of Adapter Control Block

| Field Name | Contents | | | | | | | | | | | | | | | | | | | | |
|------------------------|---|---------------------|-------------------------------------|---------------------|---------------------------|----------------------|--------------------------|---------------------|----------------------|--------------------|-----------------------------|---------------------|---|---------------------|---|------------------------|---|-------------|-----------------|--------|---------------------|
| ADP\$L_BI_IDR* | Longword mask specifying, by a single set bit, which VAXBI node is the destination of interrupts from this adapter. In VAX 8200/8250/8300/8350 systems, the VAXBI node of the primary processor becomes the destination for interrupts; in VAX 8530/8550/8700/8800/8830/8840 and VAX 6200-series systems, it is the VAXBI node at which the memory-interconnect-to-VAXBI adapter (NBIB, PBIB, or DWMBA/B) resides. | | | | | | | | | | | | | | | | | | | | |
| ADP\$W_BI_FLAGS* | VAXBI device flags field. | | | | | | | | | | | | | | | | | | | | |
| ADP\$W_BI_VECTOR* | Offset of the first interrupt vector for this VAXBI node from the start of its SCB page. ADP\$L_AVECTOR contains the address of this vector. | | | | | | | | | | | | | | | | | | | | |
| ADP\$L_SCB_PAGE* | Offset to SCB page for this VAXBI device. | | | | | | | | | | | | | | | | | | | | |
| ADP\$L_BIMASTER* | Address of the ADP of the master device of the VAXBI (for example, the DWMBA in a VAX 6200-series system). | | | | | | | | | | | | | | | | | | | | |
| ADP\$W_ADPDISP_FLAGS* | Flags used by the ADPDISP macro to control branching according to adapter characteristics. The following bit fields are defined within ADP\$W_ADPDISP_FLAGS: <table border="0" style="margin-left: 20px;"> <tr> <td>ADP\$V_ADPDISP_INIT</td> <td>ADPDISP flags have been initialized</td> </tr> <tr> <td>ADP\$V_ADAP_MAPPING</td> <td>Adapter mapping supported</td> </tr> <tr> <td>ADP\$V_DIRECT_VECTOR</td> <td>Direct-vector interrupts</td> </tr> <tr> <td>ADP\$V_AUTOPURGE_DP</td> <td>Autopurging datapath</td> </tr> <tr> <td>ADP\$V_BUFFERED_DP</td> <td>Buffered datapath supported</td> </tr> <tr> <td>ADP\$V_ODD_XFER_BDP</td> <td>Odd transfers supported on buffered data path</td> </tr> <tr> <td>ADP\$V_ODD_XFER_DDP</td> <td>Odd transfers supported on direct data path</td> </tr> <tr> <td>ADP\$V_EXTENDED_MAPREG</td> <td>Alternate map registers (registers 496 to 8191) supported</td> </tr> <tr> <td>ADP\$V_QBUS</td> <td>Q22 bus adapter</td> </tr> <tr> <td><15:9></td> <td>Reserved to DIGITAL</td> </tr> </table> | ADP\$V_ADPDISP_INIT | ADPDISP flags have been initialized | ADP\$V_ADAP_MAPPING | Adapter mapping supported | ADP\$V_DIRECT_VECTOR | Direct-vector interrupts | ADP\$V_AUTOPURGE_DP | Autopurging datapath | ADP\$V_BUFFERED_DP | Buffered datapath supported | ADP\$V_ODD_XFER_BDP | Odd transfers supported on buffered data path | ADP\$V_ODD_XFER_DDP | Odd transfers supported on direct data path | ADP\$V_EXTENDED_MAPREG | Alternate map registers (registers 496 to 8191) supported | ADP\$V_QBUS | Q22 bus adapter | <15:9> | Reserved to DIGITAL |
| ADP\$V_ADPDISP_INIT | ADPDISP flags have been initialized | | | | | | | | | | | | | | | | | | | | |
| ADP\$V_ADAP_MAPPING | Adapter mapping supported | | | | | | | | | | | | | | | | | | | | |
| ADP\$V_DIRECT_VECTOR | Direct-vector interrupts | | | | | | | | | | | | | | | | | | | | |
| ADP\$V_AUTOPURGE_DP | Autopurging datapath | | | | | | | | | | | | | | | | | | | | |
| ADP\$V_BUFFERED_DP | Buffered datapath supported | | | | | | | | | | | | | | | | | | | | |
| ADP\$V_ODD_XFER_BDP | Odd transfers supported on buffered data path | | | | | | | | | | | | | | | | | | | | |
| ADP\$V_ODD_XFER_DDP | Odd transfers supported on direct data path | | | | | | | | | | | | | | | | | | | | |
| ADP\$V_EXTENDED_MAPREG | Alternate map registers (registers 496 to 8191) supported | | | | | | | | | | | | | | | | | | | | |
| ADP\$V_QBUS | Q22 bus adapter | | | | | | | | | | | | | | | | | | | | |
| <15:9> | Reserved to DIGITAL | | | | | | | | | | | | | | | | | | | | |
| ADP\$B_ADDR_BITS* | Number of adapter address bits. This field contains the value 22 (for Q22 bus systems) and 18 (for UNIBUS adapters). | | | | | | | | | | | | | | | | | | | | |
| ADP\$L_MRQFL* | Standard-map-register wait queue's forward link. IOC\$ALOUBAMAP, IOC\$REQMAPREG, and IOC\$RELMAPREG read and write these fields. When a driver fork process requests a set of standard 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 standard-map-register wait queue, and suspends the driver fork process. When another driver calls IOC\$RELMAPREG to release a set of standard map registers, the routine dequeues a UCB fork block address from the standard-map-register wait queue, allocates the requested set of map registers to the driver, and reactivates that driver fork process. | | | | | | | | | | | | | | | | | | | | |
| ADP\$L_MRQBL* | Standard-map-register wait queue's backward link. IOC\$ALOUBAMAP, IOC\$REQMAPREG, and IOC\$RELMAPREG read and write this field. | | | | | | | | | | | | | | | | | | | | |
| ADP\$L_INTD* | Interrupt transfer vector. The VMS adapter initialization routine places executable code in this field to allow certain DIGITAL-supplied adapters or controllers to dispatch to adapter-specific interrupt and error handling routines. | | | | | | | | | | | | | | | | | | | | |

Data Structures

A.2 Adapter Control Block (ADP)

Table A-2 (Cont.) Contents of Adapter Control Block

| Field Name | Contents |
|-------------------|---|
| ADP\$_UBASCB* | Series of four longwords that contain SCB entry values, one for each bus request (BR) level or interrupt vector. The UNIBUS adapter power failure recovery procedure uses these values. |
| ADP\$_UBASPT* | System page-table entry (PTE) values for base of UNIBUS adapter register space and base of UNIBUS I/O register space. These values contained in this quadword field are used during UNIBUS adapter power failure recovery. |
| ADP\$_MRACTMDRS* | Number of active standard map register descriptors in arrays to which ADP\$W_MRNREGARY and ADP\$W_MRFREGARY point. IOC\$REQMAPREG and IOC\$RELMAPREG use these fields when allocating and deallocating standard map registers. |
| ADP\$W_DPBITMAP* | Data path allocation bit map. IOC\$REQDATAP and IOC\$RELDATAP read and write this field. The VMS adapter initialization routine sets the bit map to show as available all the buffered data paths supported by the UNIBUS adapter. (The adapter initialization routine for certain VAX processors whose UNIBUS adapters or Q22 bus interfaces do not supply buffered data paths marks three data paths as available. This facilitates the writing of machine-independent code that can execute regardless of the presence of buffered data paths.) The state of each of the available buffered data paths (whether in use or available) is recorded in the data path allocation bit map. One data path corresponds to each bit in the field. If a bit is clear, the related data path is currently allocated to a driver fork process. |
| ADP\$W_MRNFENCE* | Boundary marker for the array specified by ADP\$W_MRNREGARY; contains -1. |
| ADP\$W_MRNREGARY* | Standard map register "number of registers" array of 124 words. The number of words, or cells, that are active in this array is contained in ADP\$_MRACTMDRS. Each active cell gives the number of free standard map registers. For each active cell in this array, there is a corresponding first free map register number in the "first register" array (ADP\$W_MRFREGARY). Together, these values give the base map register and number of free map registers for a block of free map registers. This information is used to allocate and deallocate standard map registers. |
| ADP\$W_MRFFENCE* | Boundary marker for array specified by ADP\$W_MRFREGARY; contains -1. |
| ADP\$W_MRFREGARY* | Standard map register "first register" array of 124 words. The number of currently active cells in this array is contained in ADP\$_MRACTMDRS. Each active cell gives a number of the first free map register within a block of free map registers. For each active cell in this array, there is a corresponding cell in the "number of registers" array (ADP\$W_MRNREGARY) that gives a number of free map registers. Together, these values give the base map register and number of free map registers for a block of free map registers. This information is used to allocate and deallocate standard map registers. |
| ADP\$W_UMR_DIS* | Number of disabled standard map registers. During system initialization, some standard map registers can be disabled so that their corresponding UNIBUS and Q22 bus addresses can be accessed directly through UNIBUS-space or Q22-bus-space physical addresses. |

Data Structures

A.2 Adapter Control Block (ADP)

Table A–2 (Cont.) Contents of Adapter Control Block

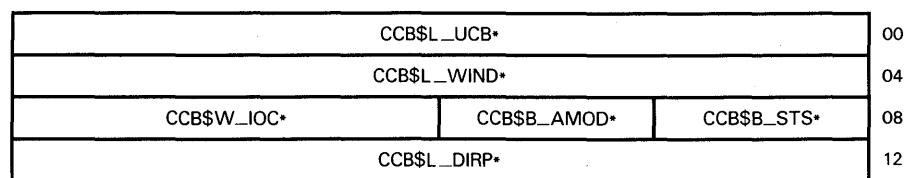
| Field Name | Contents |
|--------------------|---|
| ADP\$\$_MR2QFL* | <p>Alternate-map-register wait queue's forward link. IOC\$ALOALTMAP, IOC\$REQALTMAP, and IOC\$RELALTMAP read and write this field. When a driver fork process requests a set of Q22 bus alternate map registers and the set is not currently available, IOC\$REQALTMAP saves driver context in the device's UCB fork block, inserts the fork block address in the alternate-map-register wait queue, and suspends the driver fork process.</p> <p>When another driver calls IOC\$RELALTMAP to release a sufficient number of map registers, the routine dequeues a UCB fork block from the alternate-map-register wait queue, allocates the requested set of map registers to the driver, and reactivates that driver fork process.</p> |
| ADP\$\$_MR2QBL* | <p>Alternate-map-register wait queue's backward link. IOC\$ALOALTMAP, IOC\$REQALTMAP, and IOC\$RELALTMAP read and write this field when allocating and deallocating from the set of Q22 bus alternate map registers.</p> |
| ADP\$\$_MR2ACTMDR* | <p>Number of active map register descriptors in arrays to which ADP\$\$_MR2NREGAR and ADP\$\$_MR2FREGAR point. IOC\$ALOALTMAP, IOC\$REQALTMAP, and IOC\$RELALTMAP use these fields when allocating and deallocating Q22 bus alternate map registers.</p> |
| ADP\$\$_MR2NFENCE* | <p>Boundary marker for the array specified by ADP\$\$_MR2NREGAR; contains –1.</p> |
| ADP\$\$_MR2NREGAR* | <p>Alternate-map-register "number of registers" array of 124 words. The number of words, or cells, that are active in this array is contained in ADP\$\$_MR2ACTMDR. Each active cell gives a number of map registers in a block of free alternate map registers. For each active cell in this array, there is a corresponding first free map register number in the array specified by ADP\$\$_MR2FREGAR. Together, these values give the base map register and the number of free map registers for a block of free alternate map registers. IOC\$ALOALTMAP, IOC\$REQALTMAP, and IOC\$RELALTMAP use this information when allocating and deallocating from Q22 bus alternate map registers.</p> |
| ADP\$\$_MR2FFENCE* | <p>Boundary marker for the array specified by ADP\$\$_MR2NREGAR; contains –1.</p> |
| ADP\$\$_MR2FREGAR* | <p>Alternate map register "first register" array of 124 words. The number of words, or cells, that are active in this array is contained in ADP\$\$_MR2ACTMDR. Each active cell gives the number of the first free map register within a block of free map registers. For each active cell in this array, there is a corresponding cell in the "number of registers" array, ADP\$\$_MR2NREGAR. Together, these values give the base map register and the number of free map registers for a block of free map registers.</p> |
| ADP\$\$_UMR2_DIS* | <p>Number of disabled Q22 bus alternate map registers. During system initialization, some map registers can be disabled so that their corresponding Q22 bus addresses can be accessed directly through physical addresses.</p> |
| ADP\$\$_MR2ADDR | <p>Address of the first Q22 bus alternate map register mapped in CPU node private space. The value varies for each processor with alternate map registers. IOC\$LOADUBAMAP reads this field when accessing alternate map registers.</p> |

A.3 Channel Control Block (CCB)

When a process assigns an I/O channel to a device unit with the \$ASSIGN system service, EXE\$ASSIGN locates a free block among the process's preallocated channel control blocks (CCBs). EXE\$ASSIGN then writes into the CCB a description of the device attached to the CCB's channel.

The channel control block is the only data structure described in this appendix that exists in the control (P1) region of perprocess address space. It is illustrated in Figure A-4 and described in Table A-3.

Figure A-4 Channel Control Block (CCB)



ZK-6629-HC

Table A-3 Contents of Channel Control Block

| Field Name | Contents |
|-------------|--|
| CCB\$_UCB* | Address of UCB of assigned device unit. EXE\$ASSIGN writes a value into this field. EXE\$QIO reads this field to determine that the I/O request specifies a process I/O channel assigned to a device and to obtain the device's UCB address. |
| CCB\$_WIND* | Address of window control block (WCB) for file-structured device assignment. This field is written by an ACP or XQP and read by EXE\$QIO. A file-structured device's XQP or ACP creates a WCB when a process accesses a file on a device assigned to a process I/O channel. The WCB maps the virtual block numbers of the file to a series of physical locations on the device. |
| CCB\$_STS* | Channel status. |
| CCB\$_AMOD* | Access mode plus 1 of the channel. EXE\$ASSIGN writes the access mode value into this field. |
| CCB\$_IOC* | Number of outstanding I/O requests on channel. EXE\$QIO increases this field when it begins to process an I/O request that specifies the channel. During I/O postprocessing, the special kernel-mode AST routine decrements this field. Some FDT routines and EXE\$DASSGN read this field. |
| CCB\$_DIRP* | Address of IRP for requested deaccess. A number of outstanding I/O requests can be pending on the same process I/O channel at one time. If the process that owns the channel issues an I/O request to deaccess the device, EXE\$QIO holds the deaccess request until all other outstanding I/O requests are processed. |

Data Structures

A.4 Per-CPU Database (CPU)

A.4 Per-CPU Database (CPU)

A per-CPU database structure exists for each processor in a VMS multiprocessing environment. The per-CPU database records processor-specific information such as the current process control block (PCB), the priority of the current process, and the physical processor identifier. It points to the processor's interrupt stack and contains the list heads for the processor's fork queues and I/O postprocessing queue.

To ensure that the path of a processor's activity at booting and on the interrupt stack remains independent of the paths of other active processors in the system, VMS places a separate boot stack and a separate interrupt stack (formerly pointed to by EXE\$GL_INTSTK) adjacent to the area allocated for the per-CPU database structure. The processor's boot stack, interrupt stack, and per-CPU database fields are virtually contiguous in system address space, although three no-access guard pages prevent the expansion of the stacks beyond the areas reserved for their use. Offset CPU\$L_INTSTK in the per-CPU database points to the interrupt stack.

The fields described in the per-CPU database are illustrated in Figure A-5 and described in Table A-4.

Data Structures

A.4 Per-CPU Database (CPU)

Figure A-5 Per-CPU Database (CPU)

| | | | | |
|----------------------------|--------------------|--------------------|----------|-----|
| CPU\$L_CURPCB* | | | | 00 |
| CPU\$L_REALSTACK* | | | | 04 |
| CPU\$B_SUBTYPE* | CPU\$B_TYPE* | CPU\$W_SIZE* | | 08 |
| CPU\$B_CUR_PRI* | CPU\$B_CPUMTX* | CPU\$B_STATE* | reserved | 12 |
| CPU\$L_INTSTK* | | | | 16 |
| CPU\$L_WORK_REQ* | | | | 20 |
| CPU\$L_PERCPUVA* | | | | 24 |
| CPU\$L_SAVED_AP* | | | | 28 |
| CPU\$L_HALTPC* | | | | 32 |
| CPU\$L_HALTPSL* | | | | 36 |
| CPU\$L_SAVED_ISP* | | | | 40 |
| CPU\$L_PCBB* | | | | 44 |
| CPU\$L_SCBB* | | | | 48 |
| CPU\$L_SISR* | | | | 52 |
| CPU\$L_POBR* | | | | 56 |
| CPU\$L_POLR* | | | | 60 |
| CPU\$L_P1BR* | | | | 64 |
| CPU\$L_P1LR* | | | | 68 |
| CPU\$L_BUGCODE* | | | | 72 |
| CPU\$B_CPUDATA* (32 bytes) | | | | 76 |
| CPU\$L_MCHK_MASK* | | | | 108 |
| CPU\$L_MCHK_SP* | | | | 112 |
| CPU\$L_POPT_PAGE* | | | | 116 |
| reserved (408 bytes) | | | | 120 |
| CPU\$Q_SWIQFL* (48 bytes) | | | | 528 |
| CPU\$L_PSFL* | | | | 576 |
| CPU\$L_PSBL* | | | | 580 |
| CPU\$Q_WORK_FQFL* | | | | 584 |
| CPU\$L_QLOST_FQFL* | | | | 592 |
| CPU\$L_QLOST_FQBL* | | | | 596 |
| CPU\$B_QLOST_FLCK* | CPU\$B_QLOST_TYPE* | CPU\$W_QLOST_SIZE* | | 600 |

ZK-6630-HC

Figure A-5 Cont'd. on next page

Data Structures

A.4 Per-CPU Database (CPU)

Figure A-5 (Cont.) Per-CPU Database (CPU)

| | |
|------------------------------|-----|
| CPUSL_QLOST_FPC* | 604 |
| CPUSL_QLOST_FR3* | 608 |
| CPUSL_QLOST_FR4* | 612 |
| CPU\$Q_BOOT_TIME* | 616 |
| CPU\$Q_CPUID_MASK* | 624 |
| CPUSL_PHY_CPUID* | 632 |
| CPUSL_CAPABILITY* | 636 |
| CPUSL_TENUSEC* | 640 |
| CPUSL_UBDELAY* | 644 |
| CPUSL_KERNEL* (28 bytes) | 648 |
| CPUSL_NULLCPU* | 676 |
| CPUSW_UKERNEL* (14 bytes) | 680 |
| CPUSW_UNULLCPU* | 696 |
| CPUSW_HARDAFF* | |
| CPUSW_CLKUTICS* | 700 |
| CPUSL_RANK_VEC* | |
| CPUSL_IPL_VEC* | 704 |
| CPUSL_IPL_ARRAY* (128 bytes) | 708 |
| CPUSL_TPOINTER* | 836 |
| CPUSW_SANITY_TICKS* | 840 |
| CPUSW_SANITY_TIMER* | |

ZK-6631-HC

Table A-4 Per-CPU Database (CPU)

| Field | Contents |
|------------------|--|
| CPUSL_CURPCB* | Address of current PCB. The scheduler writes this field. |
| CPUSL_REALSTACK* | Physical address of boot stack. |
| CPUSW_SIZE* | Size of the per-CPU database, including the size of the boot stack but not the interrupt stack or the interrupt stack's guard pages. |
| CPU\$B_TYPE* | Type of data structure. VMS writes the value DYN\$C_MP into this field when it creates the per-CPU database. |
| CPU\$B_SUBTYPE* | Structure subtype. VMS writes the value DYN\$C_MP_CPU into this field when it creates the per-CPU database. |

Data Structures

A.4 Per-CPU Database (CPU)

Table A-4 (Cont.) Per-CPU Database (CPU)

| Field | Contents |
|------------------|--|
| CPU\$B_STATE* | State of this processor. VMS defines the following processor states: CPU\$C_INIT Processor is being initialized. CPU\$C_RUN Processor is running. CPU\$C_STOPPING Processor is stopping. CPU\$C_STOPPED Processor is stopped. CPU\$C_TIMEOUT Logical console has timed out. CPU\$C_BOOT_REJECTED Processor has refused to join multiprocessing system. CPU\$C_BOOTED Processor has booted, but is waiting to join multiprocessing active set. |
| CPU\$B_CPUMTX* | Count of acquisitions of CPUMTX mutex. |
| CPU\$B_CUR_PRI* | Current process priority. The scheduler writes this field. |
| CPU\$L_INTSTK* | Address of initial interrupt stack. |
| CPU\$L_WORK_REQ* | Work request bits. A processor sets one or more of these bits in another processor's per-CPU database when directing an interprocessor interrupt to that processor. The following fields are defined within CPU\$L_WORK_REQ: CPU\$V_INV_TBS Request to invalidate single address (SMP\$GL_INVALID) in translation buffer CPU\$V_INV_TBA Request to invalidate all addresses in translation buffer CPU\$V_TBACK Acknowledgment that a processor requested to invalidate its translation buffer has done so CPU\$V_BUGCHK Request to bugcheck CPU\$V_BUGCHKACK Acknowledgment that the processor has saved process context and per-CPU data so that the crash CPU can continue to perform a bugcheck CPU\$V_RECALSCHD Recalculate per-CPU mask and reschedule CPU\$V_UPDASTLVL Request to update processor AST level register (PR\$_ASTLVL) CPU\$V_UPDTODR Request to update processor time-of-day register (PR\$_TODR) CPU\$V_WORK_FQP Request to process internal fork queue (CPU\$Q_WORK_IFQ) CPU\$V_QLOST Request to stall until quorum regained CPU\$V_RESCHED Request to initiate software interrupt at IPL 3 CPU\$V_VIRTCONS Request to enter virtual console mode <28:31> Processor-specific work request bits |
| CPU\$L_PERCPUVA* | Virtual address of this per-CPU database structure. |
| CPU\$L_SAVED_AP* | Halt restart code. |
| CPU\$L_HALTPC* | Halt PC for restart. |
| CPU\$L_HALTPSL* | Halt PSL for restart. |

Data Structures

A.4 Per-CPU Database (CPU)

Table A-4 (Cont.) Per-CPU Database (CPU)

| Field | Contents |
|--------------------|--|
| CPU\$_SAVED_ISP* | Saved ISP for restart. |
| CPU\$_PCBB* | PCBB from power down. |
| CPU\$_SCBB* | SCBB from power down. |
| CPU\$_SISR* | SISR from power down. |
| CPU\$_POBR* | P0 base register (used by system power failure and bugcheck routines). |
| CPU\$_POLR* | P0 length register (used by system power failure and bugcheck routines). |
| CPU\$_P1BR* | P1 base register (used by system power failure and bugcheck routines). |
| CPU\$_P1LR* | P1 length register (used by system power failure and bugcheck routines). |
| CPU\$_BUGCODE* | Bugcheck code. |
| CPU\$_CPUDATA* | Processor-specific hardware revision information. The first longword of this 16-byte field always contains the processor's system ID (SID) register, and is also defined as CPU\$_SID. |
| CPU\$_MCHK_MASK* | Function mask for current machine check recovery block. |
| CPU\$_MCHK_SP* | Saved SP for return at end of machine check recovery block. This field is zero if there is no current recovery block. |
| CPU\$_POPT_PAGE* | System virtual address of a page reserved to this processor that is used as a P0 page table when memory management is being enabled. |
| CPU\$Q_SWIQFL* | Twelve longwords representing the forward and backward links for the software interrupt queues (fork IPLs 6 through 11). |
| CPU\$_PSFL* | I/O postprocessing queue forward link. |
| CPU\$_PSBL* | I/O postprocessing queue backward link. |
| CPU\$Q_WORK_FQFL* | Work packet queue. This field is also called CPU\$Q_WORK_IFQ. |
| CPU\$_QLOST_FQFL* | Quorum loss fork queue forward link. |
| CPU\$_QLOST_FQBL* | Quorum loss fork queue blink link. |
| CPU\$W_QLOST_SIZE* | Quorum loss fork block size. |
| CPU\$_QLOST_TYPE* | Quorum loss fork block type. |
| CPU\$_QLOST_FLCK* | Quorum loss fork lock. |
| CPU\$_QLOST_FPC* | Quorum loss fork PC. |
| CPU\$_QLOST_FR3* | Quorum loss fork R3. |
| CPU\$_QLOST_FR4* | Quorum loss fork R4. |
| CPU\$Q_BOOT_TIME* | System time at which this processor was bootstrapped. |
| CPU\$Q_CPUID_MASK* | Bit mask representing this processor's CPU ID. |
| CPU\$_PHY_CPUID* | Integer that uniquely identifies the local processor in a multiprocessor configuration. This value is system specific. (For example, in a VAX 8300/8350 configuration, it is the VAXBI node ID. For a VAX 8800, it is the left or right bit from the processor's system ID register (PR\$_SID); for a VAX 8830/8840 it is the CPU number (0 to 3) from PR\$_SID. In a VAX 6200-series configuration, it is the XMI node ID. VMS uses the physical ID principally to locate the per-CPU database and interrupt stack of a processor that it is restarting.) |

Data Structures

A.4 Per-CPU Database (CPU)

Table A-4 (Cont.) Per-CPU Database (CPU)

| Field | Contents |
|---------------------|---|
| CPU\$_CAPABILITY* | Bit mask of this processor's capabilities. VMS defines the following capabilities in \$CPBDEF: CPB\$_PRIMARY Primary CPU. CPB\$_NS Reserved to DIGITAL. CPB\$_QUORUM Quorum required. CPB\$_HARDAFF Hard affinity. Reserved for diagnostics software. |
| CPU\$_TENUSEC* | 10-microsecond delay value. |
| CPU\$_UBDELAY* | UNIBUS delay counter. |
| CPU\$_KERNEL* | Set of seven longwords that tally the processor's clock ticks in kernel mode, in executive mode, in supervisor mode, in user mode, on the interrupt stack, in compatibility mode, and in kernel-mode spin-lock busy-wait state, respectively. |
| CPU\$_NULLCPU* | Clock ticks during which the null job has been the current process on this processor. |
| CPU\$_UKERNEL* | Reserved to DIGITAL. |
| CPU\$_UNULLCPU* | Reserved to DIGITAL. |
| CPU\$_CLKUTICS* | Reserved to DIGITAL. |
| CPU\$_HARDAFF* | Count of processes with hard affinity for this processor. |
| CPU\$_RANK_VEC* | Longword recording the ranks of all spin locks currently held by the processor. Spin lock acquisition code issues a Find First Set (FFS) instruction on this longword to determine if the processor holds any locks that are lower ranked than the one it seeks. |
| CPU\$_IPL_VEC* | Vector recording, in inverse order, the IPLs of all spin locks currently held by the processor (that is, bit 0 represents IPL 31). |
| CPU\$_IPL_ARRAY* | Array of 32 longwords, corresponding in inverse order to the 32 IPLs (that is, the first longword represents IPL 31). Upon each successful spin lock acquisition by this processor, the IPL vector corresponding to the spin lock's synchronization IPL (SPL\$_IPL) is incremented. |
| CPU\$_TPOINTER* | Address of the sanity timer (CPU\$_SANITY_TIMER) of the active processor with the next highest CPU ID. |
| CPU\$_SANITY_TIMER* | Number of sanity cycles before this processor times out. |
| CPU\$_SANITY_TICKS* | Number of clock ticks until the next sanity cycle. |

A.5 Channel Request Block (CRB)

The activity of each controller in a configuration is described in a channel request block (CRB). This data structure contains pointers to the wait queue of drivers ready to gain access to a device through the controller. It also stores the entry points to the driver's interrupt service routines and unit/controller initialization routines.

The channel request block is illustrated in Figure A-6 and described in Table A-5.

Data Structures

A.5 Channel Request Block (CRB)

Figure A-6 Channel Request Block (CRB)

| | | | |
|---------------------------|-------------|----------------|-----|
| CRB\$_FQFL | | | 00 |
| CRB\$_FQBL | | | 04 |
| CRB\$_FLCK | CRB\$_TYPE* | CRB\$_W_SIZE* | 08 |
| CRB\$_FPC | | | 12 |
| CRB\$_FR3 | | | 16 |
| CRB\$_FR4 | | | 20 |
| CRB\$_WQFL* | | | 24 |
| CRB\$_WQBL* | | | 28 |
| unused | | CRB\$_TT_TYPE* | 32 |
| CRB\$_UNIT_BRK* | CRB\$_MASK* | CRB\$_W_REFC* | 36 |
| CRB\$_AUXSTRUC | | | 40 |
| CRB\$_TIMELINK* | | | 44 |
| CRB\$_DUETIME* | | | 48 |
| CRB\$_TOUTROUT* | | | 52 |
| CRB\$_LINK* | | | 56 |
| CRB\$_DLCK* | | | 60 |
| CRB\$_BUGCHECK* | | | 64 |
| CRB\$_RTINTD* (12 bytes) | | | 68 |
| CRB\$_INTD* (40 bytes) | | | 80 |
| CRB\$_BUGCHECK2* | | | 120 |
| CRB\$_RTINTD2* (12 bytes) | | | 124 |
| CRB\$_INTD2* (40 bytes) | | | 136 |

ZK-6632-HC

Data Structures

A.5 Channel Request Block (CRB)

Table A–5 Contents of Channel Request Block

| Field Name | Contents |
|----------------|---|
| CRB\$_FOFL | Fork queue forward link. The link points to the next entry in the fork queue. Controller initialization routines write this field when they must drop IPL to utilize certain executive routines, such as those that allocate memory, that must be called at a lower IPL. The CRB timeout mechanism also uses the CRB fork block to lower IPL prior to calling the CRB timeout routine. |
| CRB\$_FOBL | Fork queue backward link. The link points to the previous entry in the fork queue. |
| CRB\$_SIZE* | Size of CRB. The driver-loading procedure writes this field when it creates the CRB. |
| CRB\$_TYPE* | Type of data structure. The driver-loading procedure writes the symbolic constant DYN\$_CRB into this field when it creates the CRB. |
| CRB\$_FLCK | Fork lock at which the controller's fork operations are synchronized. If it must use the CRB fork block, a driver either uses a DPT_STORE macro to initialize this field or explicitly sets its value within the controller initialization routine. |
| CRB\$_FPC | Address of instruction at which execution resumes when the VMS fork dispatcher dequeues the fork block. EXE\$FORK writes this field when called to suspend driver execution. |
| CRB\$_FR3 | Value of R3 at the time that the executing code requests VMS to create a fork block. EXE\$FORK writes this field when called to suspend driver execution. |
| CRB\$_FR4 | Value of R4 at the time that the executing code requests VMS to create a fork block. EXE\$FORK writes this field when called to suspend driver execution. |
| CRB\$_WOFL* | Controller data channel wait queue forward link. IOC\$REQxCHANy and IOC\$RELxCHAN insert and remove driver fork block addresses in this field. A channel wait queue contains addresses of driver fork blocks that record the context of suspended drivers waiting to gain control of a controller data channel. If a channel is busy when a driver requests access to the channel, IOC\$REQxCHANy suspends the driver by saving the driver's context in the device's UCB fork block and inserting the fork block address in the channel wait queue. When a driver releases a channel because an I/O operation no longer needs the channel, IOC\$RELxCHAN dequeues a driver fork block, allocates the channel to the driver, and reactivates the suspended driver fork process. If no drivers are awaiting the channel, IOC\$RELxCHAN clears the channel busy bit. |
| CRB\$_WOBL* | Controller channel wait queue backward link. IOC\$REQxCHANy and IOC\$RELxCHAN read and write this field. |
| CRB\$_TT_TYPE* | Type of controller (for instance, DZ11 or DZ32) for terminals. A terminal port driver fills in this field. |
| CRB\$_REFC* | UCB reference count. The driver-loading procedure increases the value in this field each time it creates a UCB for a device attached to the controller. |

Data Structures

A.5 Channel Request Block (CRB)

Table A-5 (Cont.) Contents of Channel Request Block

| Field Name | Contents |
|------------------|--|
| CRB\$B_MASK* | Mask that describes controller status. The following fields are defined in CRB\$B_MASK: |
| CRB\$V_BSY | Busy bit. IOC\$REQxCHANy 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\$V_UNINIT | Indication, when set, that the VMS adapter initialization routine has created a CRB for a generic VAXBI device, but has not yet called its controller initialization routine. SYSGEN reads this bit to determine whether to call the controller initialization routine and clears it when the initialization routine completes. This facilitates SYSGEN's processing of multiunit generic VAXBI devices. |
| CRB\$B_UNIT_BRK* | Break bits for terminal lines. Used by VMS terminal port drivers. |
| CRB\$L_AUXSTRUC | Address of auxiliary data structure used by device driver to store special controller information. A device driver requiring such a structure generally allocates a block of nonpaged dynamic memory in its controller initialization routine and places a pointer to it in this field. |
| CRB\$L_TIMELINK* | Forward link in queue of CRBs waiting for periodic wakeups. This field points to the CRB\$L_TIMELINK field of the next CRB in the list. The CRB\$L_TIMELINK field of the last CRB in the list contains zero. The listhead for this queue is IOC\$GL_CRBTMOUT. Use of this field is reserved to DIGITAL. |
| CRB\$L_DUETIME* | Time in seconds, relative to EXE\$GL_ABSTIM, at which next periodic wakeup associated with the CRB is to be delivered. Compute this value by raising IPL to IPL\$_POWER, adding the desired number of seconds to the contents of EXE\$GL_ABSTIM, and storing the result in this field. Use of this field is reserved to DIGITAL. |
| CRB\$L_TOUTROUT* | Address of routine to be called at fork IPL (holding a corresponding fork lock if necessary) when a periodic wakeup associated with CRB becomes due. The routine must compute and reset the value in CRB\$L_DUETIME if another periodic wakeup request is desired. Use of this field is reserved to DIGITAL. |
| CRB\$L_LINK* | Address of secondary CRB (for MASSBUS devices only). This field is written by the driver-loading procedure and read by IOC\$REOSCHANx and IOC\$RELSCHAN. |
| CRB\$L_DLCK* | Address of controller's device lock. The driver-loading procedure initializes this field and propagates it to each UCB it creates for the device units associated with the controller. |
| CRB\$L_BUGCHECK* | Bugcheck data used to issue an ILLOBUSCFG bugcheck when the multilevel interrupt dispatching code (at CRB\$L_RTINTD) determines that a Q22 bus is illegally configured. |
| CRB\$L_RTINTD* | Portion of interrupt transfer vector created at system initialization when a MicroVAX 3600-series system or MicroVAX II system implements multilevel device interrupt dispatching. The code stored in this 12-byte field implements a conditional lowering to device IPL. See Figure A-7 and Table A-6 for a description of the contents of the interrupt transfer vector. |

Data Structures

A.5 Channel Request Block (CRB)

Table A-5 (Cont.) Contents of Channel Request Block

| Field Name | Contents |
|------------------|---|
| CRB\$_INTD* | <p>Interrupt transfer vector. This 10-longword field (described in Figure A-7 and Table A-6) stores executable code, driver entry points, and I/O adapter information. It contains pointers to the driver's controller and unit initialization routines, the interrupt dispatch block (IDB), and the adapter control block (ADP). It may also contain fields that describe the disposition of a controller's data paths and map registers. The interrupt transfer routine is located at the top of the interrupt transfer vector.</p> <p>Although certain of the symbolic offsets defined in the data structure definition macro \$VECDEF have negative values, driver code can uniformly refer to the contents of the VEC structure in the following form:</p> $\text{CRB}\$_\text{INTD} + \text{VEC}\$_x_symbol.$ |
| CRB\$_BUGCHECK2* | <p>Bugcheck data used to issue an ILLQBUSCFG bugcheck when the multilevel interrupt dispatching code (at CRB\$_RTINTD2) determines that the Q22 bus is illegally configured.</p> |
| CRB\$_RTINTD2* | <p>Portion of second interrupt transfer vector initialized and used if multilevel interrupt dispatching is enabled in a MicroVAX 3600-series system or MicroVAX II system. See Figure A-7 and Table A-6 for a description of the contents of the interrupt transfer vector.</p> |
| CRB\$_INTD2* | <p>Second interrupt transfer vector for devices with multiple interrupt vectors. The data structure definition macro \$CRBDEF supplies symbolic offsets for only the first two interrupt transfer vector structures.</p> <p>VMS creates the appropriate number of interrupt transfer vector structures (as shown in Figure A-7) within a CRB if a driver specifies that the addresses of additional interrupt service routines be loaded into these structures. For example:</p> <pre>DPT_STORE, CRB, CRB\$_INTD2 + VEC\$_ISR, D, isr_for_vec2 DPT_STORE, CRB, CRB\$_INTD + (2 * VEC\$_K_LENGTH) + VEC\$_ISR, D, isr_for_vec3</pre> <p>The offset of the <i>n</i>th vector located within the CRB is equal to the result of the following formula:</p> $\text{CRB}\$_\text{INTD} + (n * \text{VEC}\$_\text{K_LENGTH})$ <p>VMS automatically initializes the interrupt dispatching instructions and the data structure locations from information located in the primary vector. The number of device vectors and vector structures actually created can be overridden by the value specified in the /NUMVEC qualifier to the SYSGEN command CONNECT.</p> |

Data Structures

A.5 Channel Request Block (CRB)

Figure A-7 Interrupt Transfer Vector Block (VEC)

| | | | |
|---------------------------|---------------|---------------|-----|
| VEC\$L_BUGCHECK* | | | -16 |
| VEC\$L_RTINTD* (12 bytes) | | | -12 |
| VEC\$L_INTD* | | | 00 |
| VEC\$L_ISR | | | 04 |
| VEC\$L_IDB* | | | 08 |
| VEC\$L_INITIAL | | | 12 |
| VEC\$B_DATAPATH | VEC\$B_NUMREG | VEC\$W_MAPREG | 16 |
| VEC\$L_ADP* | | | 20 |
| VEC\$L_UNITINIT* | | | 24 |
| VEC\$L_START* | | | 28 |
| VEC\$L_UNITDISC* | | | 32 |
| VEC\$W_NUMALT | | VEC\$W_MAPALT | 36 |

ZK-6633-HC

Data Structures

A.5 Channel Request Block (CRB)

Table A-6 Interrupt Dispatch Vector Block (VEC)

| Field Name | Contents |
|-----------------|--|
| VEC\$_BUGCHECK* | Bugcheck data used to issue an ILLQBUSCFG bugcheck when the multilevel interrupt dispatching code determines that the Q22 bus is illegally configured. |
| VEC\$_RTINTD* | <p>Portion of interrupt transfer vector created at system initialization when a MicroVAX 3600-series system or MicroVAX II system implements multilevel device interrupt dispatching. The code stored in this 12-byte field implements a conditional lowering to device IPL, as follows:</p> <pre> CMPZV #PSL\$V_IPL, #PSL\$\$_IPL, - 4(SP), S^#DIPL BGEQ BUGCHECK SETIPL S^#DIPL </pre> |
| VEC\$_INTD* | <p>Interrupt dispatching code, written by the driver-loading procedure as follows:</p> <pre> PUSHR #^M<R0,R1,R2,R3,R4,R5> JSB @# </pre> <p>The destination of the JSB instruction is the driver's interrupt service routine, as indicated at offset VEC\$_ISR. Under normal operations, direct-vector UNIBUS or Q22 bus adapters—as well as VAXBI system interrupt dispatching—transfer control to CRB\$_INTD. The code located here causes the processor to execute the PUSHR instruction to save R0 through R5 on the stack and execute a JSB instruction to transfer control to the driver's interrupt service routine.</p> <p>In dispatching interrupts from non-direct-vector UNIBUS adapters, the UNIBUS adapter interrupt service routine transfers control to CRB\$_INTD+2, which contains the JSB instruction to the driver's interrupt service routine. Because the UNIBUS adapter's interrupt service routine has already saved R0 through R5, interrupt dispatching bypasses the PUSHR instruction in these instances.</p> <p>This field, plus VEC\$_ISR, is also known as VEC\$_DISPATCH.</p> |
| VEC\$_ISR | The DPT in every driver for an interrupting device specifies the address of a driver interrupt service routine. |
| VEC\$_IDB* | <p>Address of IDB for controller. The driver-loading procedure creates an IDB for each CRB and loads the address of the IDB in this field. Device drivers use the IDB address to obtain the virtual addresses of device registers.</p> <p>When a driver's interrupt service routine gains control, the top of the stack contains a pointer to this field.</p> |
| VEC\$_INITIAL | <p>Address of controller initialization routine. If a device controller requires initialization at driver-loading time and during recovery from a power failure, the driver specifies a value for this field in the DPT.</p> <p>The driver-loading procedure calls this routine each time the procedure loads the driver. The VMS power failure recovery procedure also calls this routine to initialize a controller after a power failure.</p> |

Data Structures

A.5 Channel Request Block (CRB)

Table A-6 (Cont.) Interrupt Dispatch Vector Block (VEC)

| Field Name | Contents |
|------------------|---|
| VEC\$W_MAPREG | The following bits are defined within VEC\$W_MAPREG: |
| VEC\$V_MAPREG | Number of first standard map register allocated to the driver that owns controller data channel. IOC\$REQMAPREG writes this field when the routine allocates a set of standard map registers to a driver fork process for a DMA transfer. IOC\$RELMAPREG reads the field to deallocate a set of map registers. Device drivers read this field in calculating the starting address of a UNIBUS or MicroVAX 3600-series/MicroVAX II Q22 bus transfer. |
| VEC\$B_NUMREG | VEC\$V_MAPLOCK Map register set is permanently allocated (when set). |
| VEC\$B_DATAPATH | Number of UNIBUS adapter or MicroVAX Q22 bus standard map registers allocated to driver. IOC\$REQMAPREG writes this 15-bit field when the routine allocates a set of standard map registers. IOC\$RELMAPREG reads this field to deallocate a set of standard map registers. |
| VEC\$V_DATAPATH | Data path specifier. The bits that make up this field are used as follows: Number of data path used in DMA transfer. The routine IOC\$REQDATAP writes this 5-bit field when a buffered data path is allocated and clears the field when the data path is released. The routine IOC\$LOADUBAMAP copies the contents of this field into UNIBUS adapter map registers. These bits also serve as implicit input to the IOC\$PURGDATAP routine. |
| VEC\$V_LWAE | Longword access enable (LWAE) bit. Drivers set this bit when they wish to limit the data path to longword-aligned, random-access mode. The routine IOC\$LOADUBAMAP copies the value in this field to the UNIBUS adapter map registers. |
| <6> | Reserved to DIGITAL. |
| VEC\$V_PATHLOCK | Buffered data path allocation indicator. Drivers set this bit to specify that the buffered data path is permanently allocated. |
| VEC\$L_ADP* | Address of ADP. The SYSGEN command CONNECT must specify the nexus number of the UNIBUS adapter used by a controller. The driver-loading procedure writes the address of the ADP for the specified UBA into the VEC\$L_ADP field. IOC\$REQMAPREG, IOC\$REQALTMAP, and IOC\$RELMAPREG read and write fields in the ADP to allocate and deallocate map registers. |
| VEC\$L_UNITINIT* | Address of device driver's unit initialization routine. If a device unit requires initialization at driver-loading time and during recovery from a power failure, the driver specifies a value for this field in the DPT. The driver-loading procedure calls this routine for each device unit each time the procedure loads the driver. The VMS power failure recovery procedure also calls this routine to initialize device units after a power failure. MASSBUS drivers that support mixed device types must not use this field. Instead, they should specify the unit initialization routine in the unit initialization field of the DDT (DDT\$L_UNITINIT). Other drivers can use either field. |

Data Structures

A.5 Channel Request Block (CRB)

Table A-6 (Cont.) Interrupt Dispatch Vector Block (VEC)

| Field Name | Contents | | | | |
|-----------------|--|--------------|--|---------------|---|
| VEC\$_START* | Address of VMS start protocol routine. Use of this field is reserved to DIGITAL. | | | | |
| VEC\$_UNITDISC* | Address of unit disconnect routine. Use of this field is reserved to DIGITAL. | | | | |
| VEC\$_MAPALT | The following bits are defined within VEC\$_MAPALT: | | | | |
| | <table style="width: 100%; border: none;"> <tr> <td style="width: 30%; vertical-align: top;">VEC\$_MAPALT</td> <td>Number of first Q22 bus alternate map register allocated to driver that owns controller data channel. IOC\$REQALTMAP writes this field when the routine allocates a set of Q22 bus alternate map registers to a driver fork process for a DMA transfer. IOC\$RELMAPREG reads the field to deallocate a set of map registers. Device drivers read this 15-bit field in calculating the starting address of a MicroVAX 3600-series/MicroVAX II Q22 bus transfer that uses a set of alternate map registers.</td> </tr> <tr> <td style="vertical-align: top;">VEC\$_ALTLOCK</td> <td>Alternate map register set is permanently allocated (when set).</td> </tr> </table> | VEC\$_MAPALT | Number of first Q22 bus alternate map register allocated to driver that owns controller data channel. IOC\$REQALTMAP writes this field when the routine allocates a set of Q22 bus alternate map registers to a driver fork process for a DMA transfer. IOC\$RELMAPREG reads the field to deallocate a set of map registers. Device drivers read this 15-bit field in calculating the starting address of a MicroVAX 3600-series/MicroVAX II Q22 bus transfer that uses a set of alternate map registers. | VEC\$_ALTLOCK | Alternate map register set is permanently allocated (when set). |
| VEC\$_MAPALT | Number of first Q22 bus alternate map register allocated to driver that owns controller data channel. IOC\$REQALTMAP writes this field when the routine allocates a set of Q22 bus alternate map registers to a driver fork process for a DMA transfer. IOC\$RELMAPREG reads the field to deallocate a set of map registers. Device drivers read this 15-bit field in calculating the starting address of a MicroVAX 3600-series/MicroVAX II Q22 bus transfer that uses a set of alternate map registers. | | | | |
| VEC\$_ALTLOCK | Alternate map register set is permanently allocated (when set). | | | | |
| VEC\$_NUMALT | Number of Q22 bus alternate map registers allocated to driver. IOC\$REQALTMAP writes this field when allocating a set of alternate map registers. IOC\$RELMAPREG reads this field to deallocate a set of alternate map registers. | | | | |

A.6 Device Data Block (DDB)

The device data block (DDB) is a block that identifies the generic device/controller name and driver name for a set of devices attached to a single controller. The driver-loading procedure creates a DDB for each controller during autoconfiguration at system startup, and dynamically creates additional DDBs for new controllers as they are added to the system using the SYSGEN command CONNECT. The procedure initializes all fields in the DDB. All the DDBs in the I/O database are linked in a singly linked list. The contents of IOC\$GL_DEVLIST point to the first entry in the list.

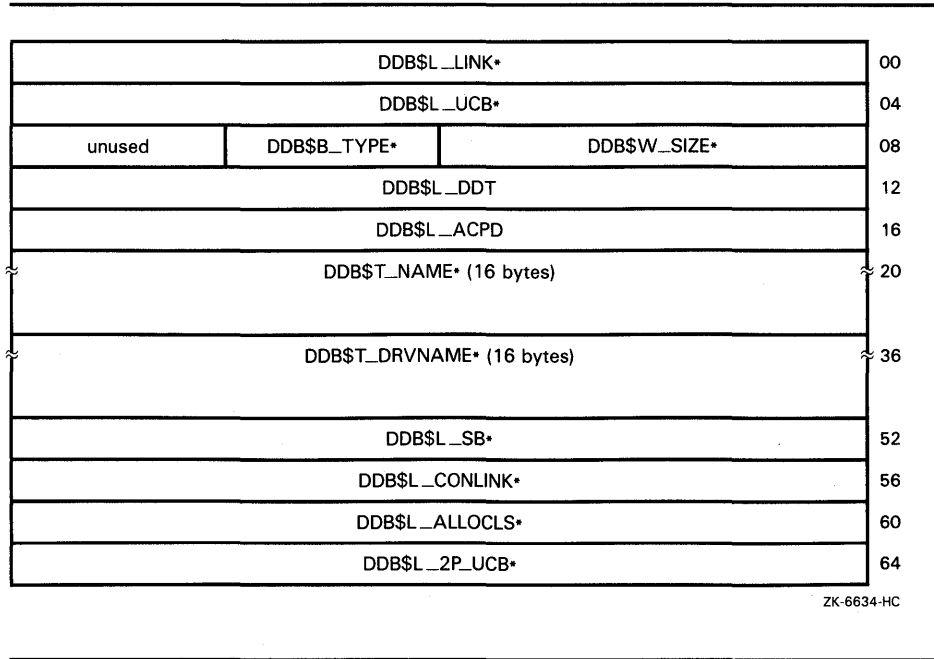
VMS routines and device drivers refer to the DDB.

The device data block is illustrated in Figure A-8 and described in Table A-7.

Data Structures

A.6 Device Data Block (DDB)

Figure A-8 Device Data Block (DDB)



Data Structures

A.6 Device Data Block (DDB)

Table A-7 Contents of Device Data Block

| Field Name | Contents |
|----------------|--|
| DDB\$_LINK* | Address of next DDB. A zero indicates that this is the last DDB in the DDB chain. |
| DDB\$_UCB* | Address of UCB for first unit attached to controller. |
| DDB\$_SIZE* | Size of DDB. |
| DDB\$_TYPE* | Type of data structure. The driver-loading procedure writes the constant DYN\$_DDB into this field when the procedure creates the DDB. |
| DDB\$_DDT | Address of DDT. VMS can transfer control to a device driver only through addresses listed in the DDT, the CRB, and the UCB fork block. The DPT of every device driver must specify a value for this field. |
| DDB\$_ACPD | Name of default ACP (or XQP) for controller. ACPs that control access to file-structured devices (or the XQP) use the high-order byte of this field, DDB\$_ACPCCLASS, to indicate the class of the file-structured device. If the ACP_MULTIPLE system parameter is set, the initialization procedure creates a unique ACP for each class of file-structured device. Drivers initialize DDB\$_ACPCCLASS by invoking a DPT_STORE macro. Values for DDB\$_ACPCCLASS are as follows: DDB\$_PACK Standard disk pack DDB\$_CART Cartridge disk pack DDB\$_SLOW Floppy disk DDB\$_TAPE Magnetic tape that simulates file-structured device |
| DDB\$_NAME* | Generic name for the devices attached to controller. The first byte of this field is the number of characters in the generic name. The remainder of the field consists of a string of up to 15 characters that, suffixed by a device unit number, identifies devices on the controller. |
| DDB\$_DRVNAME* | Name of device driver for controller. The first byte of this field is the number of characters in the driver name. The remainder of the field contains a string of up to 15 characters taken from the DPT in the driver. |
| DDB\$_SB* | Address of system block. |
| DDB\$_CONLINK* | Address of next DDB in the connection subchain. |
| DDB\$_ALLOCLS* | Allocation class of device. |
| DDB\$_2P_UCB* | Address of the first UCB on the secondary path. Another name for this field is DDB\$_DP_UCB. |

A.7 Driver Dispatch Table (DDT)

Each device driver contains a driver dispatch table (DDT). The DDT lists entry points in the driver that VMS routines call: for instance, the entry point for the driver start-I/O routine.

A device driver creates a DDT by invoking the VMS macro DDTAB. The fields in the driver dispatch table are illustrated in Figure A-9 and described in Table A-8.

Data Structures

A.7 Driver Dispatch Table (DDT)

Figure A-9 Driver Dispatch Table (DDT)

| | | |
|--------------------|----------------|----|
| DDT\$_START | | 00 |
| DDT\$_UNSOLINT | | 04 |
| DDT\$_FDT | | 08 |
| DDT\$_CANCEL | | 12 |
| DDT\$_REGDUMP | | 16 |
| DDT\$_ERRORBUF | DDT\$_DIAGBUF | 20 |
| DDT\$_UNITINIT | | 24 |
| DDT\$_ALTSTART | | 28 |
| DDT\$_MNTVER | | 32 |
| DDT\$_CLONEDUCB | | 36 |
| unused | DDT\$_FDTSIZE* | 40 |
| DDT\$_MNTV_SSSC* | | 44 |
| DDT\$_MNTV_FOR* | | 48 |
| DDT\$_MNTV_SQD* | | 52 |
| DDT\$_AUX_STORAGE* | | 56 |
| DDT\$_AUX_ROUTINE* | | 60 |

ZK-6635-HC

Data Structures

A.7 Driver Dispatch Table (DDT)

Table A-8 Contents of Driver Dispatch Table

| Field Name | Contents |
|----------------|--|
| DDT\$_START | <p>Entry point to the driver's start-I/O routine. Every driver must specify this address in the start argument to the DDTAB macro.</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\$_UNSOLINT | <p>Entry point to a MASSBUS driver's unsolicited-interrupt service routine. The driver specifies this address in the unsolic argument to the DDTAB macro.</p> <p>This field contains the address of a routine that analyzes unexpected interrupts from a device. The standard 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 interrupt service routine can call the unsolicited-interrupt service routine.</p> |
| DDT\$_FDT | <p>Address of the driver's FDT. Every driver must specify this address in the functb argument to the DDTAB macro.</p> <p>EXE\$QIO refers to the FDT to validate I/O function codes, decide which functions are buffered, and call FDT routines associated with function codes.</p> |
| DDT\$_CANCEL | <p>Entry point to the driver's cancel-I/O routine. The driver specifies this address in the cancel argument to the DDTAB macro.</p> <p>Some devices require special cleanup processing when a process or a 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\$_REGDUMP | <p>Entry point to the driver's register dumping routine. The driver specifies this address in the regdmp argument to the DDTAB macro.</p> <p>IOC\$DIAGBUFILL, ERL\$DEVICERR, and ERL\$DEVICTMO call the address contained in this field to write device register contents into a diagnostic buffer or error message buffer.</p> |
| DDT\$_DIAGBUF | <p>Size of diagnostic buffer. The driver specifies this value in the diagbf argument to the DDTAB macro. The value is the size in bytes of a diagnostic buffer for the device.</p> <p>When EXE\$QIO preprocesses an I/O request, it allocates a system buffer of the size recorded in this field (if it contains a nonzero value) if the process requesting the I/O has DIAGNOSE privilege and specifies a diagnostic buffer in the I/O request. IOC\$DIAGBUFILL fills the buffer after the I/O operation completes.</p> |
| DDT\$_ERRORBUF | <p>Size of error message buffer. The driver specifies this value in the erlgbf argument to the DDTAB macro. The value is the size in bytes of an error message 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 buffer if an error has occurred.</p> |
| DDT\$_UNITINIT | <p>Address of the device's unit initialization routine, if one exists. Drivers for MASSBUS devices use this field rather than CRB\$_INTD+VEC\$_UNITINIT. Drivers for UNIBUS, VAXBI, and Q22 devices can use either field.</p> |
| DDT\$_ALTSTART | <p>Address of a driver's alternate start-I/O routine. EXE\$ALTQUEPKT transfers control to the alternate start-I/O routine at this address.</p> |

Data Structures

A.7 Driver Dispatch Table (DDT)

Table A-8 (Cont.) Contents of Driver Dispatch Table

| Field Name | Contents |
|--------------------|--|
| DDT\$_MNTVER | Address of the VMS routine (IOC\$MNTVER) called at the beginning and end of mount verification operation. The mntver argument to the DPTAB macro defaults to this routine. Use of the mntver argument to call any routine other than IOC\$MNTVER is reserved to DIGITAL. |
| DDT\$_CLONEDUCB | Address of routine to call when UCB is cloned. |
| DDT\$_W_FDTSIZE* | Number of bytes in FDT. The driver-loading procedure uses this field to relocate addresses in the FDT to system virtual addresses. |
| DDT\$_MNTV_SSSC* | Address of routine to call when performing mount verification for a shadow-set state change. Use of this field is reserved to DIGITAL. |
| DDT\$_MNTV_FOR* | Address of routine to call when performing mount verification for a foreign device. Use of this field is reserved to DIGITAL. |
| DDT\$_MNTV_SQD* | Address of routine to call when performing mount verification for a sequential device. Use of this field is reserved to DIGITAL. |
| DDT\$_AUX_STORAGE* | Address of auxiliary storage area. Use of this field is reserved to DIGITAL. |
| DDT\$_AUX_ROUTINE* | Address of auxiliary routine. Use of this field is reserved to DIGITAL. |

A.8 Driver Prologue Table (DPT)

When loading a device driver and its database into virtual memory, the driver-loading procedure finds the basic description of the driver and its device in a driver prologue table (DPT). The DPT provides the length, name, adapter type, and loading and reloading specifications for the driver.

A device driver creates a DPT by invoking the VMS macros DPTAB and DPT_STORE. The driver prologue table is illustrated in Figure A-10 and described in Table A-9.

Data Structures

A.8 Driver Prologue Table (DPT)

Figure A-10 Driver Prologue Table (DPT)

| | | | |
|-----------------------|----------------|---------------|-----|
| DPT\$_FLINK* | | | 00 |
| DPT\$_BLINK* | | | 04 |
| DPT\$_REFC* | DPT\$_TYPE* | DPT\$_SIZE | 08 |
| DPT\$_UCBSIZE | unused | DPT\$_ADPTYPE | 12 |
| DPT\$_FLAGS | | | 16 |
| DPT\$_REINITTAB | DPT\$_INITTAB | | 20 |
| DPT\$_MAXUNITS | DPT\$_UNLOAD | | 24 |
| DPT\$_DEFUNITS | DPT\$_VERSION* | | 28 |
| DPT\$_VECTOR | DPT\$_DELIVER | | 32 |
| DPT\$_NAME (12 bytes) | | | 36 |
| DPT\$_LINKTIME* | | | 48 |
| DPT\$_ECOLEVEL* | | | 56 |
| DPT\$_UCODE* | | | 60 |
| DPT\$_LMF_1* | | | 64 |
| DPT\$_LMF_2* | | | 72 |
| DPT\$_LMF_3* | | | 80 |
| DPT\$_LMF_4* | | | 88 |
| DPT\$_LMF_5* | | | 96 |
| DPT\$_LMF_6* | | | 104 |
| DPT\$_LMF_7* | | | 112 |
| DPT\$_LMF_8* | | | 120 |
| DPT\$_DECW_SNAME* | | | 124 |

ZK-6636-HC

Data Structures

A.8 Driver Prologue Table (DPT)

Table A-9 Contents of Driver Prologue Table

| Field Name | Contents |
|---------------|---|
| DPT\$_FLINK* | Forward link to next DPT. The driver-loading procedure writes this field. The procedure links all DPTs in the system in a doubly linked list. |
| DPT\$_BLINK* | Backward link to previous DPT. The driver-loading procedure writes this field. |
| DPT\$_SIZE | Size in bytes of the driver. The DPTAB macro writes this field by subtracting the address of the beginning of the DPT from the address specified as the end argument to the DPTAB macro. The driver-loading procedure uses this value to determine the space needed in nonpaged system memory to load the driver. |
| DPT\$_TYPE* | Type of data structure. The DPTAB macro always writes the symbolic constant DYN\$_DPT into this field. |
| DPT\$_REFC* | Number of DDBs that refer to the driver. The driver-loading procedure increments the value in this field each time the procedure creates another DDB that points to the driver's DDT. |
| DPT\$_ADPTYPE | Type of adapter used by the devices using this driver. Every driver must specify the string "UBA", "MBA", "GENBI", "NULL", or "DR" as the value of the adapter argument to the DPTAB macro. Q22 bus drivers should specify "UBA" as the adapter type. The macro writes the value AT\$_UBA, AT\$_MBA, or AT\$_GENBI in this field. |
| DPT\$_UCBSIZE | Size in bytes of the unit control block for a device that uses this driver. Every driver must specify a value for this field in the ucbsize argument to the DPTAB macro. The driver-loading procedure allocates blocks of nonpaged system memory of the specified size when creating UCBs for devices associated with the driver. |
| DPT\$_FLAGS | Driver-loading flags. The driver can specify any of a set of flags as the value of the flags argument to the DPTAB macro. The driver-loading procedure modifies its loading and reloading algorithm based on the settings of these flags. Flags defined in the flag field include the following: DPT\$_SUBCNTRL Device is a subcontroller. DPT\$_SVP Device requires permanent system page to be allocated during driver loading. DPT\$_NOUNLOAD Driver cannot be reloaded. DPT\$_SCS SCS code must be loaded with this driver. DPT\$_DUSHADOW Driver is the shadowing disk class driver. DPT\$_SCSCI Common SCS/CI subroutines must be loaded with this driver. DPT\$_BVPSUBS Common BVP subroutines must be loaded with this driver. DPT\$_UCODE Driver has an associated microcode image. DPT\$_SMPMOD Driver has been designed to run in a VMS multiprocessing environment. DPT\$_DECW_DECODE Driver is a decoding driver. This field is also known as DPT\$_FLAGS. |

Data Structures

A.8 Driver Prologue Table (DPT)

Table A–9 (Cont.) Contents of Driver Prologue Table

| Field Name | Contents |
|------------------|--|
| DPT\$W_INITTAB | <p>Offset to driver initialization table. Every driver must specify a list of data structure fields and values to be written into the fields at the time that the driver-loading procedure creates the driver's data structures and loads the driver.</p> <p>The driver invokes the VMS macro DPT_STORE to specify these fields and their values.</p> |
| DPT\$W_REINITTAB | <p>Offset to driver-reinitialization table. Every driver must specify a list of data structure fields and values to be written into these fields at the time that the driver-loading procedure creates the driver's data structures and loads the driver or the driver is reloaded.</p> <p>The driver invokes the VMS macro DPT_STORE to specify these fields and their values.</p> |
| DPT\$W_UNLOAD | <p>Relative address of driver routine to be called when driver is reloaded. The driver specifies this field with the value of the unload argument to the DPTAB macro. The driver-loading procedure calls the driver unloading routine before reinitializing all device units associated with the driver.</p> |
| DPT\$W_MAXUNITS | <p>Maximum number of units on controller that this driver supports. Specify this value in the maxunits argument to the DPTAB macro. If no value is specified, the default is eight units.</p> |
| DPT\$W_VERSION* | <p>Version number that identifies format of DPT. The DPTAB macro automatically inserts a value in this field. SYSGEN checks its copy of the version number against the value stored in this field. If the values do not match, an error is generated. To correct the error, reassemble and relink the driver.</p> |
| DPT\$W_DEFUNITS | <p>Number of UCBs that the VMS autoconfiguration facility will automatically create. Drivers specify this number with the defunits argument to the DPTAB macro. If the driver also gives a value to DPT\$W_DELIVER, this field is also the number of times that the autoconfiguration facility calls the unit delivery routine.</p> |
| DPT\$W_DELIVER | <p>Relative address of the unit delivery routine that autoconfiguration facility calls for the number of UCBs specified in DPT\$W_DEFUNITS. The driver supplies the address of the unit delivery routine in the deliver argument to the DPTAB macro.</p> |
| DPT\$W_VECTOR | <p>Relative address of a driver-specific vector. A terminal class or port driver stores the address of its class or port entry vector table in this field.</p> |
| DPT\$t_NAME | <p>Name of the device driver. Field is 12 bytes. One byte records the length of the name string; the name string can be up to 11 characters. Drivers specify this field as the value of the name argument to the DPTAB macro.</p> <p>The driver-loading procedure compares the name of a driver to be loaded with the values in this field in all DPTs already loaded into system memory to ensure that it loads only one copy of a driver at a time.</p> |
| DPT\$Q_LINKTIME* | <p>Time and date at which driver was linked, taken from its image header.</p> |
| DPT\$L_ECOLEVEL* | <p>ECO level of driver, taken from its image header.</p> |
| DPT\$L_UCODE* | <p>Address of associated microcode image, if DPT\$V_UCODE is set in DPT\$L_FLAGS. Use of this field is reserved to DIGITAL.</p> |

Data Structures

A.8 Driver Prologue Table (DPT)

Table A-9 (Cont.) Contents of Driver Prologue Table

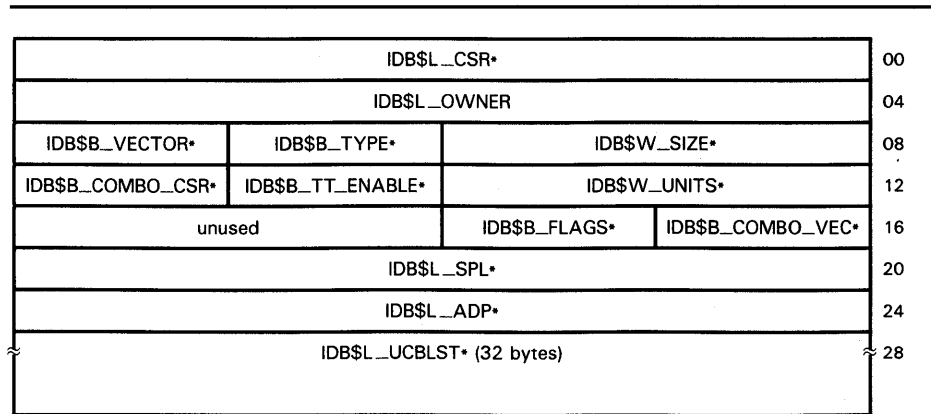
| Field Name | Contents |
|--------------------|---|
| DPT\$Q_LMF_1* | First of eight quadwords reserved to DIGITAL for the use of the VMS license management facility. (The others are DPT\$Q_LMF_2, DPT\$Q_LMF_3, DPT\$Q_LMF_4, DPT\$Q_LMF_5, DPT\$Q_LMF_6, DPT\$Q_LMF_7, and DPT\$Q_LMF_8.) |
| DPT\$W_DECW_SNAME* | Offset to counted ASCII string used by decoding drivers. |

A.9 Interrupt Dispatch Block (IDB)

The interrupt dispatch block (IDB) records controller characteristics. The driver-loading procedure creates and initializes this block when the procedure creates a CRB. The IDB points to the physical controller by storing the virtual address of the CSR. The CSR is the indirect pointer to all device unit registers.

The interrupt dispatch block is illustrated in Figure A-11 and described in Table A-10.

Figure A-11 Interrupt Dispatch Block (IDB)



ZK-6637-HC

Data Structures

A.9 Interrupt Dispatch Block (IDB)

Table A–10 Contents of Interrupt Dispatch Block

| Field Name | Contents |
|------------------|--|
| IDB\$_CSR* | Address of CSR. The SYSGEN command CONNECT specifies the address of a device's CSR. The driver-loading procedure writes the system virtual equivalent of this address into the IDB\$_CSR field. Device drivers set and clear bits in device registers by referencing all device registers at fixed offsets from the CSR address. The driver-loading procedure tests the value of this field. If the value is not a CSR address, it sets IDB\$_NO_CSR in IDB\$_FLAGS and places the device offline by clearing UCB\$_ONLINE in UCB\$_STS. In this event, it does not call the driver's controller and unit initialization routines. |
| IDB\$_OWNER | Address of UCB of device that owns controller data channel. IOC\$REQx CHANY writes a UCB address into this field when the routine allocates a controller data channel to a driver. IOC\$RELx CHAN confirms that the proper driver fork process is releasing a channel by comparing the driver's UCB with the UCB stored in the IDB\$_OWNER field. If the UCB addresses are the same, IOC\$RELx CHAN allocates the channel to a waiting driver by writing a new UCB address into the field. If no driver fork processes are waiting for the channel, IOC\$RELxCHAN clears the field. If the controller is a single-unit controller, the unit or controller initialization routine should write the UCB address of the single device into this field. |
| IDB\$_SIZE* | Size of IDB. The driver-loading procedure writes the constant IDB\$_LENGTH into this field when the procedure creates the IDB. |
| IDB\$_TYPE* | Type of data structure. The driver-loading procedure writes the symbolic constant DYN\$_IDB into this field when the procedure creates the IDB. |
| IDB\$_VECTOR* | Interrupt vector number of the device, right-shifted by two bits. SYSGEN writes a value into this field using either the autoconfiguration database or the value specified in the /VECTOR qualifier to the CONNECT command. Drivers for devices that define the interrupt vector address through a device register must use this field to load that register during unit initialization and reinitialization after a power failure. |
| IDB\$_UNITS* | Maximum number of units connected to the controller. The maximum number of units is specified in the DPT and can be overridden at driver-loading time. |
| IDB\$_TT_ENABLE* | Reserved for use by the VMS terminal driver. |
| IDB\$_COMBO_CSR* | Address of the start of CSRs for a multicontroller device such as the DMF32. (The name of this field is IDB\$_COMBO_CSR_OFFSET.) |
| IDB\$_COMBO_VEC* | Address of the start of interrupt vectors for a multicontroller device. (The name of this field is IDB\$_COMBO_VECTOR_OFFSET.) |
| IDB\$_FLAGS* | Flags associated with the IDB. The only flag currently defined is IDB\$_NO_CSR. The driver loading procedure sets this flag if IDB\$_CSR does not contain the address of a CSR. |
| IDB\$_SPL* | Address of the device lock that—in a VMS multiprocessing environment—synchronizes access to device registers and those fields in the UCB accessed at device IPL. |
| IDB\$_ADP* | Address of the adapter's ADP. The SYSGEN CONNECT command must specify the nexus number of the I/O adapter used by a device. The driver-loading procedure writes the address of the ADP for the specified I/O adapter into the IDB\$_ADP field. |

Data Structures

A.9 Interrupt Dispatch Block (IDB)

Table A-10 (Cont.) Contents of Interrupt Dispatch Block

| Field Name | Contents |
|---------------|---|
| IDB\$_UCBLST* | List of UCB addresses. The size of this field is the maximum number of units supported by the controller, as defined in the DPT. The maximum specified in the DPT can be overridden at driver load time. The driver-loading procedure writes a UCB address into this field every time the routine creates a new UCB associated with the controller. |

A.10 I/O Request Packet (IRP)

When a user process queues a valid I/O request by issuing a \$QIO or \$QIOW system service, the service creates an I/O request packet (IRP). The IRP contains a description of the request and receives the status of the I/O processing as it proceeds.

The I/O request packet is illustrated in Figure A-12 and described in Table A-11. Note that the standard IRP contains space for fields required by VMS multiprocessing and the VMS class drivers. Under no circumstances should a non-DIGITAL-supplied driver use these fields.

Data Structures

A.10 I/O Request Packet (IRP)

Figure A-12 I/O Request Packet (IRP)

| | | | |
|---------------------|-------------|-------------|----|
| IRP\$_IOQFL | | | 00 |
| IRP\$_IOQBL | | | 04 |
| IRP\$_RMOD* | IRP\$_TYPE* | IRP\$_SIZE* | 08 |
| IRP\$_PID* | | | 12 |
| IRP\$_AST* | | | 16 |
| IRP\$_ASTPRM* | | | 20 |
| IRP\$_WIND* | | | 24 |
| IRP\$_UCB* | | | 28 |
| IRP\$_PRI* | IRP\$_EFN* | IRP\$_FUNC | 32 |
| IRP\$_IOSB* | | | 36 |
| IRP\$_STS | | IRP\$_CHAN* | 40 |
| IRP\$_SVAPTE | | | 44 |
| IRP\$_BCNT | | IRP\$_BOFF | 48 |
| IRP\$_STS2 | | IRP\$_BCNT | 52 |
| IRP\$_IOST1 | | | 56 |
| IRP\$_IOST2 | | | 60 |
| IRP\$_ABCNT | | | 64 |
| IRP\$_OBCNT | | | 68 |
| IRP\$_SEGVBN | | | 72 |
| IRP\$_DIAGBUF* | | | 76 |
| IRP\$_SEQNUM* | | | 80 |
| IRP\$_EXTEND | | | 84 |
| IRP\$_ARB* | | | 88 |
| IRP\$_KEYDESC* | | | 92 |
| reserved (72 bytes) | | | 96 |

ZK-6638-HC

Data Structures

A.10 I/O Request Packet (IRP)

Table A-11 Contents of an I/O Request Packet

| Field Name | Contents |
|----------------|---|
| IRP\$L_IOQFL | I/O queue forward link. EXE\$INSERTIRP reads and writes this field when the routine inserts IRPs into a pending-I/O queue. IOC\$REQCOM reads and writes this field when the routine dequeues IRPs from a pending-I/O queue in order to send an IRP to a device driver. |
| IRP\$L_IOQBL | I/O queue backward link. EXE\$INSERTIRP and IOC\$REQCOM read and write these fields. |
| IRP\$W_SIZE* | Size of IRP. EXE\$QIO writes the symbolic constant IRP\$C_LENGTH into this field when the routine allocates and fills an IRP. |
| IRP\$B_TYPE* | Type of data structure. EXE\$QIO writes the symbolic constant DYN\$C_IRP into this field when the routine allocates and fills an IRP. |
| IRP\$B_RMOD* | Information used by I/O postprocessing. This field contains the same bit fields as the ACB\$B_RMOD field of an AST control block. For instance, the two bits defined at ACB\$V_MODE indicate the access mode of the process at time of the I/O request. EXE\$QIO obtains the processor access mode from the PSL and writes the value into this field. |
| IRP\$L_PID* | Process identification of the process that issued the I/O request. EXE\$QIO obtains the process identification from the PCB and writes the value into this field. |
| IRP\$L_AST* | Address of AST routine, if specified by the process in the I/O request. (This field is otherwise clear.) If the process specifies an AST routine address in the \$QIO call, EXE\$QIO writes the address in this field. During I/O postprocessing, the special kernel-mode AST routine queues a user mode AST to the requesting process if this field contains the address of an AST routine. |
| IRP\$L_ASTPRM* | Parameter sent as an argument to the AST routine specified by the user in the I/O request. If the process specifies an AST routine and a parameter to that AST routine in the \$QIO call, EXE\$QIO writes the parameter in this field. During I/O postprocessing, the special kernel-mode AST routine queues a user mode AST if the IRP\$L_AST field contains an address, and passes the value in IRP\$L_ASTPRM to the AST routine as an argument. |
| IRP\$L_WIND* | Address of window control block (WCB) that describes the file being accessed in the I/O request. EXE\$QIO writes this field if the I/O request refers to a file-structured device. An ACP or XQP reads this field. When a process gains access to a file on a file-structured device or creates a logical link between a file and a process I/O channel, the device ACP or XQP creates a WCB that describes the virtual-to-logical mapping of the file data on the disk. EXE\$QIO stores the address of this WCB in the IRP\$L_WIND field. |
| IRP\$L_UCB* | Address of UCB for the device assigned to the process's I/O channel. EXE\$QIO copies this value from the CCB. |
| IRP\$W_FUNC | I/O function code that identifies the function to be performed for the I/O request. The I/O request call specifies an I/O function code; EXE\$QIO and driver FDT routines map the code value to its most basic level (virtual → logical → physical) and copy the reduced value into this field. Based on this function code, EXE\$QIO calls FDT action routines to preprocess an I/O request. Six bits of the function code describe the basic function. The remaining 10 bits modify the function. |

Data Structures

A.10 I/O Request Packet (IRP)

Table A-11 (Cont.) Contents of an I/O Request Packet

| Field Name | Contents |
|----------------|--|
| IRP\$B_EFN* | Event flag number and group specified in I/O request. If the I/O request call does not specify an event flag number, EXE\$QIO uses event flag 0 by default. EXE\$QIO writes this field. The I/O postprocessing routine calls SCH\$POSTEF to set this event flag when the I/O operation is complete. |
| IRP\$B_PRI* | Base priority of the process that issued the I/O request. EXE\$QIO obtains a value for this field from the process's PCB. EXE\$INSERTIRP reads this field to insert an IRP into a priority-ordered pending-I/O queue. |
| IRP\$L_IOSB* | Virtual address of the process's I/O status block (IOSB) that receives final status of the I/O request at I/O completion. EXE\$QIO writes a value into this field if the I/O request call specifies an IOSB address. (This field is otherwise clear.) The I/O postprocessing special kernel-mode AST routine writes two longwords of I/O status into the IOSB after the I/O operation is complete. When an FDT routine aborts an I/O request by calling EXE\$ABORTIO, EXE\$ABORTIO fills the IRP\$L_IOSB field with zeros so that I/O postprocessing does not write status into the IOSB. |
| IRP\$W_CHAN* | Index number of process I/O channel for request. EXE\$QIO writes this field. |
| IRP\$W_STS | Status of I/O request. EXE\$QIO initializes this field to 0. EXE\$QIO, FDT routines, and driver fork processes modify this field according to the current status of the I/O request. I/O postprocessing reads this field to determine what sort of postprocessing is necessary (for example, deallocate system buffers and adjust quota usage). Bits in the IRP\$W_STS field describe the type of I/O function, as follows: |
| IRP\$V_BUFIO | Buffered-I/O function |
| IRP\$V_FUNC | Read function |
| IRP\$V_PAGIO | Paging-I/O function |
| IRP\$V_COMPLX | Complex-buffered-I/O function |
| IRP\$V_VIRTUAL | Virtual-I/O function |
| IRP\$V_CHAINED | Chained-buffered-I/O function |
| IRP\$V_SWAPIO | Swapping-I/O function |
| IRP\$V_DIAGBUF | Diagnostic buffer is present |
| IRP\$V_PHYSIO | Physical-I/O function |
| IRP\$V_TERMIO | Terminal I/O (for priority increment calculation) |
| IRP\$V_MBXIO | Mailbox-I/O function |
| IRP\$V_EXTEND | An extended IRP is linked to this IRP |
| IRP\$V_FILACP | File ACP I/O |
| IRP\$V_MVIRP | Mount-verification I/O function |
| IRP\$V_SRVIO | Server-type I/O |
| IRP\$V_KEY | Encrypted function (encryption key address at IRP\$L_KEYDESC) |

Data Structures

A.10 I/O Request Packet (IRP)

Table A-11 (Cont.) Contents of an I/O Request Packet

| Field Name | Contents | | | | | | | | | | |
|-----------------------|--|-----------------------|--------------------------------------|---------------------|------------------------------------|--------------|---------------------|-----------------|-------------------------------------|--------------|---|
| IRP\$L_SVAPTE | <p>For a <i>direct-I/O</i> transfer, virtual address of the first page-table entry (PTE) of the I/O-transfer buffer, written here by the FDT routine locking process pages; for <i>buffered-I/O</i> transfer, address of a buffer in system address space, written here by the FDT routine allocating buffer.</p> <p>IOC\$INITIATE copies this field into UCB\$L_SVAPTE before transferring control to a device driver start-I/O routine.</p> <p>I/O postprocessing uses this field to deallocate the system buffer for a buffered-I/O transfer or to unlock pages locked for a direct-I/O transfer.</p> | | | | | | | | | | |
| IRP\$W_BOFF | <p>Byte offset into the first page of a direct-I/O transfer. FDT routines calculate this offset and write the field.</p> <p>For buffered-I/O transfers, FDT routines must write the number of bytes to be charged to the process in this field because these bytes are being used for a system buffer.</p> <p>IOC\$INITIATE copies this field into UCB\$W_BOFF before calling a device driver start-I/O routine.</p> <p>I/O postprocessing uses IRP\$W_BOFF in conjunction with IRP\$L_BCNT and IRP\$L_SVAPTE to unlock pages locked for direct I/O. For buffered I/O, I/O postprocessing adds the value of IRP\$W_BOFF to the process byte count quota.</p> | | | | | | | | | | |
| IRP\$L_BCNT | <p>Byte count of the I/O transfer. FDT routines calculate the count value and write the field. IOC\$INITIATE copies the low-order word of this field into UCB\$W_BCNT before calling a device driver's start-I/O routine.</p> <p>For a buffered-I/O-read function, I/O postprocessing uses IRP\$L_BCNT to determine how many bytes of data to write to the user's buffer.</p> <p>The field IRP\$W_BCNT points to the low-order word of this field to provide compatibility with previous versions of VMS.</p> | | | | | | | | | | |
| IRP\$W_STS2 | <p>Second word of I/O request status. EXE\$QIO initializes this field to 0. EXE\$QIO, FDT routines, and driver fork processes modify this field according to the current status of the I/O request.</p> <p>Bits in the IRP\$W_STS2 field describe the type of I/O function, as follows:</p> <table border="0"> <tr> <td>IRP\$V_START_PAST_HWM</td> <td>I/O starts past file highwater mark.</td> </tr> <tr> <td>IRP\$V_END_PAST_HWM</td> <td>I/O ends past file highwater mark.</td> </tr> <tr> <td>IRP\$V_ERASE</td> <td>Erase I/O function.</td> </tr> <tr> <td>IRP\$V_PART_HWM</td> <td>Partial file highwater mark update.</td> </tr> <tr> <td>IRP\$V_LCKIO</td> <td>Locked I/O request, as used by DECnet direct I/O.</td> </tr> </table> | IRP\$V_START_PAST_HWM | I/O starts past file highwater mark. | IRP\$V_END_PAST_HWM | I/O ends past file highwater mark. | IRP\$V_ERASE | Erase I/O function. | IRP\$V_PART_HWM | Partial file highwater mark update. | IRP\$V_LCKIO | Locked I/O request, as used by DECnet direct I/O. |
| IRP\$V_START_PAST_HWM | I/O starts past file highwater mark. | | | | | | | | | | |
| IRP\$V_END_PAST_HWM | I/O ends past file highwater mark. | | | | | | | | | | |
| IRP\$V_ERASE | Erase I/O function. | | | | | | | | | | |
| IRP\$V_PART_HWM | Partial file highwater mark update. | | | | | | | | | | |
| IRP\$V_LCKIO | Locked I/O request, as used by DECnet direct I/O. | | | | | | | | | | |
| IRP\$L_IOST1 | <p>First I/O status longword. IOC\$REQCOM and EXE\$FINISHIO(C) write the contents of R0 into this field. The I/O postprocessing routine copies the contents of this field into the user's IOSB.</p> <p>EXE\$ZEROPARM copies a 0 and EXE\$ONEPARM copies p1 into this field. This field is a good place to put a \$QIO request argument (p1 through p6) or a computed value.</p> <p>This field is also called IRP\$L_MEDIA.</p> | | | | | | | | | | |

Data Structures

A.10 I/O Request Packet (IRP)

Table A-11 (Cont.) Contents of an I/O Request Packet

| Field Name | Contents | | | | | | |
|----------------|--|------------|--|----------|-----------------|-----------|---------------------------------|
| IRP\$_IOST2 | <p>Second I/O status longword. IOC\$REQCOM, EXE\$FINISHIO, and EXE\$FINISHIOC write the contents of R1 into this field. The I/O postprocessing routine copies the contents of this field into the user's IOSB.</p> <p>The low byte of this field is also known as IRP\$_CARCON. IRP\$_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.</p> | | | | | | |
| IRP\$_ABCNT | <p>Accumulated bytes transferred in virtual I/O transfer. IOC\$IOPOST reads and writes this field after a partial virtual transfer.</p> <p>The symbol IRP\$_W_ABCNT points to the low-order word of this field to provide compatibility with previous versions of VMS.</p> | | | | | | |
| IRP\$_OBCNT | <p>Original transfer byte count in a virtual I/O transfer. IOC\$IOPOST reads this field to determine whether a virtual transfer is complete, or whether another I/O request is necessary to transfer the remaining bytes.</p> <p>The symbol IRP\$_W_OBCNT points to the low-order word of this field to provide compatibility with previous versions of VMS.</p> | | | | | | |
| IRP\$_SEGVBN | <p>Virtual block number of the current segment of a virtual I/O transfer. IOC\$IOPOST writes this field after a partial virtual transfer.</p> | | | | | | |
| IRP\$_DIAGBUF* | <p>Address of a diagnostic buffer in system address space. If the I/O request call specifies a diagnostic buffer and if a diagnostic buffer length is specified in the DDT, and if the process has diagnostic privilege, EXE\$QIO copies the buffer address into this field.</p> <p>EXE\$QIO allocates a diagnostic buffer in system address space to be filled by IOC\$DIAGBUFILL during I/O processing. During I/O postprocessing, the special kernel-mode AST routine copies diagnostic data from the system buffer into the process diagnostic buffer.</p> | | | | | | |
| IRP\$_SEQNUM* | <p>I/O transaction sequence number. If an error is logged for the request, this field contains the universal error log sequence number.</p> | | | | | | |
| IRP\$_EXTEND | <p>Address of an IRPE linked to this IRP. FDT routines write an extension address to this field when a device requires more context than the IRP can accommodate. This field is read by IOC\$IOPOST. IRP\$_V_EXTEND in IRP\$_W_STS is set if this extension address is used.</p> | | | | | | |
| IRP\$_ARB* | <p>Address of access rights block (ARB). This block is located in the PCB and contains the process privilege mask and UIC, which are set up as follows:</p> <table style="margin-left: 2em;"> <tr> <td>ARB\$_PRIV</td> <td>Quadword containing process privilege mask</td> </tr> <tr> <td>SPARE\$_</td> <td>Unused longword</td> </tr> <tr> <td>ARB\$_UIC</td> <td>Longword containing process UIC</td> </tr> </table> | ARB\$_PRIV | Quadword containing process privilege mask | SPARE\$_ | Unused longword | ARB\$_UIC | Longword containing process UIC |
| ARB\$_PRIV | Quadword containing process privilege mask | | | | | | |
| SPARE\$_ | Unused longword | | | | | | |
| ARB\$_UIC | Longword containing process UIC | | | | | | |
| IRP\$_KEYDESC | <p>Address of encryption key.</p> | | | | | | |

A.11 I/O Request Packet Extension (IRPE)

I/O request packet extensions (IRPEs) hold additional I/O request information for devices that require more context than the standard IRP can accommodate. IRP extensions are also used when more than one buffer (region) must be locked into memory for a direct-I/O operation, or when a transfer requires a buffer that is larger than 64K. An IRPE provides space for two buffer regions, each with a 32-bit byte count.

Data Structures

A.11 I/O Request Packet Extension (IRPE)

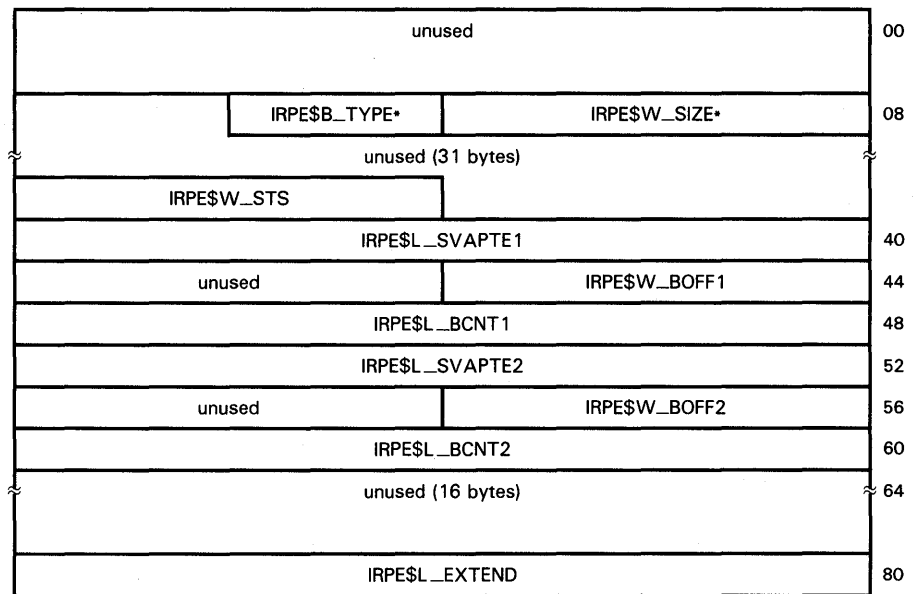
FDT routines allocate IRPEs by calling EXE\$ALLOCIRP. Driver routines link the IRPE to the IRP, store the IRPE's address in IRP\$L_EXTEND and set the bit field IRP\$V_EXTEND in IRP\$W_STS to show that an IRPE exists for the IRP. The FDT routine initializes the contents of the IRPE. Any fields within the extension not described in Table A-12 can store driver-dependent information.

If the IRP extension specifies additional buffer regions, the FDT routine must use those buffer locking routines that perform coroutine calls back to the driver if the locking procedure fails (EXE\$READLOCKR, EXE\$WRITELOCKR, and EXE\$MODIFYLOCKR). If an error occurs during the locking procedure, the driver must unlock all previously locked regions using MMG\$UNLOCK and deallocate the IRPE before returning to the buffer locking routine.

IOC\$IOPOST automatically unlocks the pages in region 1 (if defined) and region 2 (if defined) for all the IRPEs linked to the IRP undergoing completion processing. IOC\$IOPOST also deallocates all the IRPEs.

The I/O request packet extension is illustrated in Figure A-13 and described in Table A-12.

Figure A-13 I/O Request Packet Extension (IRPE)



ZK-6639-HC

Table A-12 Contents of the I/O Request Packet Extension

| Field Name | Contents |
|-----------------|--|
| IRPE\$W_SIZE* | Size of IRPE. EXE\$ALLOCIRP writes the constant IRP\$C_LENGTH to this field. |
| IRPE\$B_TYPE* | Type of data structure. EXE\$ALLOCIRP writes the constant DYN\$C_IRP to this field. |
| IRPE\$W_STS | IRPE status field. If bit IRPE\$V_EXTENDIRPE is set, it indicates that another IRPE is linked to this one. |
| IRPE\$L_SVAPTE1 | System virtual address of the page-table entry (PTE) that maps the start of region 1. FDT routines write this field. If the region is not defined, this field is zero. |
| IRPE\$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 PTE that maps the start of region 2. Set by FDT routines. This field contains a value of zero if region 2 is not defined. |
| IRPE\$W_BOFF2 | Byte offset of region 2. This field is set by FDT routines. |
| IRPE\$L_BCNT2 | Size in bytes of region 2. FDT routines write this field. |
| IRPE\$L_EXTEND | Address of next IRPE for this IRP, if any. |

A.12 Object Rights Block (ORB)

The object rights block (ORB) is a data structure that describes the rights a process must have in order to access the object with which the ORB is associated.

The ORB is usually allocated when the device is connected by means of SYSGEN's CONNECT command. SYSGEN also sets the address of the ORB in UCB\$L_ORB at that time.

The object rights block is illustrated in Figure A-14 and described in Table A-13.

Data Structures

A.12 Object Rights Block (ORB)

Figure A-14 Object Rights Block (ORB)

| | | | |
|-----------------|----------------------------|-------------|------|
| ORB\$_OWNER | | | 00 |
| ORB\$_ACL_MUTEX | | | 04 |
| ORB\$_FLAGS | ORB\$_TYPE* | ORB\$_SIZE* | 08 |
| ORB\$_REFCOUNT | | unused | 12 |
| ORB\$_MODE_PROT | | | 16 |
| ORB\$_SYS_PROT | | | 24 |
| ORB\$_OWN_PROT | | | 28 |
| ORB\$_GRP_PROT | | | 32 |
| ORB\$_WOR_PROT | | | 36 |
| ORB\$_ACLFL | | | 40 |
| ORB\$_ACLBL | | | 44 |
| ~ | ORB\$_MIN_CLASS (20 bytes) | | ~ 48 |
| ~ | ORB\$_MAX_CLASS (20 bytes) | | ~ 68 |

ZK-6640-HC

Table A–13 Contents of Object Rights Block

| Field | Contents |
|-----------------|--|
| ORB\$_OWNER | UIC of the object's owner. |
| ORB\$_ACL_MUTEX | Mutex for the object's ACL, used to control access to the ACL for reading and writing. The driver-loading procedure initializes this field with -1. |
| ORB\$_SIZE* | Size in bytes of ORB. The driver-loading procedure writes the symbolic constant ORB\$_LENGTH into this field when it creates an ORB. |
| ORB\$_TYPE* | Type of data structure. The driver-loading procedure writes the symbolic constant DYN\$_ORB into this field when it creates an ORB. |
| ORB\$_FLAGS | Flags needed for interpreting portions of the ORB that can have alternate meanings. The following fields are defined within ORB\$_FLAGS: <ul style="list-style-type: none"> ORB\$_PROT_16 The driver-loading procedure sets this bit to 1, signifying SOGW protection. ORB\$_ACL_QUEUE This flag represents the existence of an ACL queue. The driver-loading procedure does not set this bit. ORB\$_MODE_VECTOR Use vector mode protection, not byte mode. ORB\$_NOACL This object cannot have an ACL. ORB\$_CLASS_PROT Security classification is valid. |
| ORB\$_REFCOUNT | Reference count. |
| ORB\$_MODE_PROT | Mode protection vector. The low byte of this quadword is known as ORB\$_MODE. |
| ORB\$_SYS_PROT | System protection field. The low word of this field is known as ORB\$_PROT and contains the standard SOGW protection. |
| ORB\$_OWN_PROT | Owner protection field. |
| ORB\$_GRP_PROT | Group protection field. |
| ORB\$_WOR_PROT | World protection field. |
| ORB\$_ACLFL | ACL queue forward link. If ORB\$_ACL_QUEUE is 0, this field should contain 0. This field is also known as ORB\$_ACL_COUNT and is cleared by the driver-loading procedure. |
| ORB\$_ACLBL | ACL queue backward link. If ORB\$_ACL_QUEUE is 0, this field should contain 0. This field is also known as ORB\$_ACL_DESC and is cleared by the driver-loading procedure. |
| ORB\$_MIN_CLASS | Minimum classification mask. |
| ORB\$_MAX_CLASS | Maximum classification mask. |

A.13 Spin Lock Data Structure (SPL)

The spin lock data structure records all information necessary to properly grant, release, and record the ownership of a spin lock. Each static system spin lock (including the fork locks) and device lock uses an SPL to record the IPL required for spin lock acquisition, its rank, and its owner. The spin lock structure also maintains a history of spin lock use and a variety of counters used in accounting and debugging.

Data Structures

A.13 Spin Lock Data Structure (SPL)

Static system spin locks are assembled from module LDAT and are located from a vector of longword addresses starting at SMP\$AR_SPNLKVEC. UCB\$L_DLCK contains the address of the device lock for the corresponding device unit.

The fields described in the spin lock data structure are illustrated in Figure A-15 and described in Table A-14.

Figure A-15 Spin Lock Data Structure (SPL)

| | | | | |
|-------------------------------|--------------|-----------------|------------------|----|
| SPL\$B_VEC_INX* | SPL\$B_RANK* | SPL\$B_IPL* | SPL\$B_SPINLOCK* | 00 |
| SPL\$W_WAIT_CPUS* | | SPL\$W_OWN_CNT* | | 04 |
| SPL\$B_SUBTYPE* | SPL\$B_TYPE* | SPL\$W_SIZE* | | 08 |
| SPL\$L_OWN_CPU* | | | | 12 |
| SPL\$L_OWN_PC_VEC* (32 bytes) | | | | 16 |
| SPL\$L_WAIT_PC* | | | | 48 |
| SPL\$Q_ACQ_COUNT* | | | | 52 |
| SPL\$L_BUSY_WAITS* | | | | 60 |
| SPL\$Q_SPINS* | | | | 64 |
| SPL\$L_TIMO_INT* | | | | 72 |
| SPL\$L_RLS_PC* | | | | 76 |

ZK-6641-HC

Table A-14 Contents of the Spin Lock Data Structure

| Field | Contents |
|------------------|---|
| SPL\$B_SPINLOCK* | The following fields are defined within SPL\$B_SPINLOCK: SPL\$V_INTERLOCK Spin lock access interlock. When set, this bit signifies that the spin lock is owned. <7:1> Reserved to DIGITAL. |
| SPL\$B_IPL* | IPL required for spin lock acquisition. |
| SPL\$B_RANK* | Spin lock rank. Note that the internal value of a spin lock's rank, as stored in this field, is the inverse of the spin lock's logical rank, as displayed by the System Dump Analyzer and listed in Table 3-3. For instance, the structure of a spin lock with a logical rank of 0 contains the value 31 in this field. |
| SPL\$B_VEC_INX* | Index of the next entry to be written in the spin lock PC vector index (SPL\$L_OWN_PCVEC). SPL\$B_VEC_INX is updated upon each successful acquisition or release of the spin lock. |

Data Structures

A.13 Spin Lock Data Structure (SPL)

Table A-14 (Cont.) Contents of the Spin Lock Data Structure

| Field | Contents |
|--------------------|---|
| SPL\$W_OWN_CNT* | Ownership count. This field is -1 if the spin lock is unowned, zero or positive if owned. When a processor initially acquires a spin lock, this field goes from -1 to zero. A positive ownership count signifies concurrent acquisitions by a single processor. |
| SPL\$W_WAIT_CPUS* | Number of processors waiting to obtain the spin lock. |
| SPL\$W_SIZE* | Size of spin lock data structure (SPL\$C_LENGTH). |
| SPL\$B_TYPE* | Type of data structure. VMS writes the value DYN\$C_SPL in this field when it creates the SPL data structure. |
| SPL\$B_SUBTYPE* | Spin lock subtype. This field can contain the following values: SPL\$C_SPL_SPINLOCK Static system spin lock SPL\$C_SPL_FORKLOCK Fork lock SPL\$C_SPL_DEVICELOCK Device lock (dynamic spin lock) |
| SPL\$L_OWN_CPU* | Physical ID of owner CPU. This field is initialized to -1. Upon a successful acquisition, VMS copies the physical ID of the acquiring processor from CPU\$L_PHY_CPUID to this field. |
| SPL\$L_OWN_PC_VEC* | Last eight calling PCs of acquirers and releasers of the spin lock. SPL\$B_VEC_INX serves as the index of the next vector to be written in this array. |
| SPL\$L_WAIT_PC* | Last busy-wait PC. |
| SPL\$Q_ACQ_COUNT* | Count of successful acquisitions. |
| SPL\$L_BUSY_WAITS* | Count of failed acquisitions. |
| SPL\$Q_SPINS* | Count of number of spins. |
| SPL\$L_TIMO_INT* | Timeout interval before a spin lock acquisition attempt fails. |
| SPL\$L_RLS_PC* | PC of the last unconditional release of a set of nested acquisitions of the spin lock. |

A.14 Unit Control Block (UCB)

The unit control block (UCB) is a variable-length block that describes a single device unit. Each device unit on the system has its own UCB. The UCB describes or provides pointers to the device type, controller, driver, device status, and current I/O activity.

During autoconfiguration, the driver-loading procedure creates one UCB for each device unit in the system. A privileged system user can request the driver-loading procedure to create UCBs for additional devices with the SYSGEN command CONNECT, as described in Chapter 15. The procedure creates UCBs of the length specified in the DPT. The driver uses UCB storage located beyond the standard UCB fields for device-specific data and temporary driver storage.

The driver-loading procedure initializes some static UCB fields when it creates the block. VMS and device drivers can read and modify all nonstatic fields of the UCB. The UCB fields that are present for all devices are illustrated in Figure A-17 and described in Table A-16. The length of the basic UCB is defined by the symbol UCB\$K_LENGTH.

Data Structures

A.14 Unit Control Block (UCB)

UCBs are variable in length depending on the type of device and whether the driver performs error logging for the device. VMS defines a number of UCB extensions in the data structure definition macro \$UCBDEF and defines a terminal device extension in \$TTYUCBDEF. Table A-15 lists those extensions that are most often used by device drivers, indicating where each is described in this appendix. Note that use of the dual-path extension is reserved to DIGITAL; its contents should remain zero.

Table A-15 UCB Extensions and Sizes Defined in \$UCBDEF

| Extension | Used by | Size | Figure | Table |
|---------------------------------|---------------------------------|--|-------------------|-------|
| Base UCB | All devices | UCB\$K_SIZE | A-17 | A-16 |
| Error log extension | All disk and tape devices | UCB\$K_ERL_LENGTH | A-18 | A-17 |
| Dual-path extension | Reserved to DIGITAL | UCB\$K_DP_LENGTH (UCB\$K_2P_LENGTH) | — | — |
| Local tape extension | All tape devices | UCB\$K_LCL_TAPE_LENGTH | A-19 | A-18 |
| Local disk extension | All disk devices | UCB\$K_LCL_DISK_LENGTH | A-20 | A-19 |
| Terminal extension ¹ | Terminal class and port drivers | UCB\$K_TT_LENGTH | A-21 ² | A-20 |

¹The terminal UCB extension is defined by the data structure definition macro, \$TTYUCBDEF.

²Fields marked by asterisks indicate fields that may be written only by the VMS terminal class driver (TTDRIVER.EXE); a port driver may only read these fields.

In order to use an extended UCB, a device driver must specify its length in the **ucbsize** argument to the DPTAB macro. For instance:

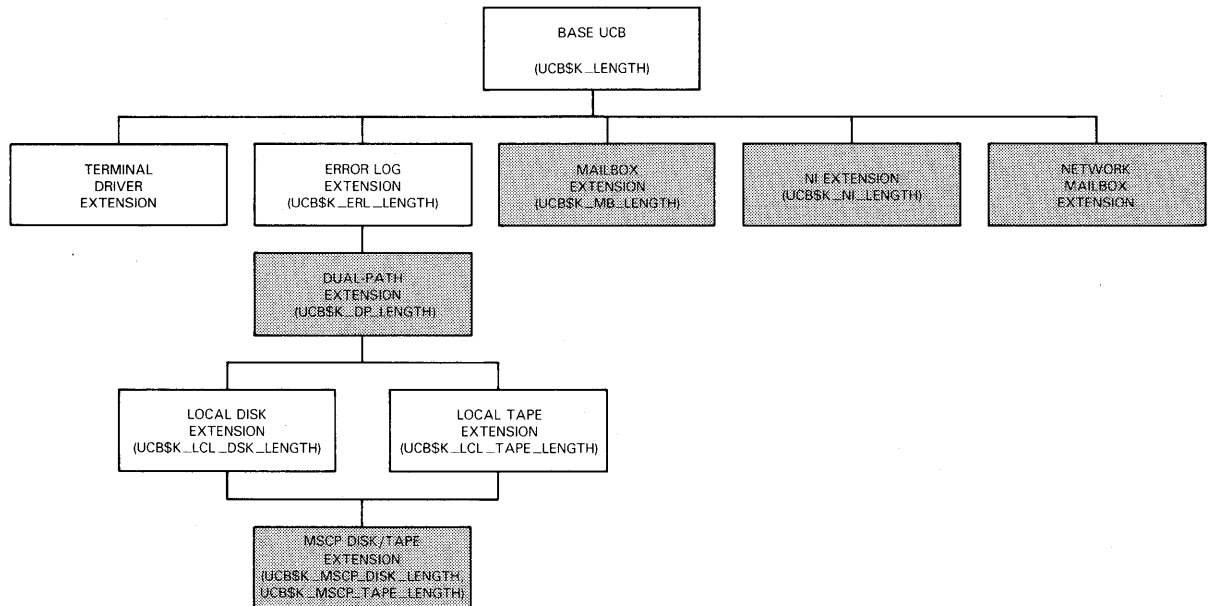
```
DPTAB    -,
         .
         .
         .
         UCBSIZE=UCB$K_LCL_TAPE_LENGTH, -
         .
         .
```

As represented in Figure A-16, each UCB extension used in a disk or tape driver builds upon the base UCB structure and any extension \$UCBDEF defines earlier in the structure. (Note that shaded UCB extensions are reserved to DIGITAL.) For instance, if you specify a UCB size of UCB\$K_LCL_TAPE_LENGTH, the size of the resulting UCB can accommodate the base UCB, the error log extension, the dual-path extension, and the local tape extension.

Data Structures

A.14 Unit Control Block (UCB)

Figure A-16 Composition of Extended Unit Control Blocks



ZK-6620-HC

A device driver can further extend a UCB by using the \$DEFINI, \$DEF, \$DEFEND, and _VIELD macros. For instance:

```
$DEFINI UCB
.=UCB$K_LCL_DISK_LENGTH
$DEF UCB$W_XX_FIELD1 .BLKW 1
$DEF UCB$W_XX_FIELD2 .BLKW 1
$DEF UCB$L_XX_FLAGS .BLKL 1
_VIELD UCB,0,<-
<XX_BIT1,,M>,-
<XX_BIT2,,M>,-
>
$DEF UCB$K_XX_LENGTH
$DEFEND UCB
```

In this case, too, the driver must ensure that it specifies the length of the extended UCB in the **ucbsize** argument of the DPTAB macro:

```
DPTAB -,
.
.
UCBSIZE=UCB$K_XX_LENGTH,-
```

Data Structures

A.14 Unit Control Block (UCB)

Figure A-17 Unit Control Block (UCB)

| | | | |
|------------------|---------------|----------------|-----|
| UCB\$_FQFL* | | | 00 |
| UCB\$_FOBL* | | | 04 |
| UCB\$_FLCK | UCB\$_TYPE* | UCB\$_SIZE* | 08 |
| UCB\$_FPC | | | 12 |
| UCB\$_FR3 | | | 16 |
| UCB\$_FR4 | | | 20 |
| UCB\$_INIQUO* | | UCB\$_BUFQUO* | 24 |
| UCB\$_ORB* | | | 28 |
| UCB\$_LOCKID* | | | 32 |
| UCB\$_CRB* | | | 36 |
| UCB\$_DLCK* | | | 40 |
| UCB\$_DDB* | | | 44 |
| UCB\$_PID* | | | 48 |
| UCB\$_LINK* | | | 52 |
| UCB\$_VCB* | | | 56 |
| UCB\$_DEVCHAR | | | 60 |
| UCB\$_DEVCHAR2 | | | 64 |
| UCB\$_AFFINITY* | | | 68 |
| reserved | | | 72 |
| UCB\$_DEVBUFSIZ | UCB\$_DEVTYPE | UCB\$_DEVCLASS | 76 |
| UCB\$_DEVDEPEND | | | 80 |
| UCB\$_DEVDEPEND2 | | | 88 |
| UCB\$_IOQFL* | | | 96 |
| UCB\$_IOQBL* | | | 100 |
| UCB\$_CHARGE* | | UCB\$_UNIT* | 104 |
| UCB\$_IRP | | | 108 |
| UCB\$_AMOD* | UCB\$_DIPL | UCB\$_REFC* | 112 |
| UCB\$_AMB* | | | 116 |
| UCB\$_STS | | | 120 |
| UCB\$_QLEN* | | UCB\$_DEVSTS | 124 |
| UCB\$_DUETIM* | | | 128 |
| UCB\$_OPCNT* | | | 132 |
| UCB\$_SVPN* | | | 136 |

ZK-6642-HC

Figure A-17 Cont'd. on next page

Data Structures

A.14 Unit Control Block (UCB)

Figure A-17 (Cont.) Unit Control Block (UCB)

| | | | |
|-----------------|--------------|--------------|-----|
| UCB\$_SVAPTE* | | | 140 |
| UCB\$_BCNT | UCB\$_BOFF | | 144 |
| UCB\$_ERRCNT | UCB\$_ERTMAX | UCB\$_ERTCNT | 148 |
| UCB\$_PDT* | | | 152 |
| UCB\$_DDT* | | | 156 |
| UCB\$_MEDIA_ID* | | | 160 |

ZK-6643-HC

Table A-16 Contents of Unit Control Block

| Field Name | Contents |
|-------------|---|
| UCB\$_FOFL* | Fork queue forward link. The link points to the next entry in the fork queue. EXE\$IOFORK and 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\$_FQBL* | Fork queue backward link. The link points to the previous entry in the fork queue. EXE\$IOFORK and VMS resource management routines write this field. |
| UCB\$_SIZE* | Size of UCB. The DPT of every driver must specify a value for this field. The driver-loading procedure uses the value to allocate space for a UCB and stores the value in each UCB created. Extra space beyond the standard bytes in a UCB (UCB\$_LENGTH) is for device-specific data and temporary storage. |
| UCB\$_TYPE* | Type of data structure. The driver-loading procedure writes the constant DYN\$_UCB into this field when the procedure creates the UCB. |
| UCB\$_FLCK | <p>Index of the fork lock that synchronizes access to this UCB at fork level. The DPT of every driver must specify a value for this field. The driver-loading procedure writes the value in the UCB when the procedure creates the UCB. All devices that are attached to a single I/O adapter and actively compete for shared adapter resources and/or a controller data channel must specify the same value for this field.</p> <p>When VMS creates a driver fork process to service an I/O request for a device, the fork process gains control at the IPL associated with the fork lock, holding the fork lock itself in a VMS multiprocessing environment. When the driver creates a fork process after an interrupt, VMS inserts the fork block into a processor-specific fork queue based on this fork IPL. A VMS fork dispatcher, executing at fork IPL, obtains the fork lock (if necessary), dequeues the fork block, and restores control to the suspended driver fork process.</p> <p>This field is also known as UCB\$_FIPL. Drivers designed to execute exclusively in a VMS uniprocessing environment store the fork IPL associated with the UCB in this field.</p> |

Data Structures

A.14 Unit Control Block (UCB)

Table A-16 (Cont.) Contents of Unit Control Block

| Field Name | Contents |
|---------------|---|
| UCB\$_FPC | <p>Fork process driver PC address. When a 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 VMS routine that reactivates a suspended driver transfers control to the saved PC address.</p> <p>VMS routines that suspend driver processing include EXE\$IOFORK, IOC\$REQxCHANy, IOC\$REQMAPREG, IOC\$REQALTMAP, IOC\$REQDATAP, and IOC\$WFIKPCH. Routines that reactivate suspended drivers include IOC\$RELCHAN, IOC\$RELMAPREG, IOC\$RELALTMAP, IOC\$RELDATAP, EXE\$FORKDSPTH, 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\$_FR3 | <p>Value of R3 at the time that a VMS routine suspends a driver fork process. The value of R3 is restored just before a suspended driver regains control.</p> |
| UCB\$_FR4 | <p>Value of R4 at the time that a VMS routine suspends a driver fork process. The value of R4 is restored just before a suspended driver regains control.</p> |
| UCB\$_BUFQUO* | <p>Buffered-I/O quota if the UCB represents a mailbox.</p> |
| UCB\$_INIQUO* | <p>Initial buffered-I/O quota if the UCB represents a mailbox.</p> |
| UCB\$_ORB* | <p>Address of ORB associated with the UCB. SYSGEN places the address in this field when you use SYSGEN's CONNECT command.</p> |
| UCB\$_LOCKID* | <p>Lock management lock ID of device allocation lock. A lock management lock is used for device allocation so that device allocation functions properly for cluster-accessible devices in a VAXcluster (DEV\$_V_CLU set within UCB\$_DEVCHAR2).</p> |
| UCB\$_CRB* | <p>Address of primary CRB associated with the device. The driver-loading procedure writes this field after it creates the associated CRB. Driver fork processes read this field to gain access to device registers. VMS routines use UCB\$_CRB to locate interrupt-dispatching code and the addresses of driver unit and controller initialization routines.</p> |
| UCB\$_DLCK* | <p>Address of device lock that—in a VMS multiprocessing environment—synchronizes access to device registers and those fields in the UCB accessed at device IPL. The driver-loading routine copies the address of the device lock in the CRB (CRB\$_DLCK) to this field as it creates a UCB for each device on a controller.</p> |
| UCB\$_DDB* | <p>Address of DDB associated with device. The driver-loading procedure writes this field when the procedure creates the associated UCB. VMS routines generally read the DDB field in order to locate device driver entry points, the address of a driver FDT, or the ACP associated with a given device.</p> |
| UCB\$_PID* | <p>Process identification number of the process that has allocated the device. Written by the \$ALLOC system service.</p> |
| UCB\$_LINK* | <p>Address of next UCB in the chain of UCBs attached to a single controller and associated with a DDB. The driver-loading procedure writes this field when the procedure adds the next UCB. Any VMS routine that examines the status of all devices on the system reads this field. Such routines include EXE\$TIMEOUT, IOC\$SEARCHDEV, and power failure recovery routines.</p> |

Data Structures

A.14 Unit Control Block (UCB)

Table A-16 (Cont.) Contents of Unit Control Block

| Field Name | Contents |
|----------------|---|
| UCB\$L_VCB* | Address of volume control block (VCB) that describes the volume mounted on the device. This field is written by the device's ACP and read by EXESQIOACPPKT, ACPs, and the XQP. |
| UCB\$L_DEVCHAR | <p>First longword of device characteristics bits. The DPT of every driver should specify symbolic constant values (defined by the \$DEVDEF macro in SYSS\$LIBRARY:STARLET.MLB) for this field. The driver-loading procedure writes the field when the procedure creates the UCB. The \$QIO system service reads the field to determine whether a device is spooled, file structured, shared, has a volume mounted, and so on.</p> <p>The system defines the following device characteristics:</p> <p>DEV\$V_REC Record-oriented device</p> <p>DEV\$V_CCL Carriage control device</p> <p>DEV\$V_TRM Terminal device</p> <p>DEV\$V_DIR Directory-structured device</p> <p>DEV\$V_SDI Single directory-structured device</p> <p>DEV\$V_SQD Sequential block-oriented device (magnetic tape, for example)</p> <p>DEV\$V_SPL Device spooled</p> <p>DEV\$V_OPR Operator device</p> <p>DEV\$V_RCT Device contains RCT</p> <p>DEV\$V_NET Network device</p> <p>DEV\$V_FOD File-oriented device (disk and magnetic tape, for example)</p> <p>DEV\$V_DUA Dual-ported device</p> <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 available for use</p> <p>DEV\$V_MNT Device mounted</p> <p>DEV\$V_MBX Mailbox device</p> <p>DEV\$V_DMT Device marked for dismount</p> <p>DEV\$V_ELG Error logging enabled</p> <p>DEV\$V_ALL Device allocated</p> <p>DEV\$V_FOR Device mounted as foreign (not file structured)</p> <p>DEV\$V_SWL Device software write-locked</p> <p>DEV\$V_IDV Device capable of providing input</p> <p>DEV\$V_ODV Device capable of providing output</p> <p>DEV\$V_RND Device allowing random access</p> <p>DEV\$V_RTM Real-time device</p> <p>DEV\$V_RCK Read-checking enabled</p> <p>DEV\$V_WCK Write-checking enabled</p> |

Data Structures

A.14 Unit Control Block (UCB)

Table A-16 (Cont.) Contents of Unit Control Block

| Field Name | Contents |
|-----------------|--|
| UCB\$_DEVCHAR2 | <p>Second longword of device characteristics. The DPT of every driver should specify symbolic constant values (defined by the \$DEVDEF macro in SYSS\$LIBRARY:STARLET.MLB) for this field. The driver-loading procedure writes the field when the procedure creates the UCB.</p> <p>The system defines the following device characteristics:</p> <p>DEV\$_CLU Device available clusterwide</p> <p>DEV\$_DET Detached terminal</p> <p>DEV\$_RTT Remote-terminal UCB extension</p> <p>DEV\$_CDP Dual-pathed device with two UCBs</p> <p>DEV\$_2P Two paths known to device</p> <p>DEV\$_MSCP Disk or tape accessed using MSCP</p> <p>DEV\$_SSM Shadow set member</p> <p>DEV\$_SRV Served by MSCP server</p> <p>DEV\$_RED Redirected terminal</p> <p>DEV\$_NNM Device name has a prefix of the format "node\$"</p> <p>DEV\$_WBC Device supports write-back caching</p> <p>DEV\$_WTC Device supports write-through caching</p> <p>DEV\$_HOC Device supports host caching</p> |
| UCB\$_AFFINITY* | <p>Bit mask of the CPU-IDs of processors in a VMS multiprocessing system that have physical connectivity to the device. Such processors can thereby access the device's registers and initiate I/O operations on the device.</p> |
| UCB\$_DEVCLASS | <p>Device class. The DPT of every driver should specify a symbolic constant (defined by the \$DCDEF macro) for this field. The driver-loading procedure writes this field when it creates the UCB.</p> <p>Drivers with set mode and device characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request.</p> <p>VMS defines the following device classes:</p> <p>DC\$_DISK Disk</p> <p>DC\$_TAPE Tape</p> <p>DC\$_SCOM Synchronous communications</p> <p>DC\$_CARD Card reader</p> <p>DC\$_TERM Terminal</p> <p>DC\$_LP Line printer</p> <p>DC\$_WORKSTATION Workstation</p> <p>DC\$_REALTIME Real time</p> <p>DC\$_BUS Bus</p> <p>DC\$_MAILBOX Mailbox</p> <p>DC\$_MISC Miscellaneous</p> <p>Note that the definition of a device as a real-time device (DC\$_REALTIME) is somewhat subjective; it implies no special treatment by VMS.</p> |

Data Structures

A.14 Unit Control Block (UCB)

Table A–16 (Cont.) Contents of Unit Control Block

| Field Name | Contents |
|------------------|---|
| UCB\$B_DEVTYPE | <p>Device type. The DPT of every driver should specify a symbolic constant (defined by the \$DCDEF macro) for this field. The driver-loading procedure writes the field when it creates the UCB.</p> <p>Drivers for devices with set mode and set characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request.</p> |
| UCB\$W_DEVBUFSIZ | <p>Default buffer size. The DPT can specify a value for this field if relevant. The driver-loading procedure writes the field when it creates the UCB.</p> <p>Drivers for devices with set mode and set characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request. This field is used by RMS for record I/O on nonfile devices.</p> |
| UCB\$Q_DEVDEPEND | <p>Device-descriptive data interpreted by the device driver itself. The DPT can specify a value for this field. The driver-loading procedure writes this field when it creates the UCB.</p> <p>Drivers for devices with set mode and set characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request.</p> |
| UCB\$Q_DEVDEPN2 | <p>Second longword for device-dependent status. This field is an extension of UCB\$Q_DEVDEPEND.</p> |
| UCB\$L_IOQFL* | <p>Pending-I/O queue listhead forward link. The queue contains the addresses of IRPs waiting for processing on a device. EXE\$INSERTIRP inserts IRPs into the pending-I/O queue when a device is busy. IOC\$REQCOM dequeues IRPs when the device is idle.</p> <p>The queue is a priority queue that has the highest priority IRPs at the front of the queue. Priority is determined by the base priority of the requesting process. IRPs with the same priority are processed first-in/first-out.</p> |
| UCB\$L_IOQBL* | <p>Pending-I/O queue listhead backward link. EXE\$INSERTIRP and IOC\$REQCOM modify the pending-I/O queue.</p> |
| UCB\$W_UNIT* | <p>Number of the physical device unit; stored as a binary value. The driver-loading procedure writes a value into this field when it creates the UCB. Drivers for multiunit controllers read this field during unit initialization to identify a unit to the controller.</p> |
| UCB\$W_CHARGE* | <p>Mailbox byte count quota charge, if the device is a mailbox.</p> |
| UCB\$L_IRP | <p>Address of IRP currently being processed on the device unit by the driver fork process. IOC\$INITIATE writes the address of an IRP into this field before the routine creates a driver fork process to handle an I/O request. From this field, a driver fork process obtains the address of the IRP being processed.</p> <p>The value contained in this field is not valid if the UCB\$V_BSY bit in UCB\$L_STS is clear.</p> |
| UCB\$W_REFC* | <p>Reference count of processes that currently have process I/O channels assigned to the device. The \$ASSIGN and \$ALLOC system services increment this field. The \$DASSGN and \$DALLOC system services decrement this field.</p> |

Data Structures

A.14 Unit Control Block (UCB)

Table A-16 (Cont.) Contents of Unit Control Block

| Field Name | Contents | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|-----------------|--|-----------|------------------|-----------|----------------------|---------------|------------------------|--------------|---------------------|--------------|--------------------|-------------|---------------------------------------|--------------|--------------------|---------------|---------------------|-----------|---------------|----------------|--------------------------|--------------|--------------------------------|-------------|-----------------------------------|--------------|----------------------------|----------------|---|----------------|---------------------------------|----------------|---|-----------------|--|
| UCB\$_DIPL | <p>Interrupt priority level (IPL) at which the device requests hardware interrupts. The DPT of every driver must specify a value for this field. The driver-loading procedure writes this field when the procedure creates the UCB. When the driver-loading procedure subsequently creates the device lock's spin lock structure (SPL), it moves the contents of this field into SPL\$_IPL.</p> <p>In a VMS uniprocessing environment, device drivers raise IPL to device IPL before reading or writing device registers or accessing other fields in the UCB synchronized at device IPL. In a VMS multiprocessing environment, drivers obtain the device lock at UCB\$_DLCK, thereby also raising IPL to device IPL in the process.</p> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_AMOD* | Access mode at which allocation occurred, if the device is allocated. Written by the \$ALLOC and \$DALLOC system services. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_AMB* | Associated mailbox UCB pointer. A spooled device uses this field for the address of its associated device. Devices that are nonshareable and not file oriented can use this field for the address of an associated mailbox. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_STS | <p>Device unit status (formerly UCB\$_W_STS). Written by drivers, IOC\$REQCOM, IOC\$CANCELIO, IOC\$INITIATE, IOC\$WFIKPCH, IOC\$WFIRLCH, EXE\$INSIOQ, and EXE\$TIMEOUT. This field is read by drivers, the \$QIO system service routines, IOC\$REQCOM, IOC\$INITIATE, and EXE\$TIMEOUT.</p> <p>This longword includes the following bits:</p> <table border="0"> <tr> <td>UCB\$_TIM</td> <td>Timeout enabled.</td> </tr> <tr> <td>UCB\$_INT</td> <td>Interrupts expected.</td> </tr> <tr> <td>UCB\$_ERLOGIP</td> <td>Error log in progress.</td> </tr> <tr> <td>UCB\$_CANCEL</td> <td>Cancel I/O on unit.</td> </tr> <tr> <td>UCB\$_ONLINE</td> <td>Device is on line.</td> </tr> <tr> <td>UCB\$_POWER</td> <td>Power has failed while unit was busy.</td> </tr> <tr> <td>UCB\$_TIMOUT</td> <td>Unit is timed out.</td> </tr> <tr> <td>UCB\$_INTTYPE</td> <td>Receiver interrupt.</td> </tr> <tr> <td>UCB\$_BSY</td> <td>Unit is busy.</td> </tr> <tr> <td>UCB\$_MOUNTING</td> <td>Device is being mounted.</td> </tr> <tr> <td>UCB\$_DEADMO</td> <td>Deallocate device at dismount.</td> </tr> <tr> <td>UCB\$_VALID</td> <td>Volume appears valid to software.</td> </tr> <tr> <td>UCB\$_UNLOAD</td> <td>Unload volume at dismount.</td> </tr> <tr> <td>UCB\$_TEMPLATE</td> <td>Template UCB from which other UCBs for this device are made. The \$ASSIGN system service checks this bit in the requested UCB and, if the bit is set, creates a UCB from the template. The new UCB is assigned instead.</td> </tr> <tr> <td>UCB\$_MNTVERIP</td> <td>Mount verification in progress.</td> </tr> <tr> <td>UCB\$_WRONGVOL</td> <td>Volume name does not match name in the VCB.</td> </tr> <tr> <td>UCB\$_DELETEUCB</td> <td>Delete this UCB when the value in UCB\$_W_REFC becomes zero.</td> </tr> </table> | UCB\$_TIM | Timeout enabled. | UCB\$_INT | Interrupts expected. | UCB\$_ERLOGIP | Error log in progress. | UCB\$_CANCEL | Cancel I/O on unit. | UCB\$_ONLINE | Device is on line. | UCB\$_POWER | Power has failed while unit was busy. | UCB\$_TIMOUT | Unit is timed out. | UCB\$_INTTYPE | Receiver interrupt. | UCB\$_BSY | Unit is busy. | UCB\$_MOUNTING | Device is being mounted. | UCB\$_DEADMO | Deallocate device at dismount. | UCB\$_VALID | Volume appears valid to software. | UCB\$_UNLOAD | Unload volume at dismount. | UCB\$_TEMPLATE | Template UCB from which other UCBs for this device are made. The \$ASSIGN system service checks this bit in the requested UCB and, if the bit is set, creates a UCB from the template. The new UCB is assigned instead. | UCB\$_MNTVERIP | Mount verification in progress. | UCB\$_WRONGVOL | Volume name does not match name in the VCB. | UCB\$_DELETEUCB | Delete this UCB when the value in UCB\$_W_REFC becomes zero. |
| UCB\$_TIM | Timeout enabled. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_INT | Interrupts expected. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_ERLOGIP | Error log in progress. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_CANCEL | Cancel I/O on unit. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_ONLINE | Device is on line. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_POWER | Power has failed while unit was busy. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_TIMOUT | Unit is timed out. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_INTTYPE | Receiver interrupt. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_BSY | Unit is busy. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_MOUNTING | Device is being mounted. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_DEADMO | Deallocate device at dismount. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_VALID | Volume appears valid to software. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_UNLOAD | Unload volume at dismount. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_TEMPLATE | Template UCB from which other UCBs for this device are made. The \$ASSIGN system service checks this bit in the requested UCB and, if the bit is set, creates a UCB from the template. The new UCB is assigned instead. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_MNTVERIP | Mount verification in progress. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_WRONGVOL | Volume name does not match name in the VCB. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| UCB\$_DELETEUCB | Delete this UCB when the value in UCB\$_W_REFC becomes zero. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Data Structures

A.14 Unit Control Block (UCB)

Table A-16 (Cont.) Contents of Unit Control Block

| Field Name | Contents |
|---------------|---|
| | UCB\$_LCL_VALID The volume on this device is valid on the local node. |
| | UCB\$_SUPMVMMSG Suppress mount-verification messages if they indicate success. |
| | UCB\$_MNTVERPND Mount verification is pending on the device and the device is busy. |
| | UCB\$_DISMOUNT Dismount in progress. |
| | UCB\$_CLUTRAN VAXcluster state transition in progress. |
| | UCB\$_WRTLOCKMV Write-locked mount verification in progress. |
| | UCB\$_SVPN_END Last byte used from page is mapped by a system virtual page number. |
| UCB\$_DEVSTS | Device-dependent status. Read and written by device drivers. The system defines the following status bits: |
| | UCB\$_JOB Job controller has been notified. |
| | UCB\$_TEMPL_BSY Template UCB is busy. |
| | UCB\$_PRMMBX Device is a permanent mailbox. |
| | UCB\$_DELMBX Mailbox is marked for deletion. |
| | UCB\$_SHMMBS Device is shared-memory mailbox. |
| | Disk drivers use bits in UCB\$_DEVSTS as follows: |
| | UCB\$_ECC ECC correction made. |
| | UCB\$_DIAGBUF Diagnostic buffer is specified. |
| | UCB\$_NOCNVRT No logical block number to media address conversion. |
| | UCB\$_DX_WRITE Console floppy write operation. |
| | UCB\$_DATACACHE Data blocks are being cached. |
| UCB\$_QLEN* | Length of pending-I/O queue (pointed to by UCB\$_IOQFL). |
| UCB\$_DUETIM* | Due time for I/O completion. Stored as the low-order 32-bit absolute time (time in seconds since the operating system was booted) at which the device will time out. IOC\$WFIKPCH and IOC\$WFIRLCH write this value when they suspend a driver to wait for an interrupt or timeout. EXE\$TIMEOUT examines this field in each UCB in the I/O database once per second. If the timeout has occurred and timeouts are enabled for the device, EXE\$TIMEOUT calls the device driver timeout handler. |
| UCB\$_OPCNT* | Count of operations completed on device unit since last bootstrap of VMS system. IOC\$REQCOM writes this field every time the routine inserts an IRP into the I/O postprocessing queue. |

Data Structures

A.14 Unit Control Block (UCB)

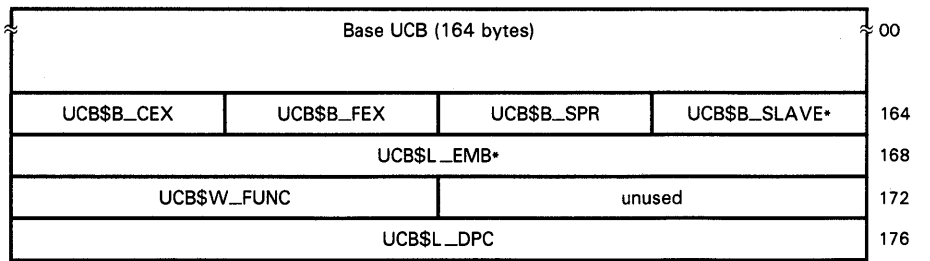
Table A-16 (Cont.) Contents of Unit Control Block

| Field Name | Contents |
|-----------------|---|
| UCB\$_SVPN* | <p>If a DPT specifies DPT\$_SVP in the flags argument to the DPTAB macro, the driver-loading procedure allocates a page of nonpaged system memory to the device. The procedure writes the system PTE's index into UCB\$_SVPN when the procedure creates the UCB.</p> <p>Disk drivers use this field for ECC error correction.</p> <p>For a <i>direct-I/O</i> transfer, the virtual address of the system PTE for the first page to be used in the transfer; for a <i>buffered-I/O</i> transfer, the virtual address of the system buffer used in the transfer.</p> <p>IOC\$INITIATE writes this field from IRP\$_SVPTE before calling a driver start-I/O routine. Drivers read this value to compute the starting address of a transfer.</p> $(\text{index} * 200_{16}) + 80000000_{16}$ |
| UCB\$_SVAPE | <p>For a <i>direct-I/O</i> transfer, the byte offset in the first page of the transfer buffer; for a <i>buffered-I/O</i> transfer, the number of bytes charged to the process for the transfer.</p> <p>IOC\$INITIATE copies this field from the IRP. Drivers read the field in calculating the starting address of a DMA transfer. If only part of a DMA transfer succeeds, the driver adjusts the value in this field to be the byte offset in the first page of the data that was not transferred.</p> |
| UCB\$_BOFF | <p>Count of bytes in the I/O transfer. IOC\$INITIATE copies this field from the IRP. Drivers read this field to determine how many bytes to transfer in an I/O operation.</p> |
| UCB\$_BCNT | <p>Error retry count of the current I/O transfer. The driver sets this field to the maximum retry count each time it begins I/O processing. Before each retry, the driver decreases the value in this field. During error logging, IOC\$REQCOM copies the value into the error message buffer.</p> |
| UCB\$_ERTCNT | <p>Maximum error retry count allowed for single I/O transfer. The DPT of some drivers specifies a value for this field. The driver-loading procedure writes the field when the procedure creates the UCB. During error logging, IOC\$REQCOM copies the value into the error message buffer.</p> |
| UCB\$_ERTMAX | <p>Number of errors that have occurred on the device since VMS booted. The driver-loading procedure initializes the field to 0 when the procedure creates the UCB. ERL\$DEVICERR and ERL\$DEVICTMO increment the value in the field and copy the value into an error message buffer. The DCL command SHOW DEVICE displays in its error count column the value contained in this field.</p> |
| UCB\$_ERRCNT | <p>Address of port descriptor table (PDT). This field is reserved for VMS SCS port drivers.</p> |
| UCB\$_PDT* | <p>Address of DDT for unit. The driver load procedure writes the contents of DDB\$_DDT for the device controller to this field when it creates the UCB.</p> |
| UCB\$_DDT* | <p>Bit-encoded media name and type, used by MSCP devices.</p> |
| UCB\$_MEDIA_ID* | |

Data Structures

A.14 Unit Control Block (UCB)

Figure A-18 UCB Error-Log Extension



ZK-6644-HC

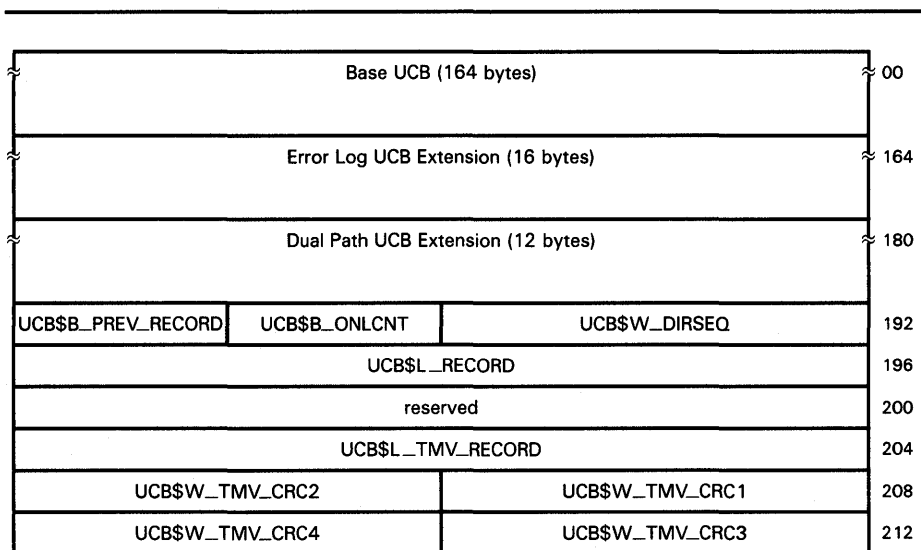
Data Structures

A.14 Unit Control Block (UCB)

Table A-17 UCB Error-Log Extension

| Field Name | Contents |
|--------------|--|
| UCB\$_SLAVE* | Unit number of slave controller. |
| UCB\$_SPR | Spare byte. This field is reserved for driver use. MASSBUS adapter drivers use this field to store a fixed offset to the MASSBUS adapter registers for the unit. |
| UCB\$_FEX | Device-specific field. This field is reserved for driver use. Certain VMS disk drivers (such as DLDRIVER in Appendix E) use this field to store an index in a hardware function dispatch table. |
| UCB\$_CEX | Device-specific field. This field is reserved for driver use. Certain VMS disk drivers (such as DLDRIVER in Appendix E) use this field to store an index into a software function case table. |
| UCB\$_EMB* | Address of error message buffer. If error logging is enabled and a device/controller error or timeout occurs, the driver calls ERL\$DEVICERR or ERL\$DEVICTMO to allocate an error message buffer and copy the buffer address into this field. IOC\$REQCOM writes final device status, error counters, and I/O request status into the buffer specified by this field. |
| UCB\$_FUNC | I/O function modifiers. This field is read and written by drivers that log errors. |
| UCB\$_DPC | Device-specific field. This field is reserved for driver use. Certain VMS disk drivers (such as DLDRIVER in Appendix E) use this field to store the driver's return PC across a dispatch to a hardware function routine. |

Figure A-19 UCB Local Tape Extension



ZK-6645-HC

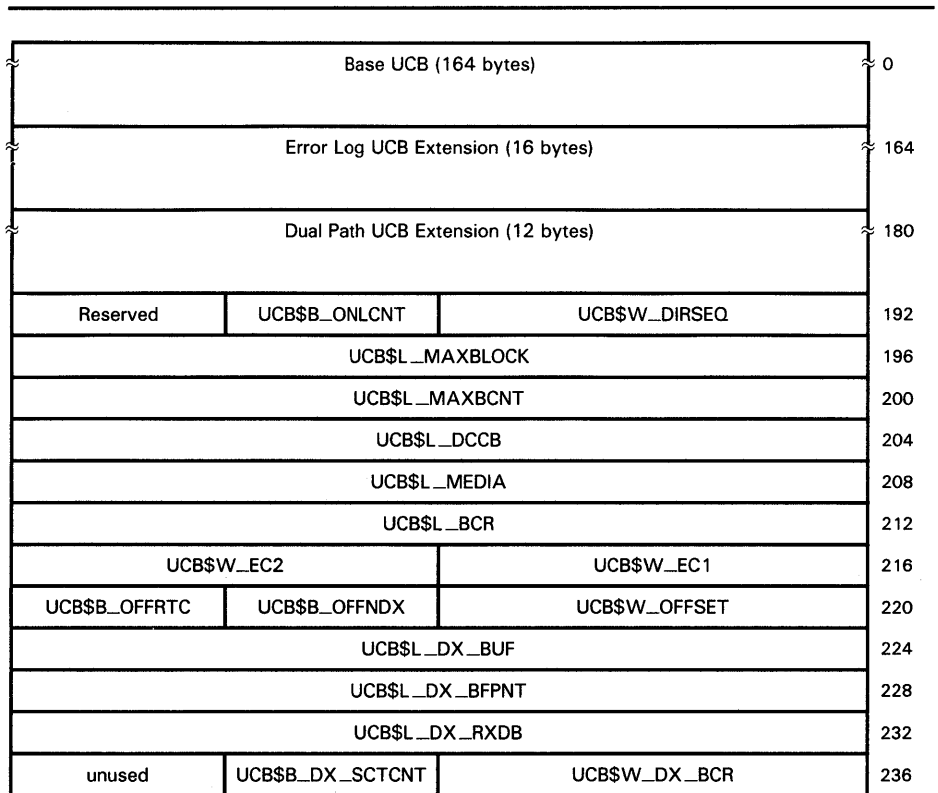
Data Structures

A.14 Unit Control Block (UCB)

Table A–18 UCB Local Tape Extension

| Field Name | Contents |
|--------------------|---|
| UCB\$W_DIRSEQ | Directory sequence number. If the high-order bit of this word, UCB\$V_AST_ARMED, is set, it indicates that the requesting process is blocking ASTs. |
| UCB\$B_ONLCNT | Number of times the device has been placed on line since VMS was last bootstrapped. |
| UCB\$B_PREV_RECORD | Tape position prior to the start of the last I/O operation. |
| UCB\$L_RECORD | Current tape position or frame counter. |
| UCB\$L_TMV_RECORD | Position following last guaranteed successful I/O operation. |
| UCB\$W_TMV_CRC1 | First CRC for mount verification's media validation. |
| UCB\$W_TMV_CRC2 | Second CRC for mount verification's media validation. |
| UCB\$W_TMV_CRC3 | Third CRC for mount verification's media validation. |
| UCB\$W_TMV_CRC4 | Fourth CRC for mount verification's media validation. |

Figure A–20 UCB Local Disk Extension



Data Structures

A.14 Unit Control Block (UCB)

Table A-19 UCB Local Disk Extension

| Field Name | Contents |
|------------------|--|
| UCB\$W_DIRSEQ | Directory sequence number. If the high-order bit of this word, UCB\$V_AST_ARMED, is set, it indicates that the requesting process is blocking ASTs. |
| UCB\$B_ONLCNT | Number of times device has been placed on line since VMS was last bootstrapped. |
| UCB\$L_MAXBLOCK | Maximum number of logical blocks on random-access device. This field is written by a disk driver during unit initialization and power recovery. |
| UCB\$L_MAXBCNT | Maximum number of bytes that can be transferred. A disk driver writes this field during unit initialization and power recovery. |
| UCB\$L_DCCB | Pointer to cache control block. |
| UCB\$L_MEDIA | Media address. |
| UCB\$L_BCR | Byte-count register. Some disk drivers use this field as an internal count of the number of bytes left to be transferred in an I/O request. The symbol UCB\$W_BCR points to the low-order word of this field. |
| UCB\$W_EC1 | ECC position register. This field records the starting bit number of an error burst. Disk driver register dumping routines copy the contents of this field into an error message or diagnostic buffer. The VMS correction routine IOC\$APPLYECC reads the contents of this field to locate the beginning of an error burst in a disk block. |
| UCB\$W_EC2 | ECC position register. Records the exclusive OR correction pattern. Disk driver register dumping routines copy the contents of this field into an error message or diagnostic buffer. The VMS ECC correction routine IOC\$APPLYECC reads the contents of this field to correct disk data. |
| UCB\$W_OFFSET | Current offset register contents. |
| UCB\$B_OFFNDX | Current offset table index. When a disk driver transfer ends in an error, the disk driver can retry the transfer a number of times with different offsets of the disk head from the centerline. This field is an index into a driver table of offset positions. |
| UCB\$B_OFFRTC | Current offset retry count. This field records the number of times to try a particular offset setting in a disk transfer retry. |
| UCB\$L_DX_BUF | Address of sector buffer (used by floppy-disk drivers). |
| UCB\$L_DX_BFPNT | Pointer to current sector (used by floppy-disk drivers). |
| UCB\$L_DX_RXDB | Address of saved receiver-data buffer (used by floppy-disk drivers). |
| UCB\$W_DX_BCR | Current floppy byte count (used by floppy-disk drivers). |
| UCB\$B_DX_SCTCNT | Current sector byte count (used by floppy-disk drivers). |

Data Structures

A.14 Unit Control Block (UCB)

Figure A-21 UCB Terminal Extension

| | | |
|----------------------|-------------------|-----|
| Base UCB (164 bytes) | | 0 |
| UCB\$_TL_CTRLY | | 164 |
| UCB\$_TL_CTRLC | | 168 |
| UCB\$_TL_OUTBAND | | 172 |
| UCB\$_TL_BANDQUE | | 176 |
| UCB\$_TL_PHYUCB | | 180 |
| UCB\$_TL_CTLPID | | 184 |
| UCB\$_TL_BRKTHRU | | 188 |
| UCB\$_TT_RDUE | | 196 |
| UCB\$_TT_RTIMOU | | 200 |
| UCB\$_TT_STATE1 | | 204 |
| UCB\$_TT_STATE2 | | 208 |
| UCB\$_TT_LOGUCB | | 212 |
| UCB\$_TT_DECHAR | | 216 |
| UCB\$_TT_DECHA1 | | 220 |
| UCB\$_TT_DECHA2 | | 224 |
| UCB\$_TT_DECHA3 | | 228 |
| UCB\$_TT_WFLINK | | 232 |
| UCB\$_TT_WBLINK | | 236 |
| UCB\$_TT_WRTBUF | | 240 |
| UCB\$_TT_MULT1 | | 244 |
| UCB\$_TT_SMLTLEN | UCB\$_TT_MULTILEN | 248 |
| UCB\$_TT_SMLT | | 252 |
| UCB\$_TT_DELFF | UCB\$_TT_DECRF | 256 |
| unused | | 260 |
| reserved | | 264 |
| UCB\$_TT_LFFILL | UCB\$_TT_CRFILL | 268 |
| unused | | 272 |
| UCB\$_TT_TYPAHD | | 276 |
| UCB\$_TT_LASTC | UCB\$_TT_LINE | 280 |
| UCB\$_TT_ESC | UCB\$_TT_FILL | 284 |
| UCB\$_TT_UNITBIT | | 288 |
| UCB\$_TT_OUTYPE | UCB\$_TT_PREMPT | 292 |
| UCB\$_TT_GETNXT | | 296 |

ZK-6647-HC

Figure A-21 Cont'd. on next page

Data Structures

A.14 Unit Control Block (UCB)

Figure A-21 (Cont.) UCB Terminal Extension

| | | | | |
|--------------------|----------------|-----------------|-----------------|-----|
| UCB\$_TT_PUTNXT | | | 300 | |
| UCB\$_TT_CLASS | | | 304 | |
| UCB\$_TT_PORT | | | 308 | |
| UCB\$_TT_OUTADR | | | 312 | |
| UCB\$_TT_PRTCTL | | UCB\$_TT_OUTLEN | 316 | |
| UCB\$_TT_DS_ST | | UCB\$_TT_DS_TX | UCB\$_TT_DS_RCV | 320 |
| UCB\$_TT_OLD | UCB\$_TT_MAINT | UCB\$_TT_DS_TIM | | 324 |
| UCB\$_TT_FBK | | | 328 | |
| UCB\$_TT_RDVERIFY | | | 332 | |
| UCB\$_TT_CLASS1 | | | 336 | |
| UCB\$_TT_CLASS2 | | | 340 | |
| UCB\$_TT_ACCPORNAM | | | 344 | |
| UCB\$_TP_MAP | | | 348 | |
| unused | | UCB\$_TP_STAT | 352 | |

ZK-6648-HC

Table A-20 UCB Terminal Extension

| Field Name | Contents |
|--------------------|---|
| UCB\$_TL_CTRLY* | Listhead of CTRL/Y AST control blocks (ACBs). |
| UCB\$_TL_CTRLC* | Listhead of CTRL/C ACBs. |
| UCB\$_TL_OUTBAND* | Out-of-band character mask. |
| UCB\$_TL_BANDQUE* | Listhead of out-of-band ACBs. |
| UCB\$_TL_PHYUCB* | Address of physical UCB. |
| UCB\$_TL_CTLPID* | Process ID of controlling process (used with SPAWN). |
| UCB\$_TL_BRKTHRU* | Facility broadcast bit mask. |
| UCB\$_TT_RDUE* | Absolute time at which a read timeout is due. |
| UCB\$_TT_RTIMOU* | Address of read timeout routine. |
| UCB\$_TT_STATE1* | First longword of terminal state information. The following fields are defined within UCB\$_TT_STATE1: |
| TTY\$_ST_POWER | Power failure |
| TTY\$_ST_CTRL | Class output |
| TTY\$_ST_FILL | Fill mode |
| TTY\$_ST_CURSOR | Cursor |
| TTY\$_ST_SENDF | Forced line feed |
| TTY\$_ST_BACKSPACE | Backspace |
| TTY\$_ST_MULTI | Multi-echo |
| TTY\$_ST_WRITE | Write in progress |

Data Structures

A.14 Unit Control Block (UCB)

Table A-20 (Cont.) UCB Terminal Extension

| Field Name | Contents |
|-------------------|---|
| | TTY\$V_ST_EOL End of line |
| | TTY\$V_ST_EDITREAD Editing read in progress |
| | TTY\$V_ST_RDVERIFY Read verify in progress |
| | TTY\$V_ST_RECALL Command recall |
| | TTY\$V_ST_READ Read in progress |
| UCB\$L_TT_STATE2* | Second longword of terminal state information. |
| | The following fields are defined within UCB\$L_TT_STATE2: |
| | TTY\$V_ST_CTRL0 Output enable |
| | TTY\$V_ST_DEL Delete |
| | TTY\$V_ST_PASALL Pass-all mode |
| | TTY\$V_ST_NOECHO No echo |
| | TTY\$V_ST_WRTALL Write-all mode |
| | TTY\$V_ST_PROMPT Prompt |
| | TTY\$V_ST_NOFLTR No control-character filtering |
| | TTY\$V_ST_ESC Escape sequence |
| | TTY\$V_ST_BADESC Bad escape sequence |
| | TTY\$V_ST_NL New line |
| | TTY\$V_ST_REFRSH Refresh |
| | TTY\$V_ST_ESCAPE Escape mode |
| | TTY\$V_ST_TYPFUL Type-ahead buffer full |
| | TTY\$V_ST_SKIPLF Skip line feed |
| | TTY\$V_ST_ESC_O Output escape |
| | TTY\$V_ST_WRAP Wrap enable |
| | TTY\$V_ST_OVRFLO Overflow condition |
| | TTY\$V_ST_AUTOP Autobaud pending |
| | TTY\$V_ST_CTRLR Clock prompt and data string from read buffer |
| | TTY\$V_ST_SKIPCRLF Skip line feed following a carriage return |
| | TTY\$V_ST_EDITING Editing operation |
| | TTY\$V_ST_TABEXPAND Expand tab characters |
| | TTY\$V_ST_QUOTING Quote character |
| | TTY\$V_ST_OVERSTRIKE Overstrike mode |
| | TTY\$V_ST_TERMNORM Standard terminator mask |
| | TTY\$V_ST_ECHAES Alternate echo string |
| | TTY\$V_ST_PRE Pre-type-ahead mode |
| | TTY\$V_ST_NINTMULTI Noninterrupt multi-echo mode |

Data Structures

A.14 Unit Control Block (UCB)

Table A–20 (Cont.) UCB Terminal Extension

| Field Name | Contents |
|---------------------|--|
| | TTY\$V_ST_RECONNECT Reconnect operation |
| | TTY\$V_ST_CTSLOW Clear-to-send low |
| | TTY\$V_ST_TABRIGHT Check for tabs to the right of the current position |
| UCB\$L_TT_LOGUCB* | Address of logical UCB, if the redirect bit is set (DEV\$V_RED in UCB\$L_DEVCHAR2). If this UCB describes the logical UCB, the contents of UCB\$L_TT_LOGUCB are zero. |
| UCB\$L_TT_DECHAR* | First longword of default device characteristics. |
| UCB\$L_TT_DECHA1* | Second longword of default device characteristics. |
| UCB\$L_TT_DECHA2* | Third longword of default device characteristics. |
| UCB\$L_TT_DECHA3* | Fourth longword of default device characteristics. |
| UCB\$L_TT_WFLINK* | Write queue forward link. |
| UCB\$L_TT_WBLINK* | Write queue backward link. |
| UCB\$L_TT_WRTBUF* | Current write buffer block. |
| UCB\$L_TT_MULTI* | Address of current multi-echo buffer. |
| UCB\$W_TT_MULTILEN* | Length of multi-echo string to be written. |
| UCB\$W_TT_SMLTLEN* | Saved length of multi-echo string. |
| UCB\$L_TT_SMLT* | Saved address of multi-echo buffer. |
| UCB\$W_TT_DESPEE* | Default speed. |
| UCB\$B_TT_DECRF* | Default carriage-return fill. |
| UCB\$B_TT_DELFF* | Default line-feed fill. |
| UCB\$B_TT_DEPARI* | Default parity/character size. |
| UCB\$B_TT_DETTYPE* | Default terminal type. |
| UCB\$W_TT_DESIZE* | Default line size. |
| UCB\$W_TT_SPEED* | Terminal line speed. This field is read and written by the class driver, and read by the port driver. It contains the following byte fields: |
| | UCB\$B_TT_TSPEED Transmit speed |
| | UCB\$B_TT_RSPEED Receive speed |
| UCB\$B_TT_CRFILL* | Number of fill characters to be output for carriage return. |
| UCB\$B_TT_LFFILL* | Number of fill characters to be output for line feed. |
| UCB\$B_TT_PARITY* | Parity, frame and stop bit information to be set when the PORT_SET_LINE service routine is called. This field is read and written by the class driver, and read by the port driver. It contains the following bit fields: |
| | UCB\$V_TT_XXPARITY Reserved to DIGITAL. |
| | UCB\$V_TT_DISPARERR Reserved to DIGITAL. |
| | UCB\$V_TT_USERFRAME Reserved to DIGITAL. |
| | UCB\$V_TT_LEN Two bits signifying character length (not counting start, stop, and parity bits), as follows: 00 ₂ = 5 bits; 01 ₂ = 6 bits; 10 ₂ = 7 bits; and 11 ₂ = 8 bits. |
| | UCB\$V_TT_STOP Number of stop bits: clear if one stop bit; set if two stop bits. |

Data Structures

A.14 Unit Control Block (UCB)

Table A–20 (Cont.) UCB Terminal Extension

| Field Name | Contents |
|--------------------|--|
| | UCB\$V_TT_PARITY Parity checking. This bit is set if parity checking is enabled. |
| | UCB\$V_TT_ODD Parity type: clear if even parity; set if odd parity. |
| UCB\$L_TT_TYPAHD* | Address of type-ahead buffer. |
| UCB\$W_TT_CURSOR* | Current cursor position. |
| UCB\$B_TT_LINE* | Current line position on page. |
| UCB\$B_TT_LASTC* | Last formatted output character. |
| UCB\$W_TT_BSPLN* | Number of back spaces to output for non-ANSI terminals. |
| UCB\$B_TT_FILL* | Current fill character count. |
| UCB\$B_TT_ESC* | Current read escape syntax state. |
| UCB\$B_TT_ESC_O* | Current write escape syntax state. |
| UCB\$B_TT_INTCNT* | Number of characters in interrupt string. |
| UCB\$W_TT_UNITBIT* | Enable and disable modem control. |
| UCB\$W_TT_HOLD | Port driver's internal flags and unit holding tank. This is read and written by the port driver, and is not accessed by the class driver. It contains the following subfields: |
| | TTY\$B_TANK_CHAR Character. |
| | TTY\$V_TANK_PREMPT Send preempt character. |
| | TTY\$V_TANK_STOP Stop output. |
| | TTY\$V_TANK_HOLD Character stored in TTY\$B_TANK_CHAR. |
| | TTY\$V_TANK_BURST Burst is active. |
| | TTY\$V_TANK_DMA DMA transfer is active. |
| UCB\$B_TT_PREMPT | Preempt character. |
| UCB\$B_TT_OUTYPE* | Amount of data to be written on a callback from the class driver. When negative, this field indicates that there is a burst of data ready to be returned; when zero, it signifies that no data is to be written; and when 1, it indicates that a single character is to be written. This field is written by the class driver and read by the port driver. |
| UCB\$L_TT_GETNXT* | Address of the class driver's input routine. This field is read by the port driver. |
| UCB\$L_TT_PUTNXT* | Address of the class driver's output routine. This field is read by the port driver. |
| UCB\$L_TT_CLASS* | Address of the class driver's vector table. This field is initialized by the CLASS_CTRL_INIT macro. The port driver reads UCB\$L_TT_CLASS whenever it must call the class driver at an entry point other than UCB\$L_TT_GETNXT or UCB\$L_TT_PUTNXT. |
| UCB\$L_TT_PORT | Address of the port driver's vector table. |
| UCB\$L_TT_OUTADR | Address of the first character of a burst of data to be written. This field is only valid when UCB\$B_TT_OUTYPE contains –1. It is read and written by the port driver, and written by the class driver. |
| UCB\$W_TT_OUTLEN | Number of characters in a burst of data to be written. This field is only valid when UCB\$B_TT_OUTYPE contains –1. It is read and written by the port driver, and written by the class driver. |

Data Structures

A.14 Unit Control Block (UCB)

Table A–20 (Cont.) UCB Terminal Extension

| Field Name | Contents |
|------------------------|--|
| UCB\$W_TT_PRTCTL | Port driver control flags. The bits in this field indicate features that are available to the port; the class driver specifies which of these features are to be enabled. The following fields are defined within UCB\$W_TT_PRTCTL. |
| TTY\$V_PC_NOTIME | No timeout. If set, the terminal class driver is not to set up timers for output. |
| TTY\$V_PC_DMAENA | DMA enabled. If set, DMA transfers are currently enabled on this port. |
| TTY\$V_PC_DMAAVL | DMA supported. If set, DMA transfers are supported for this port. |
| TTY\$V_PC_PRMMAP | Permanent map registers. If set, the port driver is to permanently allocate UNIBUS/Q22 bus map registers. |
| TTY\$V_PC_MAPAVL | Map registers available. If set, the port driver has currently allocated map registers. |
| TTY\$V_PC_XOFAVL | Auto XOFF supported. If set, auto XOFF is supported for this port. |
| TTY\$V_PC_XOFENA | Auto XOFF enabled. If set, auto XOFF is currently enabled on this port. |
| TTY\$V_PC_NOCRLF | No auto line feed. If set, a line feed is not generated following a carriage return. |
| TTY\$V_PC_BREAK | Break. If set, the port driver should generate break character; if clear, the port should turn off the break feature. |
| TTY\$V_PC_PORTFDT | FDT routine. If set, the port driver contains FDT routines. |
| TTY\$V_PC_NOMODEM | No modem. If set, the port cannot support modem operations. |
| TTY\$V_PC_NODISCONNECT | No disconnect. If set, the device cannot support virtual terminal operations. |
| TTY\$V_PC_SMART_READ | Smart read. If set, the port contains additional read capabilities. |
| TTY\$V_PC_ACCPORNAM | Access port name. If set, the port supports an access port name. |
| TTY\$V_PC_MULTISESSION | Multisession terminal. If set, the port is part of a multisession terminal. |
| UCB\$B_TT_DS_RCV | Current receive modem. |
| UCB\$B_TT_DS_TX | Current transmit modem. |
| UCB\$W_TT_DS_ST* | Current modem state. |
| UCB\$W_TT_DS_TIM* | Current modem timeout. |

Data Structures

A.14 Unit Control Block (UCB)

Table A–20 (Cont.) UCB Terminal Extension

| Field Name | Contents | | | | | | | | | | | | |
|--------------------|--|----------------|-----------------------------------|----------------|--------------------------------|-----------------|--|-----------------|--|---------------|----------------------------------|--------------|-----------------------------------|
| UCB\$B_TT_MAINT* | <p>Maintenance functions. This field is used as the argument to the port driver's PORT_MAINT routine. It is written by the class driver and read by the port driver.</p> <p>It contains several bits that allow the following maintenance functions:</p> <table style="width: 100%; border: none;"> <tr> <td style="width: 30%;">IO\$_LOOP</td> <td>Set loopback mode.</td> </tr> <tr> <td>IO\$_UNLOOP</td> <td>Reset loopback mode.</td> </tr> <tr> <td>IO\$_AUTXOF_ENA</td> <td>Enable the use of auto XON/XOFF on this line. This is the default.</td> </tr> <tr> <td>IO\$_AUTXOF_DIS</td> <td>Disable the use of auto XON/XOFF on this line.</td> </tr> <tr> <td>IO\$_LINE_OFF</td> <td>Disable interrupts on this line.</td> </tr> <tr> <td>IO\$_LINE_ON</td> <td>Reenable interrupts on this line.</td> </tr> </table> <p>Reference these bits by using the mask, shifted as follows:</p> <pre style="margin-left: 40px;">BITB #IO\$_LOOP@-7,UCB\$B_TT_MAINT(R5) ;Set loopback mode</pre> <p>UCB\$B_TT_MAINT also defines the bit UCB\$_TT_DSBL that, when set, indicates that the line has been disabled.</p> | IO\$_LOOP | Set loopback mode. | IO\$_UNLOOP | Reset loopback mode. | IO\$_AUTXOF_ENA | Enable the use of auto XON/XOFF on this line. This is the default. | IO\$_AUTXOF_DIS | Disable the use of auto XON/XOFF on this line. | IO\$_LINE_OFF | Disable interrupts on this line. | IO\$_LINE_ON | Reenable interrupts on this line. |
| IO\$_LOOP | Set loopback mode. | | | | | | | | | | | | |
| IO\$_UNLOOP | Reset loopback mode. | | | | | | | | | | | | |
| IO\$_AUTXOF_ENA | Enable the use of auto XON/XOFF on this line. This is the default. | | | | | | | | | | | | |
| IO\$_AUTXOF_DIS | Disable the use of auto XON/XOFF on this line. | | | | | | | | | | | | |
| IO\$_LINE_OFF | Disable interrupts on this line. | | | | | | | | | | | | |
| IO\$_LINE_ON | Reenable interrupts on this line. | | | | | | | | | | | | |
| UCB\$B_OLD* | The full name of this field is UCB\$B_TT_OLDPCPZORG; it currently serves as a filler byte. | | | | | | | | | | | | |
| UCB\$_TT_FBK* | Address of fallback block. | | | | | | | | | | | | |
| UCB\$_TT_RDVERIFY* | Address of read/verify table. Reserved for future use. | | | | | | | | | | | | |
| UCB\$_TT_CLASS1* | First class driver longword. | | | | | | | | | | | | |
| UCB\$_TT_CLASS2* | Second class driver longword. | | | | | | | | | | | | |
| UCB\$_TT_ACCPORNAM | Address of counted string. | | | | | | | | | | | | |
| UCB\$_TP_MAP* | UNIBUS/Q22 bus map registers. | | | | | | | | | | | | |
| UCB\$B_TP_STAT | <p>DMA port-specific status.</p> <p>The following fields are defined within UCB\$B_TP_STAT.</p> <table style="width: 100%; border: none;"> <tr> <td style="width: 30%;">TTY\$_TP_ABORT</td> <td>DMA abort requested on this line.</td> </tr> <tr> <td>TTY\$_TP_ALLOC</td> <td>Allocate map fork in progress.</td> </tr> <tr> <td>TTY\$_TP_DLLOC</td> <td>Deallocate map fork in progress.</td> </tr> </table> | TTY\$_TP_ABORT | DMA abort requested on this line. | TTY\$_TP_ALLOC | Allocate map fork in progress. | TTY\$_TP_DLLOC | Deallocate map fork in progress. | | | | | | |
| TTY\$_TP_ABORT | DMA abort requested on this line. | | | | | | | | | | | | |
| TTY\$_TP_ALLOC | Allocate map fork in progress. | | | | | | | | | | | | |
| TTY\$_TP_DLLOC | Deallocate map fork in progress. | | | | | | | | | | | | |

B **VMS Macros Invoked by Drivers**

This appendix describes VMS macros frequently used by device drivers. When referring to the macro descriptions contained herein, you should be aware of the following conventions:

- If an argument is enclosed in brackets, you can choose to include that argument or omit it.
- VMS assigns values by default to certain arguments. If you omit one of these arguments, the macro behaves as if you specified the argument with its default value. In the macro descriptions contained in this appendix, the format signifies such arguments by an equal sign (=) separating the argument from its keyword. For example:

SETIPL [ipl=31]

- If an argument takes a keyword value, you should specify the keyword value using all uppercase letters. For example:

preserve=YES
condition=RESTORE

General information about the structure of macros and their arguments in general appears in the *VAX MACRO and Instruction Set Reference Manual*.

VMS Macros Invoked by Drivers

ADPDISP

DESCRIPTION ADPDISP dispatches upon the following adapter characteristics:

| <i>select</i> | Possible Value of <i>flag</i> in <i>addrlist</i> | Definition |
|-----------------|---|--|
| ADAP_TYPE | UBA, MBA, GENBI, DR, or NULL. (See those symbols prefixed with AT\$ defined by the \$DCDEF macro in SY\$LIBRARY:STARLET.MLB.) | Adapter type. |
| ADDR_BITS | 18 or 22 | Number of adapter address bits. |
| ADAP_MAPPING | YES or NO | Does adapter support mapping? |
| AUTOPURGE_DP | YES or NO | Does adapter support autopurging datapaths? |
| BUFFERED_DP | YES or NO | Does adapter support buffered datapaths? |
| DIRECT_VECTOR | YES or NO | Does adapter directly vector device interrupts? |
| ODD_XFER_BDP | YES or NO | Does adapter support odd-aligned transfers over its buffered data paths? |
| ODD_XFER_DDP | YES or NO | Does adapter support odd-aligned transfers over its direct data paths? |
| EXTENDED_MAPREG | YES or NO | Does adapter support extended set (8,192) map registers? |
| QBUS | YES or NO | Is this a Q22 bus device? |

Specification of **select=ADAP_TYPE** causes ADPDISP to generate a CASEW instruction using ADP\$W_ADPTYPE as an index into the case table. Specification of **select=ADDR_BITS** similarly causes ADPDISP to dispatch from the contents of ADP\$B_ADDR_BITS (16 or 22 bits). If any of the other conditions is specified for **select**, ADPDISP issues a BBC or BBS instruction on the contents of bit field ADP\$V_select in ADP\$W_ADPDISP_FLAGS.

You cannot use a single invocation of ADPDISP to dispatch on more than one adapter characteristic. For example, if an autopurging datapath that supports direct vectoring is being sought, you must use the ADPDISP macro twice.

ADPDISP requires that the address of an ADP, CRB, UCB, or ECRB be specified. If anything other than an ADP is specified, the **scratch** register is used in determining the ADP address.

VMS Macros Invoked by Drivers

ADPDISP

EXAMPLES

1

```
ADPDISP -  
  SELECT=ADAP_MAPPING, -  
  ADDRLIST=<<NO,10$>, <YES,20$>>, -  
  ADPADDR=R3
```

ADPDISP transfers control to the instruction at 10\$ if the adapter does not support mapping, or to 20\$ if it does. ADPDISP uses the value in R3 to locate the ADP.

2

```
ADPDISP -  
  SELECT=ADAP_TYPE, -  
  ADDRLIST=<<CI,10$>, <MBA,20$>, <UBA,30$>>, -  
  UCBADDR=R5, -  
  SCRATCH=R1
```

ADPDISP transfers control to 10\$ if the adapter is a CI, 20\$ if the adapter is a MASSBUS adapter, and 30\$ if it is a UNIBUS adapter. ADPDISP determines the location of the ADP from a chain of pointers starting at the UCB address specified in R5. In doing so, it destroys the contents of scratch register R1.

3

```
ADPDISP -  
  SELECT=ADDR_BITS, -  
  ADDRLIST=<<18,10$>, <22,20$>>, -  
  ADPADDR=R3
```

ADPDISP transfers control to 10\$ for all adapters using an 18-bit address and 20\$ for all using a 22-bit address. The ADP address is supplied in R3.

CASE

Generates a CASE instruction and its associated table.

FORMAT **CASE** *src* ,*displist* [*,type=W*] [*,limit=#0*] [*,nmode=S#*]

PARAMETERS ***src***
Source of the index value to be used with the CASE instruction.

displist
List of destinations to which control is to be dispatched, depending on the value of the index.

[type=W]
Data type of *src* (B, W, or L).

[limit=#0]
Lower limit of the value of *src*.

[nmode=S#]
Addressing mode used to reference the case-table entries; the default, short-literal mode, is good for up to 63 entries.

EXAMPLE

```
10$:    CASE -  
          src=ITEMC,  
          displist=<FIRST,SECOND,THIRD,FOURTH>
```

This invocation of the CASE macro expands to the following code:

```
          CASEW    ITEMC,#0,S^#<<30001$-30000$>/2>-1  
30000$:            .SIGNED_WORD    FIRST-30000$  
                  .SIGNED_WORD    SECOND-30000$  
                  .SIGNED_WORD    THIRD-30000$  
                  .SIGNED_WORD    FOURTH-30000$  
30001$:
```

VMS Macros Invoked by Drivers

CLASS_CTRL_INIT

CLASS_CTRL_INIT

Generates the common code that must be executed by the controller initialization routine of all terminal port drivers.

FORMAT **CLASS_CTRL_INIT** *dpt, vector*

PARAMETERS *dpt*
Symbolic name of the port driver's driver prologue table.

vector
Address of the port driver vector table.

DESCRIPTION A terminal port driver's controller initialization routine invokes the CLASS_CTRL_INIT macro to relocate the class and port driver vector tables and perform other required initialization.

 To use the CLASS_CTRL_INIT macro, the driver must include an invocation of the \$TTYMACS definition macro (from SYS\$LIBRARY:LIB.MLB).

VMS Macros Invoked by Drivers

CLASS_UNIT_INIT

CLASS_UNIT_INIT

Generates the common code that must be executed by the unit initialization routine of all terminal port drivers.

FORMAT CLASS_UNIT_INIT

DESCRIPTION

A terminal port driver's unit initialization routine invokes the CLASS_UNIT_INIT macro to perform initialization tasks common to all port drivers. To use the CLASS_UNIT_INIT macro, the driver must include an invocation of the \$TTYMACS definition macro (from SYS\$LIBRARY:LIB.MLB).

The CLASS_UNIT_INIT macro binds the terminal port and class driver into a single, complete driver by initializing the following UCB fields as indicated:

| Field | Contents |
|-----------------|---|
| UCB\$_TT_CLASS | Class driver vector table address |
| UCB\$_TT_PORT | Port driver vector table address |
| UCB\$_TT_GETNXT | Address of the class driver's get-next-character routine (CLASS_GETNXT) |
| UCB\$_TT_PUTNXT | Address of the class driver's put-next-character routine (CLASS_PUTNXT) |
| UCB\$_DDT | Address of the terminal class driver's driver dispatch table |

Prior to invoking this macro, the unit initialization should place in R0 the address of the port driver vector table.

VMS Macros Invoked by Drivers

CPUDISP

CPUDISP

Causes a branch to a specified address according to the CPU type of the VAX processor executing the macro code.

FORMAT **CPUDISP** *addrlist* [*environ=VMS*] ,*continue=NO*

PARAMETERS ***addrlist***

List containing one or more pairs of arguments in the following format:

<CPU-type, destination>

The **CPU-type** parameter identifies the type or subtype of a VAX processor for which the macro is to generate a case table entry. The CPUDISP macro identifies the following VAX systems by type alone:

| CPU Type | VAX System |
|----------|--|
| 8PS | VAX 8830/8840 |
| 8NN | VAX 8530/8550/8700/8800 |
| 790 | VAX 8600/8650/8670 |
| 8SS | VAX 8200/8250/8300/8350 |
| 780 | VAX-11/780 and VAX-11/785 ¹ |
| 785 | VAX-11/785 |
| 750 | VAX-11/750 |
| 730 | VAX-11/730 and VAX-11/725 |
| UV1 | MicroVAX I |

¹Because the VAX-11/785 has the same CPU type as the VAX-11/780, the CPUDISP macro contains special code to distinguish between the two processors. This code tests a bit within the processor's system identification register (PR\$_SID) that indicates whether it is a VAX-11/785.

The CPUDISP macro identifies the following VAX systems by type and subtype:

| CPU Type | Subtype | VAX System |
|----------|---------|------------------------------------|
| UV | | MicroVAX II processor-based system |
| | UV2 | MicroVAX II |
| | 410 | VAXstation 2000/MicroVAX 2000 |
| CV | | CVAX processor-based system |
| | 650 | MicroVAX 3600-series system |
| | 9CC | VAX 6200-series system |

VMS Macros Invoked by Drivers

CPUDISP

You can supply any combination of generic type and subtype in a single invocation of the CPUDISP macro. Should the CPUDISP macro code be executed on the appropriate processor, the following transfers of control are possible:

- If you specify a generic type but no subtype, CPUDISP causes the branch designated for the generic type to be taken for all of its subtypes.
- If you specify one or more subtypes but not the generic type, CPUDISP causes the branch designated for each subtype to be taken.
- If you specify *both* the generic type and one or more subtypes, CPUDISP causes the branch designated for each specified subtype to be taken. For those subtypes that you do not specify, CPUDISP causes the branch designated for the generic type to be taken.

The **destination** parameter contains the address to which the code generated by the invocation of the CPUDISP macro passes control to continue with CPU-specific processing.

[*environ*=VMS]

Identification of the run-time environment of the code generated by the CPUDISP macro. There is no need to change the default value of this argument.

***continue*=NO**

Specifies whether execution should continue at the line immediately after the CPUDISP macro if the value at EXE\$GB_CPATYPE does not correspond to any of the values specified as the **CPU-type** in the **addrlist** argument. A fatal bugcheck of UNSUPRTCPU occurs if the dispatching code does not find the executing processor identified in the **addrlist** and the value of **continue** is NO.

DESCRIPTION

The CPUDISP macro provides a means for transferring control to a specified destination depending on the CPU type of the executing processor. For those processors that do not have a unique CPU type, CPUDISP also provides the means to dispatch on a particular CPU subtype.

To accomplish this, CPUDISP builds one or two case tables. The first CASEB instruction uses words in the first case table to set up a transfer based on each **CPU-type** specified in the **addrlist** argument. CPUDISP constructs the second case table in the event it encounters a CPU subtype in the **addrlist**.

CPUDISP constructs appropriate symbolic constants for each **CPU-type** listed in **addrlist**, and compares them against the contents of EXE\$GB_CPATYPE. These constants have the form PR\$_SID_TYPCPU-*type*.

For each CPU subtype it encounters in the **addrlist** argument, CPUDISP also constructs symbolic constants of the form PR\$_XSID-*xx_yyy*, where *xx* is the generic CPU type (either UV or CV) and *yyy* is the CPU subtype (UV2 or 410 for UV, or 650 or 9CC for CV). It compares the value of PR\$_XSID-*xx_yyy* against the contents of EXE\$GB_CPUDATA+15.

VMS Macros Invoked by Drivers

DDTAB

DDTAB

Generates a driver dispatch table (DDT) labeled *devnam\$DDT*.

FORMAT

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

PARAMETERS

devnam

Generic name of the device.

[*start=+IOC\$RETURN*]

Address of start-I/O routine.

[*unsolic=+IOC\$RETURN*]

Address of the routine that services unsolicited interrupts from the device. Only MASSBUS device drivers use this field.

functb

Address of the driver's function decision table.

[*cancel=+IOC\$RETURN*]

Address of cancel-I/O routine.

[*regdmp=+IOC\$RETURN*]

Address of the routine that dumps the device registers to an error message buffer or to a diagnostic buffer.

[*diagbf=0*]

Length in bytes of the diagnostic buffer.

[*erlgbf=0*]

Length in bytes of the error message buffer.

[*unitinit=+IOC\$RETURN*]

Address of unit initialization routine. MASSBUS drivers should use this field rather than *CRB\$L_INTD+VEC\$L_UNITINIT*. UNIBUS, Q22-bus, and generic VAXBI drivers can use either one.

[*altstart=+IOC\$RETURN*]

Address of alternate start-I/O routine. To initiate this routine, a driver FDT routine exits by means of VMS routine *EXE\$ALTQUEPKT* instead of *EXE\$QIODRVPKT*.

VMS Macros Invoked by Drivers

DDTAB

[mntver=+IOC\$MNTVER]

Address of the VMS routine that is called at the beginning and end of a mount verification operation. The default, IOC\$MNTVER, is suitable for all single-stream disk drives. Use of this field to call any other routine is reserved to DIGITAL.

[cloneducb=+IOC\$RETURN]

Address of routine called when a UCB is cloned by the \$ASSIGN system service.

DESCRIPTION

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

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

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

A plus sign (+) precedes the address of any specified routine that is part of VMS: that is, it is an address that is not relative to the location of the driver. No plus sign precedes the address of a routine (such as a start-I/O routine) that is part of the driver module.

EXAMPLE

```
DDTAB      -                ;DDT-creation macro
DEVNAM=XX, -                ;Name of device
START=XX_START, -          ;Start-I/O routine
FUNCTB=XX_FUNCTABLE, -    ;FDT address
CANCEL=+IOC$CANCELIO, -   ;Cancel-I/O routine
REGDMP=XX_REGDUMP, -      ;Register dumping routine
DIAGBF=<<15*4>>+<<3+5+1>*4>>, - ;Diagnostic buffer size
ERLGBF=<<15*4>>+<1*4>+<EMB$L_DV_REGSAV>> ;Error message buffer size
```

This code excerpt uses the DDTAB macro to create a driver dispatch table for the XX device type. Note that because the cancel-I/O routine is part of VMS, its address is preceded by a plus sign (+).

VMS Macros Invoked by Drivers

\$DEF

\$DEF

Defines a data-structure field within the context of a \$DEFINI macro.

FORMAT **\$DEF** *sym* [*,alloc*] [*,siz*]

PARAMETERS ***sym***
Name of the symbol by which the field is to be accessed.

[alloc]
Block-storage-allocation directives, one of the following: .BLKB, .BLKW, .BLKL, .BLKQ, or .BLKO.

[siz]
Number of block storage units to allocate.

DESCRIPTION See the descriptions of the \$DEFINI, \$DEFEND, _VIELD, and \$EQLST macros for additional information on defining symbols for data structure fields.

You can define a second symbolic name for a single field, using the \$DEF macro a second time immediately following the first definition, leaving the **alloc** argument blank in the first definition. The following example does this, equating SYNONYM2 with LABEL2:

```
$DEFINI JLB                            ;Start structure definition
$DEF LABEL1 .BLKL 1                   ;First JLB field
$DEF SYNONYM2                         ;Synonym for LABEL2 field
$DEF LABEL2 .BLKL 1                   ;Second JLB field
$DEF LABEL3 .BLKL 1                   ;Third JLB field
$DEFEND JLB                           ;End of JLB structure
```

For another example of the use of the \$DEF macro, see the description of the \$DEFINI macro.

\$DEFEND

Ends the scope of the \$DEFINI macro, thereby completing the definition of fields within a data structure.

FORMAT **\$DEFEND** *struc*

PARAMETERS *struc*
Name of the structure that is being defined.

DESCRIPTION See the descriptions of the \$DEFINI, _VIELD, and \$EQLST macros for additional information on defining symbols for data structure fields.

VMS Macros Invoked by Drivers

\$DEFINI

\$DEFINI

Begins the definition of a data structure.

FORMAT **\$DEFINI** *struc* [*,gbl=LOCAL*] [*,dot=0*]

PARAMETERS ***struc***

Name of the data structure that is being defined.

[gbl=LOCAL]

Specifies whether the symbols defined for this data structure are to be local or global symbols. The default is to make them local.

To make the definitions of symbols global, you must specify **GLOBAL** for the value of the **gbl** argument.

[dot=0]

Offset from the beginning of the data structure of the first field to be defined. The \$DEFINI macro moves this value into the current location counter (.).

DESCRIPTION

The \$DEF macro defines fields within the structure specified by the invocation of the \$DEFINI macro, and the \$DEFEND macro ends the definition. See the descriptions of the _VIELD and \$EQLST macros for additional information on defining symbols for data structure fields.

EXAMPLE

```
$DEFINI UCB, ,UCB$K_LCL_DISK_LENGTH
                                ;Start UCB extension, begin definitions
                                ; at end of local disk UCB extension
$DEF  UCB_W_DL_PBCR  .BLKW 1      ;Partial byte count
$DEF  UCB_W_DL_CS   .BLKW 1      ;Control status register
$DEF  UCB_W_DL_BA   .BLKW 1      ;Bus address register
$DEF  UCB_A_DL_BUF_PA .BLKL 1     ;Physical buffer physical address
$DEF  UCB_K_DL_LEN  .BLKW 1      ;Length of extended UCB
$DEFEND UCB
```

This code excerpt, when assembled in VMS Version 5.0, produces the following symbol listing:

```
UCB_A_DL_BUF_PA          000000D2
UCB_K_DL_LEN             000000D6
UCB$K_LCL_DISK_LENGTH   = 000000CC
UCB_W_DL_BA             000000D0
UCB_W_DL_CS             000000CE
UCB_W_DL_PBCR           000000CC
```


VMS Macros Invoked by Drivers

DEVICELOCK

- Calls either SMP\$ACQUIREL or SMP\$ACQNOIPL, depending upon the presence of **condition=NOSETIPL**. SMP\$ACQUIREL raises IPL to device IPL prior to obtaining the lock, determining appropriate IPL from the device lock's data structure (SPL\$B_IPL).

In both processing environments, the DEVICELOCK macro performs the following tasks:

- Preserves the current IPL at the specified location (if **savipl** is specified)
- Sets the SMP-modified bit in the driver prologue table (DPT\$V_SMPMOD in DPT\$L_FLAGS)

EXAMPLE

```
DEVICELOCK -
    LOCKADDR=UCB$L_DLCK(R5),- ;Lock device access
    LOCKIPL=UCB$B_DIPL(R5),- ;Raise IPL
    SAVIPL=(SP),-           ;Save current IPL
    PRESERVE=YES           ;Save R0
SETIPL #31                 ;Disable all interrupts
BBC #UCB$V_POWER,-        ;If clear - no power failure
    UCB$W_STS(R5),L1      ;...
                        ;Service power failure!
.
.
.
DEVICEUNLOCK -
    LOCKADDR=UCB$L_DLCK(R5),- ;Unlock device access
    NEWIPL=(SP)+,-         ;Restore IPL
    PRESERVE=YES          ;Save R0
BRW RETREG                ;Exit
L1:                       ;Return for no power failure
.
.
.
WFIKPCH RETREG,#2        ;Wait for interrupt
```

The start-I/O routine of DLDRIVER invokes the DEVICELOCK macro to synchronize access to the device's registers and UCB fields. Thus synchronized at device IPL, and holding the device lock in a VMS multiprocessing environment, the routine raises IPL to IPL\$_POWER (IPL 31) to check for a power failure on the local processor. If a power failure has occurred, the routine releases the device lock and pops the saved IPL from the stack before servicing the failure. If a power failure has not occurred, the routine branches to set up the I/O request. Note that, in this instance, it is the wait-for-interrupt routine, invoked by the WFIKPCH macro, that issues the DEVICEUNLOCK macro and pops the saved IPL from the stack.

VMS Macros Invoked by Drivers

DEVICEUNLOCK

EXAMPLE

```
DEVICELOCK -
    LOCKADDR=UCB$L_DLCK(R5),- ;Lock device access
    CONDITION=NOSETIPL,- ;Do not set IPL
    PRESERVE=NO ;Do not preserve R0
.
.
20$: MOVQ   UCB$L_FR3(R5),R3 ;Restore driver context
    JSB    @UCB$L_FPC(R5) ;Call driver at interrupt return address
40$: DEVICEUNLOCK -
    LOCKADDR=UCB$L_DLCK(R5),- ;Unlock device access
    PRESERVE=NO ;Do not preserve R0
```

When the device interrupts, DLDRIVER's interrupt service routine immediately obtains the device lock so that it can examine device registers and preserve their contents. It then calls the driver's start-I/O routine at the location in which it initiated device activity. The routine forks and returns control to the interrupt service routine, which releases the device lock.

DPTAB

Generates a driver prologue table (DPT) in a program section called \$\$\$105_PROLOGUE.

FORMAT **DPTAB** *end ,adapter ,[flags=0] ,ucbsize ,[unload] ,[maxunits=8] ,[defunits=1] ,[deliver] ,[vector] ,name [,psect=\$\$\$105_PROLOGUE] [,smp=NO] [,decode]*

PARAMETERS

end

Address of the end of the driver.

adapter

Type of adapter (as indicated by the symbols prefixed by AT\$ defined by the \$DCDEF macro in SYS\$LIBRARY:STARLET.MLB). The adapter type can be any of the following:

| | |
|-------|-------------------------------------|
| UBA | UNIBUS adapter or Q22 bus interface |
| MBA | MASSBUS adapter |
| GENBI | Generic VAXBI adapter |
| DR | DR device |
| NULL | No actual device for driver |

[flags=0]

Flags used in loading the driver. Drivers use the following flags:

| | |
|-----------------|---|
| DPT\$M_SVP | Indicates that the driver requires a permanently allocated system page. Disk drivers use this SPTE during ECC correction and when using the system routines IOC\$MOVFRUSER and IOC\$MOVTOUSER. When this flag is set, the driver-loading procedure allocates a permanent system page-table entry (SPTE) for the device. It stores an index to the virtual address of the SPTE in UCB\$_SVPN when it creates the UCB. A driver can calculate the system virtual address of the page corresponding to this index by using the following formula: $(index * 200_{16}) + 80000000_{16}$ |
| DPT\$M_NOUNLOAD | Indicates that the driver cannot be reloaded. When this bit is set, the driver can be unloaded only by rebooting the system. |

VMS Macros Invoked by Drivers

DPTAB

DPT\$M_SMPMOD Indicates that the driver has been designed to execute within a VMS multiprocessing environment. Use of any of the VMS multiprocessing synchronization macros (DEVICELOCK/DEVICEUNLOCK, FORKLOCK /FORKUNLOCK, or LOCK/UNLOCK) automatically sets this flag, as long as the code using the macro resides in the same module as the invocation of DPTAB.

ucbsize

Size in bytes of each UCB the driver-loading procedure creates for devices supported by the driver. This required argument allows drivers to extend the UCB to store device-dependent data describing an I/O operation. Figure A-17 describes the VMS-defined extensions to the UCB and discusses the means by which a driver can define a device-specific extension.

[unload]

Address of the driver routine invoked by the SYSGEN RELOAD command before it unloads an old version of the driver to load a new version. The driver-loading procedure calls this routine before reinitializing all controllers and device units associated with the driver.

[maxunits=8]

Maximum number of units that this driver supports on a controller. This field affects the size of the IDB created by the driver-loading procedure. If you omit the **maxunits** argument, the default is eight units. You can override the value specified in the DPT by using the /MAXUNITS qualifier to the SYSGEN CONNECT command.

[defunits=1]

Maximum number of UCBs to be created by SYSGEN's AUTOCONFIGURE command (one for each device unit to be configured). The unit numbers assigned are zero through **defunits**-1.

If you do not specify the **deliver** argument, AUTOCONFIGURE creates the number of units specified by **defunits**. If you specify the address of a unit delivery routine in the **deliver** argument, AUTOCONFIGURE calls that routine to determine whether or not to create each UCB automatically.

[deliver]

Address of the driver unit delivery routine. The unit delivery routine determines which device units supported by this driver the SYSGEN AUTOCONFIGURE command should configure automatically. If you omit the **deliver** argument, the AUTOCONFIGURE command creates the number of units specified by the **defunits** argument.

[vector]

Address of a driver-specific transfer vector. A terminal port driver specifies the address of its vector table in this argument.

name

Name of the device driver. The driver-loading procedure will permit the loading of only one copy of the driver associated with this name. A driver name can be up to 11 alphabetic characters and, by convention, is formed by appending the string DRIVER to the 2-alphabetic-character generic device name, for example, DBDRIVER.

VMS Macros Invoked by Drivers

DPTAB

[psect=\$\$\$105_PROLOGUE]

Program section in which the DPT is created. The default value of this argument is required for all non-DIGITAL-supplied device drivers.

[smp=NO]

Indication of whether the driver is suitably synchronized to execute in a VMS multiprocessing system. Note that use of any of the spin lock synchronization macros in a device driver causes the DPTAB macro to indicate multiprocessing synchronization.

[decode]

Offset to name used by workstation windowing software.

DESCRIPTION

The DPTAB macro, in conjunction with invocations of the DPT_STORE macro, creates a driver prologue table (DPT). The DPTAB macro places information in the DPT that allows the driver-loading procedure to determine the identity of the driver and the devices it supports. The DPTAB macro, in invoking the \$SPLCODDEF definition macro, also defines the spin lock indexes used in the DPT_STORE, FORKLOCK, and LOCK macros.

EXAMPLE

```

DPTAB      -                               ;DPT-creation macro
           END=XA_END,-                     ;End of driver label
           ADAPTER=UBA,-                     ;Adapter type
           FLAGS=<DPT$M_SVP!-                ;Allocate permanent SPTE
             DPT$M_SMPMOD>,-                ;Multiprocessing driver
           UCBSIZE=UCB$K_SIZE,-             ;UCB size
           NAME=XADRIVER                     ;Driver name
DPT_STORE  INIT                             ;Start of load initialization table
DPT_STORE  UCB,UCB$B_FLCK,B,-               ;Fork lock index
           SPL$C_IOLOCK8
DPT_STORE  UCB,UCB$B_DIPL,B,22              ;Device interrupt IPL
DPT_STORE  UCB,UCB$L_DEVCHAR,L,<-           ;Device characteristics
           DEV$M_AVL!-                       ;Available
           DEV$M_RTM!-                       ;Real time device
           DEV$M_ELG!-                       ;Error logging enabled
           DEV$M_IDV!-                       ;Input device
           DEV$M_ODV>                         ;Output device
DPT_STORE  UCB,UCB$B_DEVCLASS,B,-           ;Device class
           DC$REALTIME
DPT_STORE  UCB,UCB$B_DEVTYPE,B,-            ;Device type
           DT$DR11W
DPT_STORE  UCB,UCB$W_DEVBUFSIZ,W,-          ;Default buffer size
           XA_DEF_BUFSIZ
DPT_STORE  REINIT                           ;Start of reload initialization table
DPT_STORE  DDB,DDB$L_DDT,D,XA$DDT           ;Address of DDT
DPT_STORE  CRB,CRB$L_INTD+VEC$L_ISR,D,-     ;Address of interrupt service routine
           XA_INTERRUPT
DPT_STORE  CRB,CRB$L_INTD+VEC$L_INITIAL,D,-  ;Address of controller initialization routine
           XA_CONTROL_INIT
DPT_STORE  END                               ;End of initialization

```

This excerpt from XADRIVER.MAR contains the DPTAB macro and the series of DPT_STORE macros that create its driver prologue table.

VMS Macros Invoked by Drivers

DPT_STORE

DPT_STORE

Instructs the VMS driver-loading procedure to store values in a table or data structure.

FORMAT **DPT_STORE** *str_type ,str_off ,oper ,exp [,pos] [,size]*

PARAMETERS ***str_type***
Type of data structure (CRB, DDB, IDB, ORB, or UCB) into which the driver-loading procedure is to store the specified data, or a label denoting a table marker. Table marker labels indicate the start of a list of DPT_STORE macro invocations that store information for the driver-loading procedure in the driver initialization table and driver reinitialization table sections of the DPT. If this argument is a table marker label, no other argument is allowed. The following labels are used:

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

str_off
Unsigned offset into the data structure in which the data is to be stored. This value cannot be more than 65,535 bytes.

oper
Type of storage operation, one of the following:

| Type | Meaning |
|------|---|
| B | Write a byte value. |
| W | Write a word value. |
| L | Write a longword value. |
| D | Write an address relative to the beginning of the driver. |
| V | Write a bit field. If you specify a V in the oper argument, the driver-loading procedure uses the exp , pos , and size arguments as operands to an INSV instruction. |

If an at sign (@) precedes the **oper** argument, the **exp** argument indicates the address of the data that is to be stored and not the data itself.

VMS Macros Invoked by Drivers

DPT_STORE

exp

Expression indicating the value with which the driver-loading procedure is to initialize the indicated field. If an at-sign character (@) precedes the **oper** argument, the **exp** argument indicates the address of the data with which to initialize the field. For example, the following macro indicates that the contents of the location DEVICE_CHARS are to be written into the DEVCHAR field of the UCB.

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

[pos]

Starting bit position within the specified field; used only if **oper=V**.

[size]

Number of bits to be written; used only if **oper=V**.

DESCRIPTION

The DPT_STORE macro places information in the DPT that the driver-loading procedure uses to load specified values into specified fields. The DPT_STORE macro accepts two lists of fields:

- Fields to be initialized only when a driver is first loaded
- Fields to be initialized when a driver is first loaded and reinitialized if the driver is reloaded

The DPTAB macro stores the relative addresses of these two lists, called initialization and reinitialization tables, in the DPT. A driver constructs the initialization tables by following the DPTAB macro with one or more invocations of the DPT_STORE macro.

Drivers use the DPT_STORE macro with the INIT table marker label to begin a list of DPT_STORE invocations that supply initialization data for the following fields:

UCB\$_FLCK

Index of the fork lock under which the driver performs fork processing. Fork lock indexes are defined by the \$SPLCODDEF definition macro (invoked by DPTAB) as follows:

| IPL | Fork Lock Index |
|-----|-----------------|
| 8 | SPL\$_IOLOCK8 |
| 9 | SPL\$_IOLOCK9 |
| 10 | SPL\$_IOLOCK10 |
| 11 | SPL\$_IOLOCK11 |

UCB\$_DIPL

Device interrupt priority level.

VMS Macros Invoked by Drivers

DPT_STORE

Other commonly initialized fields are as follows:

| | |
|-----------------|------------------------------|
| UCB\$_DEVCHAR | Device characteristics. |
| UCB\$_DEVCLASS | Device class. |
| UCB\$_DEVTYPE | Device type. |
| UCB\$_DEVBUFSIZ | Default buffer size. |
| UCB\$_DEVDEPEND | Device-dependent parameters. |

Drivers use the DPT_STORE macro with the REINIT table marker label to begin a list of DPT_STORE invocations that supply initialization and reinitialization data for the following fields:

| | |
|-------------------------------|---|
| DDB\$_DDT | Driver dispatch table. Every driver must specify a value for this field. |
| CRB\$_INTD+ VEC\$_ISR | Interrupt service routine. |
| CRB\$_INTD2+ VEC\$_ISR | Interrupt service routine for second interrupt vector. |
| CRB\$_INTD+ VEC\$_INITIAL | Controller initialization routine. |
| CRB\$_INTD+ VEC\$_UNITINIT | Unit initialization routine (for UNIBUS, Q22 bus, and generic VAXBI device drivers). Note that MASSBUS drivers must specify the address of the unit initialization routine in an invocation of the DDTAB macro. |

For an example of the use of the DPT_STORE macro, see the description of the DPTAB macro.

DSBINT

Blocks interrupts from occurring on the local processor at or below a specified IPL.

FORMAT **DSBINT** *[ipl=31]* *[,dst=-(SP)]*
 [,environ=MULTIPROCESSOR]

PARAMETERS *[ipl=31]*
 IPL at which to block interrupts. If no **ipl** is specified, the default is IPL 31, which blocks all interrupts.

[dst=-(SP)]
 Location in which to save the current IPL. If no destination is specified, the current IPL is pushed onto the stack.

[environ=MULTIPROCESSOR]
 Processing environment in which the DSBINT synchronization macro is to be assembled. If you do not specify **environ**, or if you do specify **environ=MULTIPROCESSOR**, the DSBINT macro generates the following assembly-time warning message, where *xx* is an IPL above IPL 2:

%MACRO-W-GENWARN, Generated WARNING: Raising IPL to #xx provides no multiprocessing synchronization

If you are certain that the purpose of the macro invocation is to block only local processor events, you can disable the warning message by including **environ=UNIPROCESSOR** in the invocation.

DESCRIPTION The DSBINT macro first stores the current IPL of the local processor and then moves the specified IPL into the processor's IPL register (PR\$_IPL).

Note that the DSBINT and ENBINT macros provide full synchronization only in a uniprocessing environment. In a multiprocessor configuration, DSBINT and ENBINT are suitable only for blocking events on the local processor. To provide synchronized access to system resources and devices in a multiprocessing environment, you *must* use the DEVICELOCK /DEVICEUNLOCK, FORKLCK/FORKUNLOCK, and LOCK/UNLOCK macros.

VMS Macros Invoked by Drivers

ENBINT

ENBINT

Lowers the local processor's IPL to a specified value, thus permitting interrupts to occur at or beneath the current IPL.

FORMAT **ENBINT** *[src=(SP)+]*

PARAMETERS *[src=(SP)+]*

Location containing the IPL to be restored to the processor IPL register (PR\$_IPL) of the local processor. If you do not specify a value in *src*, ENBINT moves the value on the top of the stack into PR\$_IPL.

DESCRIPTION

The ENBINT macro complements the actions of the DSBINT macro, restoring an IPL value to PR\$_IPL. Procedures invoke this macro to lower IPL to a previously saved level. If an interrupt is pending at the current IPL or at any IPL above the IPL specified by *src*, the current procedure is immediately interrupted.

Note that the DSBINT and ENBINT macros only provide full synchronization in a uniprocessor environment. In multiprocessor configurations, DSBINT and ENBINT are only suitable for blocking events on the local processor. To provide synchronized access to system resources and devices in a multiprocessing environment, you *must* use the DEVICELock /DEVICEUNLOCK, FORKLlock /FORKUNLOCK, and LOCK/UNLOCK macros.

\$EQLST

Defines a list of symbols and assigns values to the symbols.

FORMAT **\$EQLST** *prefix* ,*[gb=LOCAL]* ,*init* ,*[incr=1]* ,*list*

PARAMETERS

prefix

Prefix to be used in forming the names of the symbols.

[gb=LOCAL]

Scope of the definition of the symbol, either **LOCAL**, the default, or **GLOBAL**.

init

Value to be assigned to the first symbol in the list.

[incr=1]

Increment by which to increase the value of each succeeding symbol in the list. The default is 1.

list

List of symbols to be defined. Each element in the list can have one of the following forms:

<**symbol**> — where **symbol** is the string appended to the prefix, forming the name of the symbol; the value of the symbol is assigned based on the values of **init** and **incr**.

<**symbol,value**> — where **symbol** is the string that is appended to the prefix, forming the name of the symbol, and **value** specifies the value of the symbol.

DESCRIPTION

See the descriptions of the \$DEFINI and _VIELD macros for additional information on defining symbols for data structure fields.

EXAMPLE

```
$EQLST XA_K_ ,0,1,<- ;Define CSR bit values
      <fnct1,2>-
      <fnct2,4>-
      <fnct3,8>-
      <statusa,2048>-
      <statusb,1024>-
      <statusc,512>-
      >
```

VMS Macros Invoked by Drivers

\$EQLST

This code excerpt produces the following symbols:

| | |
|---------------|------------|
| XA_K_FNCT1 | = 00000002 |
| XA_K_FNCT2 | = 00000004 |
| XA_K_FNCT3 | = 00000008 |
| XA_K_STATUSA | = 00000800 |
| XA_K_STATUSB | = 00000400 |
| XA_K_STAT USC | = 00000200 |

FIND_CPU_DATA

Locates the start of the current process's per-CPU database area (CPU).

FORMAT **FIND_CPU_DATA** *reg* [*,amod=G*] [*,istack=NO*]

PARAMETERS *reg*
Register to receive the base virtual address of the current processor's per-CPU database structure (CPU).

[*amod=G*]
Addressing mode.

[*istack=NO*]
Mechanism by which the base of the per-CPU database structure is calculated. Use ***istack=YES*** used only when it is certain that the processor is executing on the interrupt stack. The mechanism used when ***istack=NO*** is somewhat slower, but works whether the processor is executing on the interrupt stack or kernel stack.

DESCRIPTION The **FIND_CPU_DATA** macro loads the starting virtual address of the current processor's per-CPU database (CPU) into the specified register. A driver generally invokes the **FIND_CPU_DATA** macro in the process of determining the current process of the current CPU when executing in system context.

Such a driver must adhere to the following rules:

- It must invoke the **FIND_CPU_DATA** macro in kernel mode at or above **IPL\$_RESCHED**.
- It must ensure that it will not be rescheduled after issuing the macro while it is using the information returned by **FIND_CPU_DATA**. It typically does this by remaining at **IPL\$_RESCHED** or greater.

EXAMPLE

```
FIND_CPU_DATA R0
MOVL CPU$L_CURPCB(R0),R1
```

The **FIND_CPU_DATA** macro returns the starting virtual address of the current processor's per-CPU database in R0. The subsequent **MOVL** instruction obtains the address of the process currently active on that processor and places it in R1.

VMS Macros Invoked by Drivers

FORK

FORK

Creates a fork process, in which context the code that follows the macro invocation executes.

FORMAT

FORK

DESCRIPTION

The FORK macro calls EXE\$FORK to create a fork process. When the FORK macro is invoked, the following registers must contain the values listed:

| Register | Contents |
|----------|---|
| R3 | Contents to be placed in R3 of the fork process |
| R4 | Contents to be placed in R4 of the fork process |
| R5 | Address of fork block |
| 00(SP) | Address of caller's caller |

Unlike EXE\$IOFORK, EXE\$FORK does not disable device timeouts by clearing the UCB\$V_TIM bit in the field UCB\$L_STS.

VMS Macros Invoked by Drivers

FORKLOCK

- If you specify **fipl=YES**, the FORKLOCK macro takes the following actions:
 - If offset **FKB\$B_FLCK** (**FKB\$B_FIPL**) contains a fork lock index, it sets IPL to the IPL that corresponds to the fork lock index in the spin lock IPL vector (**SMP\$AR_IPLVEC**).
 - If offset **FKB\$B_FLCK** (**FKB\$B_FIPL**) contains a fork IPL, it sets IPL to that fork IPL.

In a *multiprocessing* environment, the FORKLOCK macro stores the fork lock index in R0 and calls SMP\$ACQUIRE. SMP\$ACQUIRE uses the value in R0 to locate the fork lock structure in the system spin lock database (a pointer to which is located at SMP\$AR_SPNLKVEC). Prior to securing the fork lock, SMP\$ACQUIRE raises IPL to its associated IPL (**SPL\$B_IPL**).

In both processing environments, the FORKLOCK macro performs the following tasks:

- Preserves R0 through the macro call (if **preserve=YES** is specified)
- Preserves the current IPL at the specified location (if **savipl** is specified)
- Sets the SMP-modified bit in the driver prologue table (**DPT\$V_SMPMOD** in **DPT\$L_FLAGS**)

EXAMPLE

```
FORKLOCK -
    LOCK=UCB$B_FLCK(R5),-      ;Lock fork database
    SAVIPL=(SP),-            ;Save the current IPL
    PRESERVE=NO              ;Do not preserve R0
INCW  UCB$W_QLEN(R5)         ;Bump device queue length
BSS   #UCB$V_BSY,UCB$W_STS(R5),-
    20$                      ;If set, device is busy
PUSHL R5                    ;Save UCB address
BSBW  IOC$INITIATE          ;Initiate I/O function
POPL  R5                    ;Restore UCB address
FORKUNLOCK -
    LOCK=UCB$B_FLCK(R5),-      ;Unlock fork database
    NEWIPL=(SP)+,-           ;Restore previous IPL
    PRESERVE=NO              ;Do not preserve R0
    RSB

20$:                          ;Place IRP in UCB pending-I/O queue
```

The VMS routine that determines whether a device can immediately service an I/O request synchronizes its access to the fork database by invoking the FORKLOCK macro. The FORKLOCK macro raises IPL to fork IPL and, in a multiprocessing environment, obtains the corresponding fork lock.

Thus synchronized, the VMS routine tests a bit in the UCB to determine whether the device is busy. If the device is not busy, VMS calls a routine that initiates driver processing of the I/O request, still at fork IPL and holding the fork lock. Later, possibly with an invocation of the WFIKPCH macro, the driver start-I/O routine returns control to this routine, which issues the FORKUNLOCK macro to relinquish fork level synchronization.

VMS Macros Invoked by Drivers

FUNCTAB

FUNCTAB

Creates a driver's function decision table (FDT) and generates FDT entries.

FORMAT **FUNCTAB** *[action] ,codes*

PARAMETERS *[action]*

Address of an FDT routine that VMS calls when preprocessing an I/O request whose function code matches a function indicated in the **codes** argument. A plus sign (+) precedes the address of any specified FDT routine that is part of VMS. No plus sign precedes the address of an FDT routine that is contained within the driver module.

You cannot specify an **action** argument in a driver's first two invocations of the FUNCTAB macro.

codes

List of I/O function codes that VMS preprocessing services by calling the FDT routine specified in the **action** argument of the FUNCTAB macro invocation. The macro expansion prefixes each code with the string IO\$_; for example, READVBLK expands to IO\$_READVBLK.

DESCRIPTION

A device driver uses several invocations of the FUNCTAB macro to generate the three components of a function decision table:

- The list of valid I/O function codes
- The list of buffered I/O function codes
- One or more FDT entries

The first two invocations of the FUNCTAB macro in a driver generate the lists of valid I/O functions and buffered I/O functions, respectively. These invocations include the **codes** argument, but not the **action** argument. If no buffered I/O functions are defined for the device, the **codes** argument to the second invocation of the FUNCTAB macro specifies an empty list.

Each succeeding invocation of the FUNCTAB macro generates an FDT entry. Each FDT entry specifies all or a subset of the valid I/O function codes and the address of an FDT routine that performs I/O preprocessing for those function codes. You can specify any valid I/O function code in more than one of these FUNCTAB macro invocations, thus causing more than one FDT routine to be called for a single valid I/O function code.

VMS Macros Invoked by Drivers

FUNCTAB

EXAMPLE

```
XX_FUNCTABLE:
FUNCTAB      ,-                ;Function decision table
              <READLBLK,-      ;Valid functions
              READPBLK,-      ;Read logical block
              READVBLK,-      ;Read physical block
              SENSEMODE,-     ;Read virtual block
              SENSECHAR,-     ;Sense reader mode
              SETMODE,-       ;Sense reader characteristics
              SETCHAR,-       ;Set reader mode
              >                ;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 function FDT routine
              <READLBLK,-      ;Read logical block
              READPBLK,-      ;Read physical block
              READVBLK,-      ;Read virtual block
              >
FUNCTAB      +EXE$SETMODE,-   ;Set mode/characteristics FDT routine
              <SETCHAR,-       ;Set reader characteristics
              SETMODE,-       ;Set reader mode
              >
FUNCTAB      +EXE$SENSEMODE,- ;Sense mode/characteristics FDT routine
              <SENSECHAR,-     ;Sense reader characteristics
              SENSEMODE,-     ;Sense reader mode
              >
```

This function decision table specifies that the routine `XX_READ` be called for all read functions that are valid for the device. `XX_READ` appears later in the driver module. VMS I/O preprocessing will call routines `EXE$SETMODE` and `EXE$SENSEMODE` for the device's set-characteristics and sense-mode functions. Because each of these routines is part of VMS, a plus sign (+) precedes its name in the `FUNCTAB` macro argument.

VMS Macros Invoked by Drivers

IFNORD, IFNOWRT, IFRD, IFWRT

IFNORD, IFNOWRT, IFRD, IFWRT

Determines the read or write accessibility of a range of memory locations.

FORMAT

$\left\{ \begin{array}{l} \text{IFNORD} \\ \text{IFNOWRT} \\ \text{IFRD} \\ \text{IFWRT} \end{array} \right\} \text{ siz , adr , dest [, mode=#0]$

PARAMETERS

siz

Offset of the last byte to check from the first byte to check, a number less than or equal to 512.

adr

Address of first byte to check.

dest

Address to which the macro transfers control, according to the following conditions:

| Macro | Condition |
|---------|---|
| IFNORD | If either of the specified bytes cannot be read in the specified access mode |
| IFNOWRT | If either of the specified bytes cannot be written in the specified access mode |
| IFRD | If both bytes can be read in the specified access mode |
| IFWRT | If both bytes can be written in the specified access mode |

[mode=#0]

Mode in which access is to be checked; zero, the default, causes the check to be performed in the mode contained in the previous-mode field of the current PSL.

DESCRIPTION

The IFNORD and IFRD macros use the PROBER instruction to check the read accessibility of the specified range of memory by checking the accessibility of the first and last bytes in that range. The IFNORD macro passes control to the specified destination if either of the specified bytes cannot be read in the specified access mode. The IFRD macro transfers control if both bytes can be read in the specified access mode. Otherwise, the macros transfer to the next in-line instruction.

VMS Macros Invoked by Drivers

IFNORD, IFNOWRT, IFRD, IFWRT

The IFNOWRT and IFWRT macros use the PROBEW instruction to check the write accessibility of the specified range of memory by checking the accessibility of the first and last bytes in that range. The IFNOWRT macro passes control to the specified destination if either of the specified bytes cannot be written in the specified access mode. The IFWRT macro transfers control to the specified destination if both bytes can be written in the specified access mode. Otherwise, the macros transfer to the next in-line instruction.

EXAMPLE

```
MOVZWL  $SS_ACCVIO,RO          ;Assume read access failure
MOVL    ENTRY_LIST(AP),R11     ;Get address of entry point list
IFRD    #4*4,(R11),50$         ;Branch forward if process
                                           ; has read access
BRW     ERROR                  ;Otherwise stop with error
```

The connect-to-interrupt driver uses the IFRD macro to verify that the process has read access to the four longwords that make up the entry point list. The address of the entry point list was specified in the **p2** argument of the \$QIO request to the driver.

VMS Macros Invoked by Drivers

INVALIDATE_TB

INVALIDATE_TB

Flushes a single page-table entry (PTE) or all PTEs from the translation buffers of all processors in a VAX system.

FORMAT **INVALIDATE_TB** *[addr][,reg][,inst1][,inst2][,inst3]*

PARAMETERS *[addr]*
Virtual address mapped by the PTE for which invalidation is required. If **addr** is blank, then the macro invalidates all PTEs in the translation buffer.

[reg]
Register into which the macro moves the value of **addr**.

[inst1]
First instruction that writes the PTE.

[inst2]
Second instruction that writes the PTE.

[inst3]
Third instruction that writes the PTE.

DESCRIPTION When privileged code alters page mapping information, modifying a valid PTE in an active page table, it must remove the stale value of that PTE from any translation buffers that may have cached it, both on the processor performing the modifications and on any other processor in a VMS multiprocessing system.

You must use the INVALIDATE_TB macro to flush the stale value of a previously valid PTE from the translation buffer of any processor in a VAX system and execute up to three instructions that modify the PTE while all processors are prevented from referencing that page. If you do not supply an **addr** argument, the INVALIDATE_TB macro invalidates all translation buffer entries.

INVALIDATE_TB executes the instructions supplied in the macro invocation as part of a coroutine call to SMP\$INVALID. These instructions, therefore, should not reference the stack and should not use R2. Note that the INVALIDATE_TB macro destroys the contents of R2.

To invoke INVALIDATE_TB, code must be executing at or below IPL\$_INVALIDATE, holding—in a VMS multiprocessing environment—no spin lock ranked higher than INVALIDATE. If you issue the INVALIDATE_TB macro from pageable code, you must ensure that the location of the code has been locked in memory.

VMS Macros Invoked by Drivers

INVALIDATE_TB

EXAMPLE

```
MOVL      8(SP),R2           ;Load virtual address to invalidate
MOVL      12(SP),R3          ;Load address of PTE
INVALIDATE_TB  R2,-          ;Invalidate translation buffer
          INST1=<BICL2 #PTE$M_VALID,(R3)> ;Clear PTE valid bit
```

The INVALIDATE_TB macro causes the PTE corresponding to the virtual address supplied in R2 to be flushed from the system's translation buffers. The macro causes the specified BICL2 instruction to be executed while other processors in the system are prevented from referencing the stale PTE.

VMS Macros Invoked by Drivers

IOFORK

IOFORK

Disables timeouts from a target device and creates a fork process, in which context the code that follows the macro invocation executes.

FORMAT IOFORK

DESCRIPTION The IOFORK macro calls EXE\$IOFORK to disable timeouts from a target device (by clearing UCB\$V_TIM in UCB\$L_STS) and to create a fork process for a device driver.

When the IOFORK macro is invoked, the following registers must contain the values listed:

| Register | Contents |
|----------|---|
| R3 | Contents to be placed in R3 of the fork process |
| R4 | Contents to be placed in R4 of the fork process |
| R5 | Address of a UCB that will be used as a fork block for the fork process to be created |
| 00(SP) | Address of caller's caller |

EXAMPLE

```
WFIKPCH XA_TIME_OUT,IRP$L_MEDIA(R3)        ;Wait for interrupt
IOFORK                                        ;Device has interrupted; fork
```

The start-I/O routine of a driver initiates an I/O request by invoking the WFIKPCH macro. The WFIKPCH macro sets UCB\$V_INT and UCB\$V_TIM in UCB\$L_STS to record an expected interrupt and enable timeouts from the device, saving the PC of the instruction following IOFORK at UCB\$L_FPC in the driver's fork block. When the device interrupts, the driver's interrupt service routine clears UCB\$V_INT and issues the instruction JSB @UCB\$L_FPC(R5), transferring control to the IOFORK macro invocation.

The IOFORK macro clears the UCB\$V_TIM bit, creates a fork block, inserts it in the appropriate fork queue, requests a software interrupt at that fork IPL from the local processor, and returns control to the driver's interrupt service routine at the instruction following the JSB. When the processor's IPL drops below the fork level, the fork dispatcher dequeues the fork block, obtains proper synchronization, and resumes execution at the instruction in the driver that follows the IOFORK invocation.

LOADALT

Lloads a set of Q22-bus alternate map registers.

FORMAT **LOADALT**

DESCRIPTION The LOADALT macro calls IOC\$LOADALTMAP to load a set of Q22 bus alternate map registers (registers 496 to 8191). Map registers must already be allocated before the LOADALT macro can be invoked.

When the LOADALT macro is invoked, register R5 must contain the address of the UCB. LOADALT destroys the contents of R0 through R2.

VMS Macros Invoked by Drivers

LOADMBA

LOADMBA

Loads MASSBUS map registers.

FORMAT

LOADMBA

DESCRIPTION

The LOADMBA macro calls IOC\$LOADMBAMAP to load MASSBUS map registers. The driver must own the MASSBUS adapter, and thus the map registers, before it can invoke LOADMBA.

When the LOADMBA macro is invoked, the following registers must contain the following values:

| Register | Contents |
|----------|---|
| R4 | Address of the MBA's configuration register (MBA\$_CSR) |
| R5 | Address of UCB |

LOADMBA destroys the contents of R0 through R2.

LOADUBA

Loads a set of UNIBUS map registers or a set of the first 496 Q22-bus map registers.

FORMAT **LOADUBA**

DESCRIPTION

The LOADUBA macro calls IOC\$LOADUBAMAP to load a set of UNIBUS map registers or a set of the first 496 Q22-bus map registers. Map registers must already be allocated before the LOADUBA macro can be invoked.

When the LOADUBA macro is invoked, register R5 must contain the address of the UCB. LOADUBA destroys the contents of R0 through R2.

VMS Macros Invoked by Drivers

LOCK

In either processing environment, the LOCK macro performs the following tasks:

- Preserves the current IPL at the specified location (if **savipl** is specified)
- Sets the SMP-modified bit in the driver prologue table (DPT\$V_
SMPMOD in DPT\$L_FLAGS)

VMS Macros Invoked by Drivers

PURDPR

PURDPR

Purges a UNIBUS adapter buffered data path.

FORMAT

PURDPR

DESCRIPTION

The PURDPR macro calls IOC\$PURGDATAP to purge a UNIBUS adapter buffered data path. A driver within an I/O subsystem configuration that does not provide buffered data paths may use the PURDPR macro because the purge operation detects memory parity errors that may have occurred during the transfer. When the PURDPR macro is invoked, R5 must contain the address of the UCB.

When PURDPR returns control to its caller, the following registers contain the following values:

| Register | Contents |
|----------|---|
| R0 | Status of the purge (success or failure) |
| R1 | Contents of data-path register, provided for the use of the driver's register dumping routine |
| R2 | Address of first map register, provided for the use of the driver's register dumping routine |
| R3 | Address of the CRB |

READ_SYSTIME

Reads the current system time.

FORMAT **READ_SYSTIME** *dst*

PARAMETER *dst*
Quadword into which the macro inserts the system time.

DESCRIPTION The READ_SYSTIME macro generates the code required to obtain a consistent copy of the system time from EXE\$GQ_SYSTIME.

Use of the READ_SYSTIME macro is subject to the following restrictions:

- IPL must be less than 23.
- The processor must be executing in kernel mode.
- When using the macro within pageable program sections (or within code executing at IPL 2 and below), you must ensure that the pages involved are locked in memory.

EXAMPLE

```
READ_SYSTIME R0
```

The READ_SYSTIME macro inserts the current system time in R0 and R1.

VMS Macros Invoked by Drivers

RELALT

RELALT

Releases a set of Q22-bus alternate map registers allocated to the driver.

FORMAT

RELALT

DESCRIPTION

The RELALT macro calls IOC\$RELALTMAP to release a set of Q22-bus alternate map registers (registers 496 to 8191) allocated to the driver. When the RELALT macro is invoked, R5 must contain the address of the UCB. RELALT destroys the contents of R0 through R2.

RELCHAN

Releases all controller data channels allocated to a device.

FORMAT **RELCHAN**

DESCRIPTION The RELCHAN macro calls IOC\$RELCHAN to release all controller data channels allocated to a device. When the RELCHAN macro is invoked, R5 must contain the address of the UCB. RELCHAN destroys the contents of R0 through R2.

VMS Macros Invoked by Drivers

RELDPR

RELDPR

Releases a UNIBUS adapter data path register allocated to the driver.

FORMAT

RELDPR

DESCRIPTION

The RELDPR macro calls IOC\$RELDATAP to release a UNIBUS adapter buffered data path allocated to the driver.

When the RELDPR macro is invoked, R5 must contain the address of the UCB. RELDPR destroys the contents of R0 through R2.

RELMPR

Releases a set of UNIBUS map registers or a set of the first 496 Q22-bus map registers allocated to the driver.

FORMAT

RELMPR

DESCRIPTION

The RELMPR macro calls IOC\$RELMAPREG to release a set of map registers allocated to the driver. When the RELMPR macro is invoked, R5 must contain the address of the UCB. RELMPR destroys the contents of R0 through R2.

VMS Macros Invoked by Drivers

RELSCHAN

RELSCHAN

Releases all secondary channels allocated to the driver.

FORMAT

RELSCHAN

DESCRIPTION

The RELSCHAN macro calls IOC\$RELSCHAN to release all secondary data channels (for example, the MASSBUS adapter's controller data channel) allocated to the driver.

When the RELSCHAN macro is invoked, R5 must contain the address of the UCB. RELSCHAN destroys the contents of R0 through R2.

VMS Macros Invoked by Drivers

REQALT

REQALT

Obtains a set of Q22-bus alternate map registers.

FORMAT REQALT

DESCRIPTION

The REQALT macro calls IOC\$REQALTMAP to obtain a set of Q22-bus alternate map registers (registers 496 to 8191). When the REQALT macro is invoked, the following registers must contain the following values:

| Register | Contents |
|----------|----------------------------|
| R5 | Address of UCB |
| 00(SP) | Address of caller's caller |

The REQALT macro destroys the contents of R0 through R2.

VMS Macros Invoked by Drivers

REQCOM

REQCOM

Invokes VMS device-independent I/O postprocessing.

FORMAT

REQCOM

DESCRIPTION

The REQCOM macro calls IOC\$REQCOM to complete the processing of an I/O request after the driver has finished its portion of the processing.

When the REQCOM macro is invoked, the following registers must contain the following values:

| Register | Contents |
|----------|-------------------------------|
| R0 | First longword of I/O status |
| R1 | Second longword of I/O status |
| R5 | Address of UCB |

The REQCOM macro destroys the contents of R0 through R3. All other registers are also destroyed if the action of the macro initiates the processing of a waiting I/O request for the device.

REQDPR

Requests a UNIBUS adapter buffered data path.

FORMAT REQDPR

DESCRIPTION

The REQDPR macro calls IOC\$REQDATAP to request a UNIBUS adapter buffered data path.

When the REQDPR macro is invoked, the following registers must contain the following values:

| Register | Contents |
|----------|----------------------------|
| R5 | Address of UCB |
| 00(SP) | Address of caller's caller |

The REQDPR macro destroys the contents of R0 through R2.

VMS Macros Invoked by Drivers

REQMPR

REQMPR

Obtains a set of UNIBUS map registers or a set of the first 496 Q22 bus map registers.

FORMAT REQMPR

DESCRIPTION The REQMPR macro calls IOC\$REQMAPREG to obtain a set of map registers. When the REQMPR macro is invoked, the following registers must contain the following values:

| Register | Contents |
|----------|----------------------------|
| R5 | Address of UCB |
| 00(SP) | Address of caller's caller |

The REQMPR macro destroys the contents of R0 through R2.

REQPCHAN

Obtains a controller's data channel.

FORMAT **REQPCHAN** *[pri]*

PARAMETERS *[pri]*
Priority of request. If the priority is **HIGH**, REQPCHAN calls IOC\$REQPCHANH; otherwise it calls IOC\$REQPCHANL.

DESCRIPTION The REQPCHAN macro calls IOC\$REQPCHANH or IOC\$REQPCHANL, depending on the priority specified, to obtain a controller's data channel.

When the REQPCHAN macro is invoked, the following registers must contain the following values:

| Register | Contents |
|----------|----------------------------|
| R5 | Address of UCB |
| 00(SP) | Address of caller's caller |

The REQPCHAN macro returns the address of the device's CSR in R4 and destroys the contents of R0 through R2.

VMS Macros Invoked by Drivers

REQSCHAN

REQSCHAN

Obtains a secondary MASSBUS data channel.

FORMAT **REQSCHAN** [*pri*]

PARAMETER *[pri]*
Priority of request. If the priority is **HIGH**, REQSCHAN calls IOC\$REQSCHANH; otherwise it calls IOC\$REQSCHANL.

DESCRIPTION The REQSCHAN macro calls IOC\$REQSCHANH or IOC\$REQSCHANL, depending on the priority specified, to obtain a secondary MASSBUS data channel.

When the REQSCHAN macro is invoked, the following registers must contain the following values:

| Register | Contents |
|----------|----------------------------|
| R5 | Address of UCB |
| 00(SP) | Address of caller's caller |

The REQSCHAN macro returns the address of the device's CSR in R4 and destroys the contents of R0 through R2.

SAVIPL

Saves the current IPL of the local processor.

FORMAT **SAVIPL** *[dst--(SP)]*

PARAMETER *[dst--(SP)]*
Address of longword in which to save the current IPL.

DESCRIPTION The SAVIPL macro stores the current IPL of the local processor, as recorded in the processor IPL register (PR\$_IPL), in the specified location.

VMS Macros Invoked by Drivers

SETIPL

SETIPL

Sets the current IPL of the local processor.

FORMAT **SETIPL** *[ipl=31][environ=MULTIPROCESSOR]*

PARAMETERS ***[ipl=31]***
Level at which to set the current IPL. The default value sets IPL to 31, blocking all interrupts on the local processor.

[environ=MULTIPROCESSOR]
Processing environment in which the SETIPL synchronization macro is to be assembled. If you do not specify **environ**, or if you do specify **environ=MULTIPROCESSOR**, the SETIPL macro generates the following assembly-time warning message, where *xx* is an IPL above IPL 2:

%MACRO-W-GENWARN, Generated WARNING: Raising IPL to #xx provides no multiprocessing synchronization

If you are certain that the purpose of the macro invocation is to block only local processor events, you can disable the warning message by including **environ=UNIPROCESSOR** in the invocation.

DESCRIPTION The SETIPL macro sets the IPL of the local processor by moving the specified **ipl** or IPL 31 into its IPL register (PR\$_IPL).

Note that the SETIPL macro provides full synchronization only in a uniprocessing environment. In a multiprocessor configuration, SETIPL is suitable only for blocking events on the local processor. To provide synchronized access to system resources and devices in a multiprocessing environment, you *must* use the DEVICELock/DEVICEUNLOCK, FORKLock/FORKUNLOCK, and LOCK/UNLOCK macros.

VMS Macros Invoked by Drivers

SETIPL

EXAMPLE

```
DEVICELOCK - ;Secure device lock
  LOCKADDR=UCB$L_DLCK(R5),- ;(also raises IPL to device lock's IPL)
  SAVIPL=- (SP) ;Save current IPL on stack
SETIPL #IPL$POWER,- ;Raise IPL to 31
  ENVIRON=UNIPROCESSOR ;Avoid assembly-time warning
BBC #UCB$V_POWER,-
  UCB$W_STS(R5),30$ ;If clear, no power failure
;Service power failure
.
.
.
DEVICEUNLOCK - ;Release device lock
  LOCKADDR=UCB$L_DLCK(R5),-
  NEWIPL=(SP)+ ;Restore old IPL from stack
.
.
.
;Branch
30$: ;Start device
.
.
.
WFIKPCH ;Wait for interrupt
```

Here, the DEVICELOCK macro achieves synchronized systemwide access to the device registers. The SETIPL macro then synchronizes the local processor against its own powerful interrupt event. The code does not need to synchronize systemwide against powerful events, because its interest is truly limited to the local processor.

Note that the WFIKPCH macro conditionally releases the device lock and restores the old IPL prior to returning control to the caller's caller.

VMS Macros Invoked by Drivers

SOFTINT

SOFTINT

Requests a software interrupt from the local processor at a specified IPL.

FORMAT **SOFTINT** *ipl*

PARAMETER *ipl*
IPL at which the software interrupt is being requested.

DESCRIPTION The SOFTINT macro moves the specified **ipl** into the local processor's Software Interrupt Request Register (PR\$_SIRR), thus requesting a software interrupt at that IPL on the processor.

The processor may take either of the following actions:

- If the local processor is executing at an IPL below the level of the requested interrupt, it immediately transfers control to a software interrupt service routine for the appropriate IPL.
- If the local processor is executing at an IPL equal or above the level of the requested interrupt, it does not transfer control to the software interrupt service routine until its IPL drops below the specified **ipl**.

The SOFTINT macro does not provide the capability of requesting a software interrupt from another processor in a VMS multiprocessing environment.

TIMEWAIT

Waits for a specified bit to be cleared or set within a specified length of time.

FORMAT **TIMEWAIT** *time ,bitval ,source ,context [,sense=.TRUE.]*

PARAMETERS *time*
Number of 10-microsecond intervals to wait. VMS multiplies this value by a processor-specific value in order to calculate the interval to wait. The processor-specific value is inversely proportional to the speed of the processor, but is never less than 1.

bitval
Mask that determines which bits to test.

source
Address of bits to test.

context
Context in which the bits are to be tested (B, W, or L).

[sense=.TRUE.]
If **.TRUE.**, test for one or more of the specified bits set; otherwise test for all bits cleared.

DESCRIPTION The TIMEWAIT macro checks for a specific state by testing bits for a specified length of time.

 If the state comes into existence during the specified interval, the TIMEWAIT macro places a success code in R0 and returns control to its caller. If the state does not occur during the specified period, the TIMEWAIT macro places a failure code in R0 and returns control to its caller. The TIMEWAIT macro destroys the contents of R1, and preserves the contents of all other registers.

 Because the TIMEDWAIT macro provides more flexibility and a more controlled environment for detection of events or conditions, DIGITAL recommends its use over the TIMEWAIT macro.

EXAMPLE

```
MOVQ    R0, -(SP)           ;Save R0,R1
TIMEWAIT #3, #RL_CS_M_CRDY, -
        RL_CS(R4), W
MOVQ    (SP)+, R0          ;Restore R0,R1
```

DLDRIVER's unit initialization routine uses the TIMEWAIT macro to wait 30 microseconds for the RL11 controller to be ready before proceeding.

VMS Macros Invoked by Drivers

TIMEDWAIT

TIMEDWAIT

Waits a specified interval of time for an event or condition to occur.

| | |
|---------------|--|
| FORMAT | TIMEDWAIT <i>time</i> [, <i>ins1</i>] [, <i>ins2</i>] [, <i>ins3</i>] [, <i>ins4</i>] [, <i>ins5</i>] [, <i>ins6</i>] [, <i>donelbl</i>] [, <i>imbedlbl</i>] [, <i>ublbl</i>] |
|---------------|--|

| | |
|-------------------|---|
| PARAMETERS | <i>time</i> Number of 10-microsecond intervals to wait. VMS multiplies this value by a processor-specific value in order to calculate the interval to wait. The processor-specific value is inversely proportional to the speed of the processor, but is never less than 1. |
|-------------------|---|

If you do not specify any embedded instructions, increase the value of **time** by 25 percent.

If you specify embedded instructions that take longer to execute than the average, such as the POLYD instruction, they will cause TIMEDWAIT to wait proportionally longer.

[*ins1*]

First instruction in the loop.

[*ins2*]

Second instruction in the loop.

[*ins3*]

Third instruction in the loop.

[*ins4*]

Fourth instruction in the loop.

[*ins5*]

Fifth instruction in the loop.

[*ins6*]

Sixth instruction in the loop.

[*donelbl*]

Label placed after the instruction at the end of the TIMEDWAIT loop; embedded instructions can pass control to this label in order to pass control to the instruction following the invocation of the TIMEDWAIT macro.

[*imbedlbl*]

Label placed at the first of the embedded instructions; after executing a processor-specific delay, the TIMEDWAIT macro passes control here to retest for the condition.

VMS Macros Invoked by Drivers

TIMEDWAIT

[ublbl]

Label placed at the instruction that performs the processor-specific delay after each execution of the loop of embedded instructions; embedded instructions can pass control here in order to skip the execution of the rest of the embedded instructions in a given execution of the embedded loop.

DESCRIPTION

The TIMEDWAIT macro waits for a period of time for an event or condition to occur. You can specify up to six instructions for this macro to execute in a loop to determine whether the event has occurred.

The TIMEDWAIT macro does not read the processor's clock. The interval it waits is approximate and depends upon the processor and the set of instructions you choose for testing to see if the condition exists.

TIMEDWAIT returns a status code (success or failure) in R0, destroys the contents of R1, and preserves all other registers.

EXAMPLE

```
TIMEDWAIT TIME=#600*1000,-      ;6-second wait loop
      INS1=<TSTB  RL_CS(R4)>,-    ;Is controller busy
      INS2=<BLSS  15$>,-        ;If LSS - yes
      DONELBL=15$              ;Label to exit wait loop
BLBC   R0,25$                  ;Time expired - exit
```

The unit initialization routine of DLDRIVER issues the TIMEDWAIT macro to wait a maximum of six seconds if another unit is busy on the controller's channel.

VMS Macros Invoked by Drivers

UNLOCK

UNLOCK

Relinquishes synchronized access to a system resource as appropriate to the processing environment.

FORMAT **UNLOCK** *lockname* [*,newipl*] [*,condition*] [*,preserve=YES*]

PARAMETERS *lockname*

Name of the system resource to be released or restored.

[*newipl*]

Location containing the IPL to which to lower. A prior invocation of the LOCK macro may have stored this IPL value.

[*condition*]

Indication of a special use of the macro. The only defined **condition** is **RESTORE**, which causes the macro—in a VMS multiprocessing environment—to call SMP\$RESTORE instead of SMP\$RELEASE, thus releasing a single acquisition of the spin lock by the local processor.

[*preserve=YES*]

Indication that the macro should preserve R0 across an invocation. If you do not need to retain the contents of R0, specifying **preserve=NO** can enhance system performance.

DESCRIPTION

In a *uniprocessing* environment, the UNLOCK macro lowers IPL to **newipl**. If an interrupt is pending at the current IPL or at any IPL above **newipl**, the current procedure is immediately interrupted.

In a *multiprocessing* environment, the UNLOCK macro performs the following tasks:

- Preserves R0 through the macro call (if **preserve=YES** is specified).
- Generates a spin lock index of the format **SPL\$_lockname** and stores it in R0.
- Calls SMP\$RELEASE or, if **condition=RESTORE** is specified, SMP\$RESTORE. These routines index into the system spin lock database (a pointer to which is located at SMP\$AR_SPNLKVEC) to release the appropriate spin lock.
- Moves any specified **newipl** into the local processor's IPL register (PR\$_IPL). If an interrupt is pending at the current IPL or at any IPL above **newipl**, the current procedure is immediately interrupted.

In either processing environment, the UNLOCK macro sets the SMP-modified bit in the driver prologue table (DPT\$_SMPMOD in DPT\$_L_FLAGS).

\$VEC

Defines an entry in a port driver vector table within the context of a \$VECINI macro.

FORMAT **\$VEC** *entry, routine*

PARAMETERS *entry*
Name of the vector table entry, specified without the PORT_ prefix.

routine
Name of the service routine within the driver that corresponds to the entry point.

DESCRIPTION A terminal port driver uses the \$VEC macro to validate and generate a vector table entry. A driver need not invoke the \$VEC macro to associate a routine with each entry in the vector table. The \$VECINI macro initializes all unspecified entry points with the address of the driver's null entry point.

To use the \$VEC macro, the driver must include an invocation of the \$TTYMACS definition macro (from SYS\$LIBRARY:LIB.MLB). See the description of the \$VECINI macro for an example of creating a port driver vector table.

VMS Macros Invoked by Drivers

\$VECEND

\$VECEND

Ends the scope of the \$VECINI macro, thereby completing the definition of a port driver vector table.

FORMAT **\$VECEND** [*end*]

PARAMETER [*end*]
Flag controlling the generation of the end of the vector table. This argument is generally omitted so that the \$VECEND macro can generate the end of the vector table. Otherwise, the \$VECEND macro does not generate the end of the table.

DESCRIPTION A terminal port driver uses the \$VECEND macro to generate the longword of zeros that terminates a port driver vector table initialized by the \$VECINI and \$VEC macros. It also positions the location counter at label **drivername\$VECEND**, as defined by the \$VECINI macro.

To use the \$VECEND macro, the driver must include an invocation of the \$TTYMACS definition macro (from SYS\$LIBRARY:LIB.MLB). See the descriptions of the \$VECINI and \$VEC macros for additional information on creating a port driver vector table.

\$VECINI

Begins the definition of a port vector table.

FORMAT **\$VECINI** *drivername, null_routine [,prefix=PORT_]*

PARAMETERS *drivername*

Prefix (usually two letters) of the driver name (for example, DZ).

null_routine

Address of the driver's null entry point, usually specified in the format **drivername\$NULL**. This address contains an RSB instruction.

[,prefix=PORT_]

Prefix to be added to the symbols defined in subsequent invocations of the \$VEC macro.

DESCRIPTION

A terminal port driver uses the \$VECINI macro to begin the definition of a port vector table and initialize each table entry to point to the driver's null entry point. The \$VECINI macro generates the label **drivername\$VEC** at the beginning of the table and **drivername\$VECEND** at the end of the table.

The \$VEC macro defines valid entries within the port driver vector table specified by the invocation of the \$VECINI macro, and the \$VECEND macro ends the table's definition.

To use the \$VECINI macro, the driver must include an invocation of the \$TTYMACS definition macro (from SYS\$LIBRARY:LIB.MLB).

EXAMPLE

```
$VECINI DZ32,DZ$NULL
$VEC  STARTIO,DZ32$STARTIO      ;Start new output
$VEC  SET_LINE,DZ32$SET_LINE    ;Set new parity/speed
$VEC  XON,DZ32$XON              ;Send XON
$VEC  XOFF,DZ32$XOFF            ;Send XOFF
$VEC  STOP,DZ32$STOP           ;Stop current output
$VEC  ABORT,DZ32$ABORT          ;Abort current output
$VEC  RESUME,DZ32$RESUME        ;Resume stopped output
$VEC  MAINT,DZ32$MAINT          ;Invoke maintenance functions
$VECEND
```

In this example, the \$VECINI macro creates a port driver vector table. The table entries defined by the eight subsequent invocations of the \$VEC macro (PORT_STARTIO, PORT_SET_LINE, and so on) are set up to point to the specified routines in the port driver. The \$VECINI macro initializes any entry point not defined by a \$VEC macro (for instance, PORT_SET_MODEM) with the address of the null entry point, DZ\$NULL.

The \$VECEND macro concludes the definition of the port driver vector table.

VMS Macros Invoked by Drivers

\$VIELD, _VIELD

\$VIELD, _VIELD

Defines symbolic offsets and masks for bit fields.

FORMAT { **\$VIELD** } *mod , inibit , fields*
 { **_VIELD** }

PARAMETERS *mod*
Module in which this bit field is defined; the prefix portion of the name of the symbol to be defined.

inibit
Bit within the field on which the positions of the bits to be defined are based.

fields
One or more fields of the form *<sym,[size=1],[mask]>* , where these arguments are defined as follows:

| Argument | Meaning |
|----------|--|
| sym | String appended to the string "mod\$" to form the name of this bit field. |
| [size=1] | Size in bits of this bit field. If you specify a value greater than 1, the VIELD macro generates a symbol for the size of the bit field. |
| [mask] | Character "M" if the VIELD macro is to generate a symbol for the mask of the bit field, blank otherwise. |

DESCRIPTION The \$VIELD and _VIELD macros define bit fields whose names have the form *mod\$x_sym* and *mod_x_sym* (where *x* can be V, S, or M and *sym* is a value supplied in the **fields** argument). Because the dollar-sign character (\$) is reserved for use in VMS-defined symbols, use of the _VIELD macro is recommended for non-DIGITAL-supplied device drivers.

See the descriptions of the \$DEFINI and \$EQLST macros for additional information on defining symbols for data structure fields.

VMS Macros Invoked by Drivers

\$VIELD, _VIELD

EXAMPLE

```
$EQLST  XA_K_ , , 0, 1, <-           ;Define CSR bit values
        <fnc1, 2>-
        <fnc2, 4>-
        <fnc3, 8>-
_VIELD  XX_CSR, 0, <-           ;Control/status register
        <GO, , M>,-             ;Start device
        <FNCT, 3, M>,-         ;Function bits
        <XBA, 2, M>,-         ;Extended address bits
        <IE, , M>,-           ;Enable interrupts
        <MAINT>,-             ;Maintenance bit
        <ATTN>,-             ;Status from other processors
        >
```

This code excerpt produces the following symbols:

```
XX_CSR_M_FNCT           = 0000000E
XX_CSR_M_GO             = 00000001
XX_CSR_M_IE             = 00000040
XX_CSR_M_XBA            = 00000030
XX_CSR_S_FNCT           = 00000003
XX_CSR_S_XBA            = 00000002
XX_CSR_V_FNCT           = 00000001
XX_CSR_V_GO             = 00000000
XX_CSR_V_IE             = 00000006
XX_CSR_V_MAINT          = 00000007
XX_CSR_V_XBA            = 00000004
```

VMS Macros Invoked by Drivers

WFIKPCH, WFIRLCH

WFIKPCH, WFIRLCH

Suspends a driver fork thread and folds its context into a fork block in anticipation of a device interrupt or timeout. When WFIKPCH is invoked, the fork thread keeps ownership of the controller channel while waiting; when WFIRLCH is invoked, the fork thread releases ownership of the controller channel.

FORMAT { **WFIKPCH** } *excpt* [, *time=65536*]
 { **WFIRLCH** }

PARAMETERS *excpt*
Name of a device timeout handling routine; the address of this routine must be within 65,536 bytes of the address at which the WFIKPCH macro is invoked.

[*time=65536*]
Timeout interval, expressed as the number of seconds to wait for an interrupt before a device timeout is considered to exist. A value equal to or greater than 2 is required because the timeout detection mechanism is accurate only to within one second.

DESCRIPTION The WFIKPCH and WFIRLCH macros push **time** on the stack and call IOC\$WFIKPCH and IOC\$WFIRLCH, respectively. After the JSB instruction that makes the routine call, either of these macros constructs a word that contains the relative offset to the timeout handling routine specified in **excpt**. Because these routines compute and store the address of the following instruction in the fork block at UCB\$L_FPC, the software timer interrupt service routine can determine the routine's location and call it if the device times out before it can deliver an interrupt.

IOC\$WFIKPCH and IOC\$WFIRLCH assume that, prior to the invocation of the macro, a DEVICELock macro has been issued—both to synchronize with other device activity and to leave the IPL of the previous code thread on the top of the stack. Upon storing the context of and suspending the current code thread, IOC\$WFIKPCH and IOC\$WFIRLCH return control to their caller's caller at the stored IPL.

When the WFIKPCH or WFIRLCH macro is invoked, the following locations must contain the values listed:

| Location | Contents |
|----------|---|
| R5 | Address of UCB |
| 00(SP) | IPL at which control is passed to the caller's caller |
| 04(SP) | Address (in the caller's caller) at which to return control |

VMS Macros Invoked by Drivers

WFIKPCH, WFIRLCH

The suspended code thread is resumed by the occurrence of an interrupt signaling the successful completion of a device operation. When an interrupt occurs, control returns to the instruction following the macro. If a device timeout occurs before an interrupt can be posted, the timeout handling routine specified in **except** is called. In both instances, subsequent code can assume that only R3 and R4 have been preserved across the suspension.

See the descriptions of the DEVICELock, IOFORK, and SETIPL macros for examples of the use of the WFIKPCH macro.

C

Operating System Routines

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

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

Operating System Routines

COM\$DELATTNAST

COM\$DELATTNAST

Delivers all attention ASTs linked in the specified list.

module

COMDRVSUB

input

| Location | Contents |
|----------|---|
| R4 | Address of listhead of AST control blocks |
| R5 | Address of UCB |

output

| Location | Contents |
|--------------------|-----------|
| Specified listhead | Empty |
| R0 through R11 | Preserved |

synchronization

COM\$DELATTNAST executes and exits at the caller's IPL, and acquires no spin locks.

DESCRIPTION

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

COM\$DRVDEALMEM

Deallocates system dynamic memory.

module

COMDRVSUB

input

| Location | Contents |
|-------------|---|
| R0 | Address of block to be deallocated |
| IRP\$W_SIZE | Size of block in bytes (must be at least 24 bytes long) |

output

| Location | Contents |
|----------------|-----------|
| R0 through R11 | Preserved |

synchronization

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

DESCRIPTION

COM\$DRVDEALMEM calls EXE\$DEANONPAGED to deallocate the buffer specified by R0. If the driver is running at an IPL above IPL\$_SYNCH, COM\$DRVDEALMEM transforms the block to be deallocated into a fork block, and requests a software interrupt at IPL\$_QUEUEAST. The code that executes in the fork process jumps to EXE\$DEANONPAGED.

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

Operating System Routines

COM\$FLUSHATTNS

COM\$FLUSHATTNS

Flushes an attention AST list.

module

COMDRVSUB

input

| Location | Contents |
|--------------|---|
| R4 | Address of PCB |
| R5 | Address of UCB |
| R6 | Number of the assigned I/O channel |
| R7 | Address of listhead of AST control blocks |
| UCB\$_DLCK | Address of device lock |
| PCB\$_PID | Process ID |
| PCB\$_ASTCNT | ASTs remaining in quota |

output

| Location | Contents |
|--------------------|--|
| R0 | SS\$_NORMAL |
| R1, R2, R7 | Destroyed |
| PCB\$_ASTCNT | Incremented by the number of AST control blocks that are flushed |
| Specified listhead | Updated |

synchronization

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

DESCRIPTION

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

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

COM\$POST

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

module

COMDRVSUB

input

| Location | Contents |
|---------------|---|
| R3 | Address of IRP |
| R5 | Address of UCB |
| IRP\$_MEDIA | Data to be copied to the I/O status block |
| IRP\$_MEDIA+4 | Data to be copied to the I/O status block |

output

| Location | Contents |
|-------------|-------------|
| R0 and R1 | Destroyed |
| UCB\$_OPCNT | Incremented |

synchronization

Drivers call COM\$POST at or above fork IPL. COM\$POST executes at its caller's IPL and returns control at that IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

A driver fork process calls COM\$POST after it has completed device-dependent I/O processing for an I/O request initiated by EXE\$ALTQUEPKT.

COM\$POST inserts the IRP into the local processor's I/O postprocessing queue headed by CPU\$_PSBL, requests an IPL\$_IOPOST software interrupt, and returns control to its caller. Unlike IOC\$IOPOST, it does not attempt to dequeue any IRP waiting for the device or change the busy status of the device.

Operating System Routines

COM\$SETATTNAST

COM\$SETATTNAST

Enables or disables attention ASTs.

module

COMDRVSUB

input

| Location | Contents |
|---------------|---|
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| R7 | Address of listhead of AST control blocks |
| AP | Address of \$QIO system service argument list |
| IRP\$W_CHAN | I/O request channel index number |
| UCB\$L_DLCK | Address of device lock |
| PCB\$W_ASTCNT | Number of ASTs remaining in process quota |
| PCB\$L_PID | Process ID |
| 00(AP) | Address of process's AST routine |
| 04(AP) | AST parameter |
| 08(AP) | Access mode for AST |

output

| Location | Contents |
|--------------------|---|
| R0 | SS\$_NORMAL, SS\$_EXQUOTA, or SS\$_INSMEM |
| R1 and R2 | Destroyed |
| R3 | Address of IRP |
| R5 | Address of UCB |
| R6, R7, R8 | Destroyed |
| PCB\$W_ASTCNT | Decrementated |
| Specified listhead | Updated |

synchronization

COM\$SETATTNAST raises IPL to device IPL, acquiring the corresponding device lock. It returns control to its caller at the caller's IPL.

DESCRIPTION

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

If the **p1** argument of the request contains a zero, COM\$SETATTNAST transfers control to COM\$FLUSHATTNS, which disables all ASTs indicated by the PID and I/O channel number (IRP\$W_CHAN). COM\$FLUSHATTNS searches through the AST control block (ACB) list, extracts each identified ACB, deallocates, and returns to the caller of COM\$SETATTNAST.

Operating System Routines

COM\$SETATTNAST

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

- Spin lock index SPL\$C_QUEUEAST
- Address of the AST routine (as specified in **p1**)
- AST parameter (as specified in **p2**)
- Access mode (as specified in **p3** and maximized against the current process's access mode and bit ACB\$V_QUOTA set to indicate a process-requested AST)
- Number of the assigned I/O channel
- PID of the requesting process

COM\$SETATTNAST links the ACB to the start of the specified linked list of ACBs located in a UCB extension area. (See Section A.14 for information on defining an extension to a UCB.) COM\$DELATTNAST can later use the expanded ACB to fork to IPL\$_QUEUEAST, at which IPL it reformats the block into a standard ACB.

If the process exceeds buffered I/O or AST quotas, or if there is no memory available to allocate the expanded ACB, COM\$SETATTNAST restores PCB\$W_ASTCNT to its original value and transfers control to EXE\$ABORTIO with error status.

Operating System Routines

ERL\$DEVICERR, ERL\$DEVICTMO, ERL\$DEVICEATTN

ERL\$DEVICERR, ERL\$DEVICTMO, ERL\$DEVICEATTN

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

module

ERRORLOG

input

| Location | Contents |
|-----------------|---|
| R5 | Address of UCB |
| DDT\$W_ERRORBUF | Size of error message buffer in bytes |
| UCB\$L_DEVCHAR | Bit DEV\$V_ELG set |
| UCB\$W_FUNC | Bit IO\$V_INHERLOG clear |
| UCB\$L_IRP | Address of IRP currently being processed (ERL\$DEVICERR and ERL\$DEVICTMO only) |
| UCB\$L_ORB | ORB address |

output

| Location | Contents |
|----------------|---------------------------------|
| UCB\$W_ERRCNT | Incremented |
| UCB\$L_EMB | Address of error message buffer |
| UCB\$L_STS | UCB\$V_ERLOGIP set |
| R0 through R11 | Preserved |

synchronization

A driver calls ERL\$DEVICERR, ERL\$DEVICTMO, or ERL\$DEVICEATTN, at or above fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment. These routines return control to the caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

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

- Increments UCB\$W_ERRCNT to record a device error. If the error-log-in-progress bit (UCB\$V_ERLOGIP in UCB\$L_STS) is set, the routine returns control to its caller.
- Allocates from the current error log allocation buffer an error message buffer of the length specified in the device's DDT (in argument **erlgbf** to the DDTAB macro). This allocation is performed at IPL\$_EMB holding the EMB spin lock.

Operating System Routines

ERL\$DEVICERR, ERL\$DEVICTMO, ERL\$DEVICEATTN

- Initializes the buffer with the current system time, error log sequence number, and error type code. These routines use the following error type codes:

| | |
|-----------------|-----------------------------|
| ERL\$DEVICERR | Device error (EMB\$_DE) |
| ERL\$DEVICTMO | Device timeout (EMB\$_DT) |
| ERL\$DEVICEATTN | Device attention (EMB\$_DA) |

- Places the address of the error message buffer in UCB\$_EMB.
- Sets UCB\$_ERLOGIP in UCB\$_STS.
- Loads fields from the UCB, the IRP, and the DDB into the buffer, including the following:

| | |
|----------------|---|
| UCB\$_DEVCLASS | Device class |
| UCB\$_DEVTYPE | Device type |
| IRP\$_PID | Process ID of the process originating the I/O request (ERL\$_DEVICERR and ERL\$_DEVICTMO) |
| IRP\$_BOFF | Transfer parameter (ERL\$DEVICERR and ERL\$DEVICTMO) |
| IRP\$_BCNT | Transfer parameter (ERL\$DEVICERR and ERL\$DEVICTMO) |
| UCB\$_MEDIA | Disk size |
| UCB\$_UNIT | Unit number |
| UCB\$_ERRCNT | Count of device errors |
| UCB\$_OPCNT | Count of completed operations |
| ORB\$_OWNER | UIC of volume owner |
| UCB\$_DEVCHAR | Device characteristics |
| UCB\$_SLAVE | Slave unit number |
| IRP\$_FUNC | I/O function value (ERL\$DEVICERR and ERL\$DEVICTMO) |
| DDB\$_NAME | Device name (concatenated with cluster node name if appropriate) |

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

Note that a driver must define the local disk UCB extension or local tape UCB extension, as described in Section A.14, to use these error logging routines.

Operating System Routines

EXE\$ABORTIO

EXE\$ABORTIO

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

module

SYSQIOREQ

input

| Location | Contents |
|---------------|--|
| R0 | First longword of status for the I/O status block |
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| IRP\$L_IOSB | Address of I/O status block |
| IRP\$B_RMOD | ACB\$V_QUOTA set indicates process-specified AST pending |
| PCB\$W_ASTCNT | Count of available AST queue entries |

output

| Location | Contents |
|---------------|-------------------------------------|
| IRP\$L_IOSB | Zero |
| IRP\$B_RMOD | ACB\$V_QUOTA clear |
| PCB\$W_ASTCNT | Incremented if ACP\$V_QUOTA was set |

synchronization

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

EXE\$ABORTIO exits at normal process IPL (IPL 0).

DESCRIPTION

EXE\$ABORTIO performs the following actions:

- 1 Clears IRP\$L_IOSB so that no status is returned by I/O postprocessing
- 2 Clears ACP\$V_QUOTA in IRP\$B_RMOD to prevent the delivery of any AST to the process specified in the I/O request
- 3 Updates the count of available AST entries at PCB\$W_ASTCNT, if necessary
- 4 Inserts the IRP in the local processor's I/O postprocessing queue headed by CPU\$L_PSBL
- 5 If the queue is empty, requests a software interrupt from the local processor at IPL\$_IOPOST

This interrupt causes I/O postprocessing to occur before the remaining instructions in EXE\$ABORTIO are executed.

Operating System Routines

EXE\$ABORTIO

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

- Lowers IPL to zero
- Issues the RET instruction that restores the original access mode of the caller of the \$QIO system service and returns control to the system service dispatcher

EXE\$ABORTIO returns in R0 the final status code saved when the exit routine was called. Any ASTs specified when the I/O request was issued will not be delivered, and any event flags requested will not be set.

Operating System Routines

EXE\$ALLOCBUF, EXE\$ALLOCIRP

EXE\$ALLOCBUF, EXE\$ALLOCIRP

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

module

MEMORYALC

input

| Location | Contents |
|------------|--|
| R1 | Size of requested buffer in bytes (EXE\$ALLOCBUF only). This value should include the 12 bytes required to store header information. |
| PCB\$L_STS | PCB\$V_SSRWAIT clear if the process should wait if no memory is available for requested buffer; set if resource wait mode is disabled. |

output

| Location | Contents |
|----------------------------------|--|
| R0 | SS\$_NORMAL or SS\$_INSFMEM. |
| R1 | Size of requested buffer in bytes (IRP\$_LENGTH for EXE\$ALLOCIRP). |
| R2 | Address of allocated buffer. |
| R4 | See the following discussion. |
| IRP\$_SIZE (in allocated buffer) | Size of requested buffer in bytes (for EXE\$ALLOCBUF), IRP\$_LENGTH (for EXE\$ALLOCIRP). |
| IRP\$_TYPE (in allocated buffer) | DYN\$_BUFIO (for EXE\$ALLOCBUF), DYN\$_IRP (for EXE\$ALLOCIRP). |

synchronization

EXE\$ALLOCBUF and EXE\$ALLOCIRP set IPL to IPL\$_ASTDEL. As a result they cannot be called by code executing above IPL\$_ASTDEL. They return control to their callers at the caller's IPL.

DESCRIPTION

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

If sufficient memory is not available, EXE\$ALLOCBUF and EXE\$ALLOCIRP move the current PCB (CTL\$GL_PCB) into R4 to determine whether the process has resource wait mode enabled. If PCB\$V_SSRWAIT in PCB\$L_STS is clear, these routines place the process in a resource wait state until memory is released.

The caller must check and adjust process quotas (JIB\$_BYTCNT and/or JIB\$_BYTLM) by calling EXE\$DEBIT_BYTCNT or EXE\$DEBIT_BYTCNT_BYTLM. (Note that you can perform this task *and* allocate a buffer of the requested size by using the routines EXE\$DEBIT_BYTCNT_ALO and EXE\$DEBIT_BYTCNT_BYTLM_ALO. These routines invoke EXE\$ALLOCBUF.)

Operating System Routines

EXE\$ALLOCBUF, EXE\$ALLOCIRP

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

Note that the value returned in R1 and placed at IRP\$W_SIZE in the allocated buffer is the size of the requested buffer. The actual size of the allocated buffer is determined according to the algorithms used by EXE\$ALONONPAGED and the size of the lookaside list packets. The nonpaged pool deallocation routine (EXE\$DEANONPAGED), called in buffered I/O postprocessing, uses similar algorithms when returning memory to nonpaged pool.

Operating System Routines

EXE\$ALONONPAGED

EXE\$ALONONPAGED

Allocates a block of memory from nonpaged pool.

module

MEMORYALC

input

| Location | Contents |
|----------|----------------------------------|
| R1 | Size of requested block in bytes |

output

| Location | Contents |
|----------|--|
| R0 | SS\$_NORMAL or SS\$_INSFMEM. |
| R1 | If the allocation succeeds from one of the lookaside lists, the value returned in R1 remains the size of the requested block. If the allocated block is from general nonpaged pool, the value in R1 is the requested size, rounded up to a 16-byte multiple. |
| R2 | Address of allocated block. |

synchronization

EXE\$ALONONPAGED executes at its caller's IPL and at IPL\$_POOL, obtaining the POOL spin lock in a VMS multiprocessing environment. For this reason, it cannot be called by code executing above IPL\$_POOL.

EXE\$ALONONPAGED returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

Depending upon the size of the requested block, EXE\$ALONONPAGED allocates nonpaged pool either from one of the lookaside lists (SRP, IRP, or LRP) or from the variable region of nonpaged dynamic memory.

EXE\$ALONONPAGED does not initialize the header of the allocated block of memory.

EXE\$ALONPAGVAR

Allocates a block of memory from the variable region of nonpaged pool.

module

MEMORYALC

input

| Location | Contents |
|----------|----------------------------------|
| R1 | Size of requested block in bytes |

output

| Location | Contents |
|----------|--|
| R0 | SS\$_NORMAL or SS\$_INSFMEM |
| R1 | Size of requested buffer, rounded up to a 16-byte multiple |
| R2 | Address of allocated block |

synchronization

EXE\$ALONPAGVAR executes at its caller's IPL and at IPL\$_POOL, holding the POOL spin lock in a VMS multiprocessing environment. For this reason, its caller cannot be executing at an IPL above IPL\$_POOL.

EXE\$ALONPAGVAR returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

EXE\$ALONPAGVAR allocates a block of memory of the requested size from the variable region of nonpaged dynamic memory. Because EXE\$ALONPAGVAR does not attempt to service the request from the lookaside lists, it is suitable for driver fork processes that may afterwards return the allocated block to nonpaged pool in pieces.

EXE\$ALONPAGVAR does not initialize the header of the allocated block of memory.

Operating System Routines

EXE\$ALOPHYCNTG

EXE\$ALOPHYCNTG

Allocates a physically-contiguous block of memory.

module

MEMORYALC

input

| Location | Contents |
|----------|---|
| R1 | Number of physically contiguous pages to allocate |

output

| Location | Contents |
|----------|---|
| R0 | SS\$_NORMAL, SS\$_INSFMEM, or SS\$_INSFSPTS |
| R2 | System virtual address of allocated block, if the allocation succeeds |

synchronization

EXE\$ALOPHYCNTG raises IPL to IPL\$_SYNCH and obtains the MMG spin lock. As a result, its caller cannot be executing above IPL\$_SYNCH or hold any spin lock ranked higher than MMG. (For instance, a driver fork process executing at IPL\$_SYNCH holding the IOLOCK8 fork lock can call EXE\$ALOPHYCNTG.)

EXE\$ALOPHYCNTG returns control to its caller at the caller's IPL. The caller retains any spin lock it held at the time of the call.

DESCRIPTION

EXE\$ALOPHYCNTG allocates a physically contiguous block of memory. You cannot deallocate memory allocated by EXE\$ALOPHYCNTG.

Note that the number of SPT slots available depends on the value of the SPTREQ system parameter.

EXE\$ALTQUEPKT

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

module

SYSQIOREQ

input

| Location | Contents |
|----------------|--|
| R3 | Address of IRP |
| R5 | Address of UCB |
| DDT\$_ALTSTART | Address of alternate start-I/O routine |
| UCB\$_FLCK | Fork lock index |
| UCB\$_DDB | Address of unit's DDB |
| DDB\$_DDT | Address of DDT |

output

| Location | Contents |
|---------------|-----------|
| R0 through R5 | Destroyed |

synchronization

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

DESCRIPTION

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

Operating System Routines

EXE\$CREDIT_BYTCNT, EXE\$CREDIT_BYTCNT_BYTLM

EXE\$CREDIT_BYTCNT, EXE\$CREDIT_BYTCNT_BYTLM

Return credit to a job's buffered-I/O byte count quota and byte limit.

module

EXSUBROUT

input

| Location | Contents |
|---------------|---|
| R0 | Number of bytes to return to the byte count quota (and byte limit) |
| R4 | Address of current PCB |
| JIB\$B_FLAGS | JIB\$V_BYTCNT_WAITERS set if there are processes waiting for byte count quota from this JIB |
| JIB\$L_BYTCNT | Job's byte count usage quota |
| JIB\$L_BYTLM | Job's byte limit (used by EXE\$CREDIT_BYTCNT_BYTLM) |

output

| Location | Contents |
|---------------|---------------------------------------|
| R0 | Destroyed |
| JIB\$L_BYTCNT | Updated |
| JIB\$L_BYTLM | Updated (by EXE\$CREDIT_BYTCNT_BYTLM) |

synchronization

EXE\$CREDIT_BYTCNT and EXE\$CREDIT_BYTCNT_BYTLM raise IPL to IPL\$_SYNCH and obtain the JIB spin lock and the SCHED spin lock (if JIB\$V_BYTCNT_WAITERS is set) in a VMS multiprocessing environment. As a result, their callers cannot be executing above IPL\$_SYNCH or hold any spin lock ranked higher than JIB. (For instance, a driver fork process executing at IPL\$_SYNCH holding the IOLOCK8 fork lock can call these routines. It cannot, however, hold the SCHED spin lock.)

EXE\$CREDIT_BYTCNT and EXE\$CREDIT_BYTCNT_BYTLM return control to their callers at the caller's IPL. Their caller retains any spin locks it held at the time of the call.

DESCRIPTION

EXE\$CREDIT_BYTCNT provides a synchronized method of crediting a job's byte count quota to JIB\$L_BYTCNT. EXE\$CREDIT_BYTCNT_BYTLM also credits a job's byte limit to JIB\$L_BYTLM.

Both routines round the value specified in R0 up to the nearest 16-byte boundary before applying it to the JIB. Both check JIB\$V_BYTCNT_WAITERS to determine if any process is waiting for the return of nonpaged pool quota for this JIB. If a process is waiting, EXE\$CREDIT_BYTCNT calls a system routine that attempts to fill any pending requests.

EXE\$DEANONPAGED

Deallocates a block of memory and returns it to nonpaged pool.

module

MEMORYALC

input

| Location | Contents |
|-------------|------------------------------------|
| R0 | Address of block to be deallocated |
| IRP\$W_SIZE | Size of block in bytes |

output

| Location | Contents |
|-----------|-----------|
| R1 and R2 | Destroyed |

synchronization

EXE\$DEANONPAGED executes at the caller's IPL, at IPL\$_SYNCH holding the SCHED spin lock, and at IPL\$_POOL holding the POOL spin lock. As a result, its caller cannot be executing above IPL\$_SYNCH. EXE\$DEANONPAGED returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

EXE\$DEANONPAGED deallocates the specified block of memory to nonpaged dynamic memory, returning it to a lookaside list or the variable region of nonpaged pool as appropriate. It also reports to the scheduler the availability of the deallocated pool.

EXE\$DEANONPAGED issues a BADDALRQSZ bugcheck if the address of the pool to be deallocated is not aligned on a 16-byte boundary.

Operating System Routines

EXE\$DEBIT_BYTCNT(_NW), EXE\$DEBIT_BYTCNT_BYTLM(_NW)

EXE\$DEBIT_BYTCNT(_NW), EXE\$DEBIT_BYTCNT_BYTLM(_NW)

Determine whether a job's buffered I/O byte count quota usage permits the process to be granted additional buffered I/O and, if so, adjust the job's byte count quota and byte limit.

module

EXSUBROUT

input

| Location | Contents |
|----------------|--|
| R1 | Number of bytes to be deducted; bit 31, when set, disables the routine's check against IOC\$GW_MAXBUF |
| R4 | Address of current PCB |
| PCB\$L_STS | PCB\$V_SSRWAIT clear if the process should wait for buffered-I/O byte quota; set if resource wait mode is disabled |
| IOC\$GW_MAXBUF | Maximum number of buffered I/O bytes the system allows to a single request |
| JIB\$L_BYTCNT | Job's byte count usage quota |
| JIB\$L_BYTLM | Job's byte limit (used by EXE\$DEBIT_BYTCNT_BYTLM and EXE\$DEBIT_BYTCNT_BYTLM_NW) |

output

| Location | Contents |
|---------------|---|
| R0 | SS\$_NORMAL or SS\$_EXQUOTA |
| R1 | Number of bytes deducted; bit 31 cleared |
| JIB\$L_BYTCNT | Updated if successful |
| JIB\$L_BYTLM | Updated if successful (by EXE\$DEBIT_BYTCNT_BYTLM and EXE\$DEBIT_BYTCNT_BYTLM_NW) |

synchronization

EXE\$DEBIT_BYTCNT, EXE\$DEBIT_BYTCNT_NW, EXE\$DEBIT_BYTCNT_BYTLM, and EXE\$DEBIT_BYTCNT_BYTLM_NW raise IPL to IPL\$_SYNCH and obtain the JIB spin lock in a VMS multiprocessing environment. As a result, their callers cannot be executing above IPL\$_SYNCH or hold any spin lock ranked higher than JIB. (For instance, a driver fork process executing at IPL\$_SYNCH holding the IOLOCK8 fork lock can call these routines. It cannot, however, hold the SCHED spin lock.)

EXE\$DEBIT_BYTCNT, EXE\$DEBIT_BYTCNT_NW, EXE\$DEBIT_BYTCNT_BYTLM, and EXE\$DEBIT_BYTCNT_BYTLM_NW return control to their callers at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Operating System Routines

EXE\$DEBIT_BYTCNT(_NW), EXE\$DEBIT_BYTCNT_BYTLM(_NW)

DESCRIPTION

EXE\$DEBIT_BYTCNT and EXE\$DEBIT_BYTCNT_NW check whether a process has sufficient quota for a buffer of the specified size and, if so, deduct the corresponding number of bytes from the job's byte count quota. EXE\$DEBIT_BYTCNT_BYTLM and EXE\$DEBIT_BYTCNT_BYTLM_NW also adjust the job's byte limit. All routines round the value specified in R1 up to the nearest 16-byte boundary before applying it to the JIB.

If the process's quota usage is too large, EXE\$DEBIT_BYTCNT and EXE\$DEBIT_BYTCNT_BYTLM place the process into a resource wait state, based on the setting of PCB\$V_SSRWAIT, until sufficient quota is returned to the job. EXE\$DEBIT_BYTCNT_NW and EXE\$DEBIT_BYTCNT_BYTLM_NW do not refer to PCB\$V_SSRWAIT and return an error if the process has exceeded its job's quota. These latter routines never wait for sufficient quota.

If bit 31 in R1 is clear, all routines compare the byte count in R1 against IOC\$GW_MAXBUF, returning an error if the system's maximum buffer allotment to a process is exceeded.

Operating System Routines

EXE\$DEBIT_BYTCNT_ALO, EXE\$DEBIT_BYTCNT_BYTLM_ALO

EXE\$DEBIT_BYTCNT_ALO, EXE\$DEBIT_BYTCNT_BYTLM_ALO

Determine whether a job's buffered I/O byte count quota usage permits the process to be granted additional buffered I/O and, if so, allocates the requested amount of nonpaged pool and adjust the job's byte count quota and byte limit.

module

EXSUBROUT

input

| Location | Contents |
|----------------|---|
| R1 | Number of bytes to be allocated (including the 12 bytes required for the buffer's header) and deducted; bit 31, when set, disables the routine's check against IOC\$GW_MAXBUF |
| R4 | Address of current PCB |
| PCB\$L_STS | PCB\$V_SSRWAIT clear if the process should wait for buffered-I/O byte quota; set if resource wait mode is disabled |
| IOC\$GW_MAXBUF | Maximum number of buffered I/O bytes the system allows to a single request |
| JIB\$L_BYTCNT | Job's byte count usage quota |
| JIB\$L_BYTLM | Job's byte limit (used by EXE\$DEBIT_BYTCNT_BYTLM_ALO) |

output

| Location | Contents |
|-----------------------------------|--|
| R0 | SS\$_NORMAL or SS\$_EXQUOTA |
| R1 | Number of bytes deducted; bit 31 cleared |
| R2 | Address of requested buffer |
| R3 | Destroyed |
| JIB\$L_BYTCNT | Updated if successful |
| JIB\$L_BYTLM | Updated if successful (by EXE\$DEBIT_BYTCNT_BYTLM_ALO) |
| IRP\$W_SIZE (in allocated buffer) | Size of requested buffer in bytes |
| IRP\$B_TYPE (in allocated buffer) | DYN\$_BUFIO |

synchronization

EXE\$DEBIT_BYTCNT_ALO and EXE\$DEBIT_BYTCNT_BYTLM_ALO raise IPL to IPL\$_SYNCH and obtain the JIB spin lock in a VMS multiprocessing environment. As a result, their callers cannot be executing above IPL\$_SYNCH or hold any spin lock ranked higher than JIB. (For instance, a driver fork process executing at IPL\$_SYNCH holding the IOLOCK8 fork lock can call these routines. It cannot, however, hold the SCHED spin lock.)

Operating System Routines

EXE\$DEBIT_BYTCNT_ALO, EXE\$DEBIT_BYTCNT_BYTLM_ALO

EXE\$DEBIT_BYTCNT_ALO and EXE\$DEBIT_BYTCNT_BYTLM_ALO return control to their callers at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

EXE\$DEBIT_BYTCNT_ALO checks whether a process has sufficient quota for a buffer of the specified size and, if so, allocates the buffer from nonpaged pool and deducts the corresponding number of bytes from the job's byte count quota. EXE\$DEBIT_BYTCNT_BYTLM_ALO also adjusts the job's byte limit. Both routines round the value specified in R1 up to the nearest 16-byte boundary before applying it to the JIB.

If the process's quota usage is too large, EXE\$DEBIT_BYTCNT_ALO and EXE\$DEBIT_BYTCNT_BYTLM_ALO place the process into a resource wait state, based on the setting of PCB\$V_SSRWAIT, until sufficient quota is returned to the job.

If bit 31 in R1 is clear, these routines compare the byte count in R1 against IOC\$GW_MAXBUF, returning an error if the system's maximum buffer allotment to a process is exceeded.

Operating System Routines

EXE\$FINISHIO, EXE\$FINISHIOC

EXE\$FINISHIO, EXE\$FINISHIOC

Complete the servicing of an I/O request and return status to the I/O status block specified in the request.

module

SYSQIOREQ

input

| Location | Contents |
|----------|---|
| R0 | First longword of status for the I/O status block |
| R1 | Second longword of status for the I/O status block (EXE\$FINISHIO only) |
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |

output

| Location | Contents |
|-------------|---|
| R0 | SS\$_NORMAL |
| IRP\$_IOST1 | First longword of I/O status |
| IRP\$_IOST2 | Second longword of I/O status (cleared by EXE\$FINISHIOC) |
| UCB\$_OPCNT | Incremented |

synchronization

EXE\$FINISHIO and EXE\$FINISHIOC execute at their caller's IPL and raise to fork IPL, acquiring the associated fork lock in a VMS multiprocessing environment. As a result, their callers cannot be executing above fork IPL. A driver usually transfers control to these routines at IPL\$_ASTDEL.

EXE\$FINISHIO and EXE\$FINISHIOC exit at IPL 0 (normal process IPL).

DESCRIPTION

EXE\$FINISHIOC clears the contents of R1. Then, EXE\$FINISHIO or EXE\$FINISHIOC takes the following steps to complete the processing of the I/O request:

- Increases the number of I/O operations completed on the current device in the operation count field of the UCB (UCB\$_OPCNT). This task is performed at fork IPL, holding the associated fork lock in a VMS multiprocessing environment.
- Stores the contents of R0 and R1 in the IRP.
- Inserts the IRP in the local processor's I/O postprocessing queue headed by CPU\$_PSBL.
- If the queue is empty, requests a software interrupt from the local processor at IPL\$_IOPOST.

Operating System Routines

EXE\$FINISHIO, EXE\$FINISHIOC

This interrupt causes postprocessing to occur before the remaining instructions in EXE\$FINISHIO or EXE\$FINISHIOC are executed.

When all I/O postprocessing has been completed, EXE\$FINISHIO or EXE\$FINISHIOC regains control and completes the I/O operation as follows:

- Places status SS\$_NORMAL in R0
- Lowers IPL to zero
- Issues the RET instruction that restores the original access mode of the caller of the \$QIO system service and returns control to the system service dispatcher

The image that issued the \$QIO receives SS\$_NORMAL status in R0, indicating that the I/O request has completed without device-independent error.

Operating System Routines

EXE\$FORK

EXE\$FORK

Creates a fork process on the local processor.

module

FORKCNTRL

macro

FORK

input

| Location | Contents |
|-------------|------------------------------|
| R5 | Address of fork block |
| 00(SP) | Return PC of caller |
| 04(SP) | Return PC of caller's caller |
| FKB\$B_FLCK | Fork lock index or fork IPL |

output

| Location | Contents |
|-------------------------|--------------|
| R3 | Destroyed |
| R4 | Fork IPL |
| FKB\$L_FR3 (UCB\$L_FR3) | R3 of caller |
| FKB\$L_FR4 (UCB\$L_FR4) | R4 of caller |
| FKB\$L_FPC (UCB\$L_FPC) | 00(SP) |

synchronization

EXE\$FORK acquires no spin locks and leaves IPL unchanged. It returns control to its caller's caller.

DESCRIPTION

EXE\$FORK saves the contents of R3 and R4 (in FKB\$L_FR3 and FKB\$L_FR4, respectively) in the fork block specified by R5, and pops the return PC value from the top of the stack into FKB\$L_FPC.

If FKB\$B_FLCK contains a fork lock index, EXE\$FORK determines the fork IPL by using this value as an index into the spin lock IPL vector (SMP\$AR_IPLVEC). EXE\$FORK inserts the fork block into the fork queue on the local processor (headed by CPU\$Q_SWIQFL) corresponding to this IPL. If the queue is empty, EXE\$FORK issues a SOFTINT macro, requesting a software interrupt from the local processor at that fork IPL.

Unlike EXE\$IOfORK, EXE\$FORK does *not* disable timeouts by clearing UCB\$V_TIM in the UCB\$L_STS field.

EXE\$INSERTIRP

Inserts an IRP into the specified queue of IRPs according to the base priority of the process that issued the I/O request.

module

SYSQIOREQ

input

| Location | Contents |
|------------|--|
| R2 | Address of I/O queue listhead for the device |
| R3 | Address of IRP |
| IRP\$B_PRI | Base priority of process requesting the I/O |

output

| Location | Contents |
|-------------------|---|
| R1 | Destroyed |
| PSL <2> (Z bit) | Set if the entry is first in the queue, cleared if at least one entry is already in the queue |
| Pending-I/O queue | IRP inserted |

synchronization

EXE\$INSERTIRP must be called at fork IPL or higher. In a VMS multiprocessing environment, the caller must also hold the associated fork lock. EXE\$INSERTIRP does not alter IPL or acquire any spin locks. It returns to its caller.

DESCRIPTION

EXE\$INSERTIRP determines the position of the specified IRP in the pending-I/O queue according to two factors:

- Priority of the IRP, which is derived from the requesting process's base priority as stored in the IRP\$B_PRI
- Time that the entry is queued; for each priority, the queue is ordered on a first-in/first-out basis

EXE\$INSERTIRP inserts the IRP into the queue at that position, adjusts the queue links, and sets the Z bit in the PSL to indicate the status of the queue.

Operating System Routines

EXE\$INSIOQ, EXE\$INSIOQC

EXE\$INSIOQ, EXE\$INSIOQC

Insert an IRP in a device's pending-I/O queue and call the driver's start-I/O routine if the device is not busy.

module

SYSQIOREQ

input

| Location | Contents |
|-------------|--|
| R3 | Address of IRP |
| R5 | Address of UCB |
| UCB\$_FLCK | Fork lock index |
| UCB\$_STS | UCB\$_BSY set indicates device is busy, clear indicates device is idle |
| UCB\$_IOQFL | Address of pending-I/O queue listhead |
| UCB\$_QLEN | Length of pending-I/O queue |

output

| Location | Contents |
|------------|---|
| R0, R1, R2 | Destroyed. Other registers (used by the driver's start-I/O routine) are destroyed if the start-I/O routine is called. |
| UCB\$_STS | UCB\$_BSY set. |
| UCB\$_QLEN | Incremented. |

synchronization

EXE\$INSIOQ and EXE\$INSIOQC immediately raise to fork IPL and, in a VMS multiprocessing environment, obtain the corresponding fork lock. As a result, their callers must not be executing at an IPL higher than fork IPL or hold a spin lock ranked higher than the fork lock.

EXE\$INSIOQ unconditionally releases ownership of the fork lock before returning control to the caller without possession of the fork lock. If a fork process must retain possession of the fork lock, it should call EXE\$INSIOQC instead.

DESCRIPTION

EXE\$INSIOQ and EXE\$INSIOQC increment UCB\$_QLEN and proceed according to the status of the device (as indicated by UCB\$_BSY in UCB\$_STS) as follows:

- If the device is busy, call EXE\$INSERTIRP to place the IRP on the device's pending-I/O queue.
- If the device is idle, call IOC\$INITIATE to begin device processing of the I/O request immediately. IOC\$INITIATE transfers control to the driver's start-I/O routine.

EXE\$INSTIMQ

Inserts a timer queue element (TQE) into the timer queue.

module

EXSUBROUT

input

| Location | Contents |
|------------------|---|
| R0, R1 | Quadword expiration time for TQE |
| R5 | Address of TQE to be inserted |
| EXE\$GQ_1ST_TIME | Expiration time of first TQE in timer queue |

output

| Location | Contents |
|------------------|---|
| R2, R3 | Destroyed |
| TOE\$Q_TIME | Quadword expiration time for TQE |
| EXE\$GQ_1ST_TIME | Updated if TQE is inserted at the head of the timer queue |

synchronization

EXE\$INSTIMQ immediately raises to IPL\$_TIMER (IPL\$_SYNCH), obtaining the TIMER spin lock in a VMS multiprocessing environment. As a result, its caller must not be executing above IPL\$_SYNCH or hold any spin locks of a higher rank. (For instance, a driver fork process executing at IPL\$_SYNCH holding the IOLOCK8 fork lock can call EXE\$INSTIMQ.)

EXE\$INSTIMQ returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

EXE\$INSTIMQ inserts the specified TQE into the timer queue according to its expiration time. If the expiration time of the new TQE is sooner than that of the first TQE in the queue, EXE\$INSTIMQ raises IPL to interval clock IPL (obtaining the HWCLK spin lock in a VMS multiprocessing environment), inserts it on the head of the queue, and updates EXE\$GQ_1ST_TIME.

Operating System Routines

EXE\$IOFORK

EXE\$IOFORK

Creates a fork process on the local processor for a device driver, disabling timeouts from the associated device.

module

FORKCNTRL

macro

IOFORK

input

| Location | Contents |
|---------------------------|---|
| R5 | Address of fork block (usually the UCB) |
| 00(SP) | Return PC of caller |
| 04(SP) | Return PC of caller's caller |
| FKB\$B_FLCK (UCB\$B_FLCK) | Fork lock index or fork IPL |

output

| Location | Contents |
|-------------------------|---|
| R3 | Destroyed |
| R4 | Fork IPL |
| UCB\$L_STS | UCB\$V_TIM cleared, disabling device timeouts |
| FKB\$L_FR3 (UCB\$L_FR3) | R3 of caller |
| FKB\$L_FR4 (UCB\$L_FR4) | R4 of caller |
| FKB\$L_FPC (UCB\$L_FPC) | 00(SP) |

synchronization

EXE\$IOFORK acquires no spin locks and leaves IPL unchanged. It returns control to its caller's caller.

DESCRIPTION

EXE\$IOFORK first disables timeouts from the target device by clearing UCB\$V_TIM in UCB\$L_STS.

It saves the contents of R3 and R4 (in FKB\$L_FR3 and FKB\$L_FR4, respectively) in the fork block specified by R5, and pops the return PC value from the top of the stack into FKB\$L_FPC.

If FKB\$B_FLCK contains a fork lock index, EXE\$IOFORK determines the fork IPL by using this value as an index into the spin lock IPL vector (SMP\$AR_IPLVEC). EXE\$IOFORK inserts the fork block into the fork queue on the local processor (headed by CPU\$Q_SWIQFL) corresponding to this IPL. If the queue is empty, EXE\$IOFORK issues a SOFTINT macro, requesting a software interrupt from the local processor at that fork IPL.

EXE\$MODIFY

Translates a logical read or write function into a physical read or write function, transfers \$QIO system service parameters to the IRP, validates and prepares a user buffer, and proceeds with or aborts a direct-I/O, DMA read/write operation.

module

SYSQIOFDT

input

| Location | Contents |
|-------------|--|
| R3 | Address of IRP. |
| R4 | Address of current PCB. |
| R5 | Address of UCB. |
| R6 | Address of CCB. |
| R7 | Bit number of the I/O function code. |
| R8 | Address of FDT entry for this routine. |
| 00(AP) | Virtual address of buffer (p1). |
| 04(AP) | Number of bytes in transfer (p2 .) The maximum number of bytes that EXE\$MODIFY can transfer is 65,535 (128 pages minus one byte). |
| 12(AP) | Carriage control byte (p4). |
| IRP\$W_FUNC | I/O function code. |

output

| Location | Contents |
|---------------|---|
| R0, R1, R2 | Destroyed |
| IRP\$L_IOST2 | p4 |
| IRP\$W_STS | IRP\$V_FUNC set, indicating a read function |
| IRP\$W_FUNC | Logical read or write function code converted to physical function |
| IRP\$L_SVAPTE | System virtual address of the process page-table entry (PTE) that maps the first page of the buffer |
| IRP\$W_BOFF | Byte offset to start of transfer in page |
| IRP\$L_BCNT | Size of transfer in bytes |

synchronization

EXE\$MODIFY is called as a driver FDT routine at IPL\$_ASTDEL.

DESCRIPTION

A driver uses EXE\$MODIFY as an FDT routine when the driver must both read from and write to the user-specified buffer. Because EXE\$MODIFY transfers control to EXE\$QIODRVPKT if its operations are successful or EXE\$ABORTIO if they are not, it must be the last FDT routine called to perform the preprocessing of I/O read/write requests. A driver cannot use EXE\$MODIFY for buffered I/O operations.

Operating System Routines

EXE\$MODIFY

EXE\$MODIFY performs the following functions:

- Sets IRP\$V_FUNC in IRP\$W_STS to indicate a read function.
- Writes the **p4** argument of the \$QIO request into IRP\$L_IOST2 (IRP\$B_CARCON).
- Translates logical read and write functions to physical read and write functions.
- Examines the size of the transfer, as specified in the **p2** argument of the \$QIO request, and takes one of the following actions:
 - If the transfer byte count is zero, EXE\$MODIFY transfers control to EXE\$QIODRVPKT to deliver the IRP to the driver's start-I/O routine. The driver start-I/O routine should check for zero-length buffers to avoid mapping them to UNIBUS, Q22-bus, MASSBUS, or VAXBI node space. An attempted mapping can cause a system failure.
 - If the byte count is not zero, EXE\$MODIFY loads the byte count and the starting address of the transfer into R1 and R0, respectively, and calls EXE\$MODIFYLOCK.

EXE\$MODIFYLOCK calls EXE\$MODIFYLOCKR. EXE\$MODIFYLOCKR calls EXE\$READCHKR, which performs the following tasks:

- Moves the transfer byte count into IRP\$L_BCNT. If the byte count is negative, it returns SS\$_BADPARAM status to EXE\$MODIFYLOCKR.
- Determines if the specified buffer is write accessible for a read I/O function, with one of the following results:
 - If the buffer allows write access, EXE\$READCHKR sets IRP\$V_FUNC in IRP\$W_STS and returns SS\$_NORMAL to EXE\$MODIFYLOCKR.
 - If the buffer does not allow write access, EXE\$READCHKR returns SS\$_ACCVIO status to EXE\$MODIFYLOCKR.

If EXE\$READCHKR succeeds, EXE\$MODIFYLOCKR moves into IRP\$W_BOFF the byte offset to the start of the buffer and calls MMG\$IOLOCK. MMG\$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:¹

- If MMG\$IOLOCK succeeds, EXE\$MODIFYLOCKR stores in IRP\$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns control to EXE\$MODIFY. EXE\$MODIFY calls EXE\$QIODRVPKT to deliver the IRP to the driver's start-I/O routine.
- If MMG\$IOLOCK fails, it returns SS\$_ACCVIO, SS\$_INSFWSL, or page fault status to EXE\$MODIFYLOCKR.

¹ For read requests, MMG\$IOLOCK performs an optimization for any nonvalid page contained within the buffer. It creates a demand-zero page rather than fault into memory the requested page. However, if the buffer extends to more than one page, this optimization is not possible.

Operating System Routines

EXE\$MODIFY

If either EXE\$READCHKR or MMG\$IOLOCK returns an error status other than a page fault condition, EXE\$MODIFYLOCKR calls EXE\$ABORTIO. In the event of a page fault, EXE\$MODIFYLOCKR adjusts direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the \$QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the \$QIO system service will resubmit the I/O request.

Operating System Routines

EXE\$MODIFYLOCK, EXE\$MODIFYLOCKR

EXE\$MODIFYLOCK, EXE\$MODIFYLOCKR

Validate and prepare a user buffer for a direct-I/O, DMA read/write operation.

module

SYSQIOFDT

input

| Location | Contents |
|----------|-------------------------------------|
| R0 | Virtual address of buffer |
| R1 | Number of bytes in transfer |
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| R6 | Address of CCB |
| R7 | Bit number of the I/O function code |

output

| Location | Contents |
|---------------|---|
| R0 | SS\$_NORMAL |
| R1 | System virtual address of the process page-table entry (PTE) that maps the first page of the buffer |
| R2 | 1, indicating a read function |
| IRP\$W_STS | IRP\$V_FUNC set, indicating a read function |
| IRP\$L_SVAPTE | System virtual address of the PTE that maps the first page of the buffer |
| IRP\$W_BOFF | Byte offset to start of transfer in page |
| IRP\$L_BCNT | Size of transfer in bytes |

synchronization

EXE\$MODIFYLOCK and EXE\$MODIFYLOCKR are called by a driver FDT routine at IPL\$_ASTDEL.

DESCRIPTION

A driver typically calls EXE\$MODIFYLOCKR instead of EXE\$MODIFYLOCK when it must lock multiple areas into memory for a single I/O request and must regain control, if the request is to be aborted, to unlock these areas. A driver uses either of these routines when it must both read and write to the user-specified buffer and it is not desirable to automatically deliver the IRP to the device unit after the buffer has been successfully locked. A driver cannot use EXE\$MODIFYLOCK or EXE\$MODIFYLOCKR for buffered I/O operations.

Operating System Routines

EXE\$MODIFYLOCK, EXE\$MODIFYLOCKR

EXE\$MODIFYLOCK calls EXE\$MODIFYLOCKR.

EXE\$MODIFYLOCKR calls EXE\$READCHKR, which performs the following tasks:

- Moves the transfer byte count into IRP\$L_BCNT. If the byte count is negative, it returns SS\$_BADPARAM status to EXE\$MODIFYLOCKR.
- Determines if the specified buffer is write accessible for a read I/O function, with one of the following results:
 - If the buffer allows write access, EXE\$READCHKR sets IRP\$V_FUNC in IRP\$W_STS and returns SS\$_NORMAL to EXE\$MODIFYLOCKR.
 - If the buffer does not allow write access, EXE\$READCHKR returns SS\$_ACCVIO status to EXE\$MODIFYLOCKR.

If EXE\$READCHKR succeeds, EXE\$MODIFYLOCKR moves into IRP\$W_BOFF the byte offset to the start of the buffer and calls MMG\$IOLOCK, disabling a paging mechanism used in write-only operations. MMG\$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:²

- If MMG\$IOLOCK succeeds, EXE\$MODIFYLOCKR stores in IRP\$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns success status to its caller.
- If MMG\$IOLOCK fails, it returns SS\$_ACCVIO, SS\$_INSFWSL, or page fault status to EXE\$MODIFYLOCKR.

If the initial call was to EXE\$MODIFYLOCK and either EXE\$READCHKR or MMG\$IOLOCK returns an error status other than a page fault condition, EXE\$MODIFYLOCKR calls EXE\$ABORTIO. In the event of a page fault, EXE\$MODIFYLOCKR adjusts direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the \$QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the \$QIO system service will resubmit the I/O request.

If the initial call was to EXE\$MODIFYLOCKR and an error occurs, EXE\$MODIFYLOCKR, by means of a coroutine call, returns control to the driver's FDT routine with status in R0. The driver performs whatever device-specific actions are required to abort the request, preserving the contents of R0 and R1. When the driver issues the RSB instruction, control is returned to EXE\$MODIFYLOCKR. EXE\$MODIFYLOCKR proceeds to abort or resubmit the I/O request.

Otherwise, these routines return success status to their callers.

² For read requests, MMG\$IOLOCK performs an optimization for any nonvalid page contained within the buffer. It creates a demand-zero page rather than fault into memory the requested page. However, if the buffer extends to more than one page, this optimization is not possible.

Operating System Routines

EXE\$MODIFYLOCK, EXE\$MODIFYLOCKR

A driver FDT routine that calls EXE\$MODIFYLOCKR must distinguish between successful and unsuccessful status when it resumes, as shown in the following example:

```
        JSB    G^EXE$MODIFYLOCKR
        BLBS   BUF_LOCK_OK
BUF_LOCK_FAIL:
;
; clean up this $QIO bookkeeping
;
        RSB
BUF_LOCK_OK:
.
.
;
;continue processing this I/O request
;
```

EXE\$ONEPARM

Copies a single \$QIO parameter into the IRP and delivers the IRP to a driver's start-I/O routine.

module

SYSQIOFDT

input

| Location | Contents |
|----------|--|
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| R6 | Address of CCB |
| R7 | Bit number of the I/O function code |
| R8 | Address of FDT entry for this routine |
| 00(AP) | Address of first function-dependent parameter of the \$QIO request (p1) |

output

| Location | Contents |
|--------------|-----------|
| IRP\$L_MEDIA | p1 |

synchronization

EXE\$ONEPARM is called as a driver FDT routine at IPL\$_ASTDEL.

DESCRIPTION

EXE\$ONEPARM processes an I/O function code that requires only one parameter. This parameter should need no checking; for instance, for read or write accessibility. EXE\$ONEPARM stores the parameter, found at 00(AP), in IRP\$L_MEDIA and transfers control to EXE\$QIODRVPKT to deliver the IRP to the driver.

Operating System Routines

EXE\$QIODRVPKT

EXE\$QIODRVPKT

Delivers an IRP to the driver's start-I/O routine or pending-I/O queue, returns success status in R0, lowers IPL to 0, and returns to the system service dispatcher.

module

SYSQIOREQ

input

| Location | Contents |
|--------------|---|
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| UCB\$B_FLCK | Fork lock index or fork IPL |
| UCB\$L_STS | UCB\$V_BSY set if device is busy, clear if device is idle |
| UCB\$L_IOQFL | Address of pending-I/O queue listhead |
| UCB\$W_QLEN | Length of pending-I/O queue |

output

| | |
|-------------|----------------|
| UCB\$L_STS | UCB\$V_BSY set |
| UCB\$W_QLEN | Incremented |

synchronization

EXE\$QIODRVPKT is called by a driver's FDT routine at IPL\$_ASTDEL. It exits at IPL 0 (normal process IPL).

DESCRIPTION

EXE\$QIODRVPKT calls EXE\$INSIOQ. EXE\$INSIOQ checks the status of the device and calls either EXE\$INSERTIRP or IOC\$INITIATE to place the IRP in the device's pending-I/O queue or deliver it to the driver's start-I/O routine, respectively.

When EXE\$INSIOQ returns to EXE\$QIODRVPKT at IPL\$_ASTDEL, EXE\$QIODRVPKT returns control to the system service dispatcher in the following steps:

- 1 Loads SS\$_NORMAL into R0
- 2 Lowers IPL to zero
- 3 Issues the RET instruction that restores the original access mode of the caller of the \$QIO system service and returns control to the system service dispatcher

The image that requested the I/O operation receives status SS\$_NORMAL in R0, indicating that the I/O request has completed without device-independent error.

EXE\$QIORETURN

Sets a success status code in R0, lowers IPL to 0, and returns to the system service dispatcher.

module

SYSQIOREQ

input

| Location | Contents |
|-------------|-----------------------------|
| R5 | Address of UCB |
| UCB\$B_FLCK | Fork lock index or fork IPL |

output

| Location | Contents |
|----------|-------------|
| R0 | SS\$_NORMAL |

synchronization

EXE\$QIORETURN is typically called by a driver FDT routine at IPL\$_ASTDEL. Its caller cannot be executing above fork IPL or hold any spin locks other than the appropriate fork lock.

EXE\$QIORETURN releases any fork lock held by its caller before it issues the RET instruction.

DESCRIPTION

EXE\$QIORETURN performs the following actions:

- Loads SS\$_NORMAL into R0
- Lowers IPL to zero
- Issues the RET instruction that restores the original access mode of the caller of the \$QIO system service and returns control to the system service dispatcher

The image that requested the I/O operation receives status SS\$_NORMAL in R0, indicating that the I/O request has completed without device-independent error.

Operating System Routines

EXE\$READ

EXE\$READ

Translates a logical read function into a physical read function, transfers \$QIO system service parameters to the IRP, validates and prepares a user buffer, and proceeds with or aborts a direct-I/O, DMA read/write operation.

module

SYSQIOFDT

input

| Location | Contents |
|-------------|--|
| R3 | Address of IRP. |
| R4 | Address of current PCB. |
| R5 | Address of UCB. |
| R6 | Address of CCB. |
| R7 | Bit number of the I/O function code. |
| R8 | Address of FDT entry for this routine. |
| 00(AP) | Virtual address of buffer (p1). |
| 04(AP) | Number of bytes in transfer (p2). The maximum number of bytes that EXE\$READ can transfer is 65,535 (128 pages minus one byte). |
| 12(AP) | Carriage control byte (p4). |
| IRP\$W_FUNC | I/O function code. |

output

| Location | Contents |
|---------------|---|
| R0, R1, R2 | Destroyed |
| IRP\$B_IOST2 | p4 |
| IRP\$W_STS | IRP\$V_FUNC set, indicating a read function |
| IRP\$W_FUNC | Logical read function code converted to physical |
| IRP\$L_SVAPTE | System virtual address of the process page-table entry (PTE) that maps the first page of the buffer |
| IRP\$W_BOFF | Byte offset to start of transfer in page |
| IRP\$L_BCNT | Size of transfer in bytes |

synchronization

EXE\$READ is called as a driver FDT routine at IPL\$_ASTDEL.

DESCRIPTION

A driver uses EXE\$READ as an FDT routine when the driver must write to the user-specified buffer. Because EXE\$READ transfers control to EXE\$QIODRVPKT if its operations are successful or EXE\$ABORTIO if they are not, it must be the last FDT routine called to perform the preprocessing of read I/O requests. A driver cannot use EXE\$READ for buffered-I/O operations.

Operating System Routines

EXE\$READ

EXE\$READ performs the following functions:

- Sets IRP\$V_FUNC in IRP\$W_STS to indicate a read function
- Writes the **p4** argument of the \$QIO request into IRP\$L_IOST2 (IRP\$B_CARCON).
- Translates a logical read function to a physical read function.
- Examines the size of the transfer, as specified in the **p2** argument of the \$QIO request, and takes one of the following actions:
 - If the transfer byte count is zero, EXE\$READ transfers control to EXE\$QIODRVPKT to deliver the IRP to the driver's start-I/O routine. The driver start-I/O routine should check for zero-length buffers to avoid mapping them to UNIBUS, Q22-bus, MASSBUS, or VAXBI node space. An attempted mapping can cause a system failure.
 - If the byte count is not zero, EXE\$READ loads the byte count and the starting address of the transfer into R1 and R0, respectively, and calls EXE\$READLOCK.

EXE\$READLOCK calls EXE\$READLOCKR.

EXE\$READLOCKR calls EXE\$READCHKR, which performs the following tasks:

- Moves the transfer byte count into IRP\$L_BCNT. If the byte count is negative, it returns SS\$_BADPARAM status to EXE\$READLOCKR.
- Determines whether the specified buffer is write accessible for a read I/O function, with one of the following results:
 - If the buffer allows write access, EXE\$READCHKR sets IRP\$V_FUNC in IRP\$W_STS, and returns SS\$_NORMAL to EXE\$READLOCKR.
 - If the buffer does not allow write access, EXE\$READCHKR returns SS\$_ACCVIO status to EXE\$READLOCKR.

If EXE\$READCHKR succeeds, EXE\$READLOCKR moves into IRP\$W_BOFF the byte offset to the start of the buffer and calls MMG\$IOLOCK. MMG\$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:³

- If MMG\$IOLOCK succeeds, EXE\$READLOCKR stores in IRP\$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns control to EXE\$READ. EXE\$READ transfers control to EXE\$QIODRVPKT to deliver the IRP to the driver's start-I/O routine.
- If MMG\$IOLOCK fails, it returns SS\$_ACCVIO, SS\$_INSFWSL, or page fault status to EXE\$READLOCKR.

³ For read requests, MMG\$IOLOCK performs an optimization for any nonvalid page contained within the buffer. It creates a demand-zero page rather than fault into memory the requested page. However, if the buffer extends to more than one page, this optimization is not possible.

Operating System Routines

EXE\$READ

If either EXE\$READCHKR or MMG\$IOLOCK returns an error status other than a page fault condition, EXE\$READLOCKR transfers control to EXE\$ABORTIO. In the event of a page fault, EXE\$READLOCKR adjusts direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the \$QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the \$QIO system service will resubmit the I/O request.

EXE\$READCHK, EXE\$READCHKR

Verify that a process has write access to the pages in the buffer specified in a \$QIO request.

module

SYSQIOFDT

input

| Location | Contents |
|----------|---------------------------|
| R0 | Virtual address of buffer |
| R1 | Size of transfer in bytes |
| R3 | Address of IRP |

output

| Location | Contents |
|-------------|--|
| R0 | Virtual address of buffer (EXE\$READCHK), SS\$_NORMAL (EXE\$READCHKR), or error status |
| R1 | Size of transfer in bytes |
| R2 | 1, indicating a read function |
| R3 | Address of IRP |
| IRP\$W_STS | IRP\$V_FUNC set, indicating a read function |
| IRP\$L_BCNT | Size of transfer in bytes |

synchronization

EXE\$READCHK and EXE\$READCHKR are called by a driver FDT routine at IPL\$_ASTDEL.

DESCRIPTION

A driver uses either of these routines to check the write accessibility of a user-specified buffer. A driver typically calls EXE\$READCHKR instead of EXE\$READCHK when it must regain control before the request is aborted in the event the buffer is inaccessible.

EXE\$READCHK calls EXE\$READCHKR.

EXE\$READCHKR performs the following tasks:

- Moves the transfer byte count into IRP\$L_BCNT. If the byte count is negative, it returns SS\$_BADPARAM status to its caller.
- Determines whether the specified buffer is write accessible for a read I/O function, with one of the following results:
 - If the buffer allows write access, EXE\$READCHKR sets IRP\$V_FUNC in IRP\$W_STS and returns SS\$_NORMAL to its caller.
 - If the buffer does not allow write access, EXE\$READCHKR returns SS\$_ACCVIO status to its caller.

Operating System Routines

EXE\$READCHK, EXE\$READCHKR

If the initial call was to EXE\$READCHK, and EXE\$READCHKR returns error status, EXE\$READCHK transfers control to EXE\$ABORTIO to terminate the I/O request. If the initial call was to EXE\$READCHKR, and an error occurs, EXE\$READCHKR returns control to the driver. Otherwise, these routines return success status to their callers.

A driver FDT routine that calls EXE\$READCHKR must distinguish between successful and unsuccessful status when it resumes, as shown in the following example:

```
        JSB    G^EXE$READCHKR
        BLBS   BUF_ACCESS_OK
BUF_ACCESS_FAIL:
;
; clean up this $QIO bookkeeping
;
        JSB    G^EXE$ABORTIO
BUF_ACCESS_OK:
.
.
;
;continue processing this I/O request
;
```

EXE\$READLOCK, EXE\$READLOCKR

Validate and prepare a user buffer for a direct-I/O, DMA read operation.

module

SYSQIOFDT

input

| Location | Contents |
|----------|-------------------------------------|
| R0 | Virtual address of buffer |
| R1 | Number of bytes in transfer |
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| R6 | Address of CCB |
| R7 | Bit number of the I/O function code |

output

| Location | Contents |
|---------------|---|
| R0 | SS\$_NORMAL |
| R1 | System virtual address of the process page-table entry (PTE) that maps the first page of the buffer |
| R2 | 1, indicating a read function |
| IRP\$W_STS | IRP\$V_FUNC set, indicating a read function |
| IRP\$L_SVAPTE | System virtual address of the PTE that maps the first page of the buffer |
| IRP\$W_BOFF | Byte offset to start of transfer in page |
| IRP\$L_BCNT | Size of transfer in bytes |

synchronization

EXE\$READLOCK and EXE\$READLOCKR are called by a driver FDT routine at IPL\$_ASTDEL.

DESCRIPTION

A driver typically calls EXE\$READLOCKR instead of EXE\$READLOCK when it must lock multiple areas into memory for a single I/O request and must regain control, if the request is to be aborted, to unlock these areas. A driver uses either of these routines when it must write to the user-specified buffer and it is not desirable to automatically deliver the IRP to the device unit after the buffer has been successfully locked. A driver cannot use EXE\$READLOCK or EXE\$READLOCKR for buffered I/O operations.

EXE\$READLOCK calls EXE\$READLOCKR.

EXE\$READLOCKR calls EXE\$READCHKR, which performs the following tasks:

- Moves the transfer byte count into IRP\$L_BCNT. If the byte count is negative, it returns SS\$_BADPARAM status to EXE\$READLOCKR.

Operating System Routines

EXE\$READLOCK, EXE\$READLOCKR

- Determines whether the specified buffer is write accessible for a read I/O function, with one of the following results:
 - If the buffer allows write access, EXE\$READCHKR sets IRP\$V_FUNC in IRP\$W_STS and returns SS\$_NORMAL to EXE\$READLOCKR.
 - If the buffer does not allow write access, EXE\$READCHKR returns SS\$_ACCVIO status to EXE\$READLOCKR.

If EXE\$READCHKR succeeds, EXE\$READLOCKR moves into IRP\$W_BOFF the byte offset to the start of the buffer and calls MMG\$IOLOCK. MMG\$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:⁴

- If MMG\$IOLOCK succeeds, EXE\$READLOCKR stores in IRP\$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns success status to its caller.
- If MMG\$IOLOCK fails, it returns SS\$_ACCVIO, SS\$_INSFWSL, or page fault status to EXE\$READLOCKR.

If the initial call was to EXE\$READLOCK and either EXE\$READCHKR or MMG\$IOLOCK returns an error status other than a page fault condition, EXE\$READLOCKR transfers control to EXE\$ABORTIO. In the event of a page fault, EXE\$READLOCKR adjusts direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the \$QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the \$QIO system service will resubmit the I/O request.

If the initial call was to EXE\$READLOCKR and an error occurs, EXE\$READLOCKR, by means of a coroutine call, returns control to the driver's FDT routine with status in R0. The driver performs whatever device-specific actions are required to abort the request, preserving the contents of R0 and R1. When the driver issues the RSB instruction, control is returned to EXE\$READLOCKR. EXE\$READLOCKR proceeds to abort or resubmit the I/O request.

Otherwise, these routines return success status to their callers.

⁴ For read requests, MMG\$IOLOCK performs an optimization for any nonvalid page contained within the buffer. It creates a demand-zero page rather than fault into memory the requested page. However, if the buffer extends to more than one page, this optimization is not possible.

Operating System Routines

EXE\$READLOCK, EXE\$READLOCKR

A driver FDT routine that calls EXE\$READLOCKR must distinguish between successful and unsuccessful status when it resumes, as shown in the following example:

```
        JSB    G^EXE$READLOCKR
        BLBS   BUF_LOCK_OK
BUF_LOCK_FAIL:
;
; clean up this $QIO bookkeeping
;
        RSB
BUF_LOCK_OK:
.
.
;
; continue processing this I/O request
;
```

Operating System Routines

EXE\$SENSEMODE

EXE\$SENSEMODE

Copies device-dependent characteristics from the device's UCB into R1, writes a success code into R0, and completes the I/O operation.

module

SYSQIOFDT

input

| Location | Contents |
|------------------|--|
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| R6 | Address of CCB |
| R7 | Bit number of the I/O function code |
| R8 | Address of FDT entry for this routine |
| 00(AP) | Address of first function-dependent parameter of the \$QIO request |
| UCB\$Q_DEVDEPEND | Device-dependent status |

output

| Location | Contents |
|----------|-------------------------|
| R0 | SS\$_NORMAL |
| R1 | Device-dependent status |

synchronization

EXE\$SENSEMODE is called as a driver FDT routine at IPL\$_ASTDEL.

DESCRIPTION

A driver uses EXE\$SENSEMODE as an FDT routine to process the sense-device-mode (IO\$_SENSEMODE) and sense-device-characteristics (IO\$_SENSECHAR) I/O functions.

EXE\$SENSEMODE loads the contents of UCB\$Q_DEVDEPEND into R1, places SS\$_NORMAL status into R0, and transfers control to EXE\$FINISHIO to insert the IRP in the I/O postprocessing queue.

EXE\$SETCHAR, EXE\$SETMODE

Write device-specific status and control information into the device's UCB and complete the I/O request (EXE\$SETCHAR); or write the information into the IRP and deliver the IRP to the driver's start-I/O routine (EXE\$SETMODE).

module

SYSQIOFDT

input

| Location | Contents |
|-----------------|--|
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| R6 | Address of CCB |
| R7 | Bit number of the I/O function code |
| R8 | Address of FDT entry for this routine |
| 00(AP) | Address of location containing device characteristics quadword (p1) |
| UCB\$B_DEVCLASS | Device class |

output

| Location | Contents |
|------------------|---|
| R0 | SS\$_NORMAL, SS\$_ACCVIO, or SS\$_ILLIOFUNC |
| UCB\$B_DEVCLASS | Byte 0 of quadword (EXE\$SETCHAR, IO\$_SETCHAR function only) |
| UCB\$B_DEVTYPE | Byte 1 of quadword (EXE\$SETCHAR, IO\$_SETCHAR function only) |
| UCB\$W_DEVBUFSIZ | Bytes 2 and 3 of quadword (EXE\$SETCHAR) |
| UCB\$Q_DEVDEPEND | Bytes 4 through 7 of quadword (EXE\$SETCHAR) |
| IRP\$L_MEDIA | First longword of device characteristics (EXE\$SETMODE) |
| IRP\$L_MEDIA+4 | Second longword of device characteristics (EXE\$SETMODE) |

synchronization

EXE\$SETCHAR or EXE\$SETMODE is called as a driver FDT routine at IPL\$_ASTDEL.

DESCRIPTION

A driver uses EXE\$SETCHAR or EXE\$SETMODE as an FDT routine to process the set-device-mode (IO\$_SETMODE) and set-device-characteristics (IO\$_SETCHAR) functions. If setting device characteristics requires device activity or synchronization with fork processing, the driver's FDT entry *must* specify EXE\$SETMODE. Otherwise, it can specify EXE\$SETCHAR.

Operating System Routines

EXE\$SETCHAR, EXE\$SETMODE

EXE\$SETCHAR and EXE\$SETMODE examine the current value of UCB\$B_DEVCLASS to determine whether the device permits the specified function. If the device class is disk (DC\$_DISK), the routines place SS\$_ILLIOFUNC status in R0 and transfer control to EXE\$ABORTIO to terminate the request.

EXE\$SETCHAR and EXE\$SETMODE then ensure that the process has read access to the quadword containing the new device characteristics. If it does not, the routines place SS\$_ACCVIO status in R0 and transfer control to EXE\$ABORTIO to terminate the request.

If the request passes these checks, EXE\$SETCHAR and EXE\$SETMODE proceed as follows:

- EXE\$SETCHAR stores the specified characteristics in the UCB. For an IO\$_SETCHAR function, the device type and class fields (UCB\$B_DEVCLASS and UCB\$B_DEVTYPE, respectively) receive the first word of data. For both IO\$_SETCHAR and IO\$_SETMODE functions, EXE\$SETCHAR writes the second word into the default-buffer-size field (UCB\$W_DEVBUSIZ) and the third and fourth words into the device-dependent-characteristics field (UCB\$Q_DEVDEPEND).

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

- EXE\$SETMODE stores the specified quadword of characteristics in IRP\$L_MEDIA, places normal completion status (SS\$_NORMAL) in R0, and transfers control to EXE\$QIODRVPKT to deliver the IRP to the driver's start-I/O routine.

The driver's start-I/O routine copies data from IRP\$L_MEDIA and the following longword into UCB\$W_DEVBUSIZ, UCB\$Q_DEVDEPEND, and, if the I/O function is IO\$_SETCHAR, UCB\$B_DEVCLASS and UCB\$B_DEVTYPE as well.

EXE\$SNDEVMSG

Builds and sends a device-specific message to the mailbox of a system process, such as the job controller or OPCOM.

module

MBDRIVER

input

| Location | Contents |
|------------------------------------|------------------------|
| R3 | Address of mailbox UCB |
| R4 | Message type |
| R5 | Address of device UCB |
| UCB\$W_UNIT | Device unit number |
| UCB\$L_DDB | Address of device DDB |
| DDB\$T_NAME and mailbox UCB fields | Device controller name |

output

| Location | Contents |
|---------------|---|
| R0 | SS\$_NORMAL, SS\$_MBTOOSML, SS\$_MBFULL, SS\$_INSFMEM, or SS\$_NOPRIV |
| R1 through R4 | Destroyed |

synchronization

Because EXE\$SNDEVMSG raises IPL to IPL\$_MAILBOX and obtains the MAILBOX spin lock in a VMS multiprocessing environment, its caller cannot be executing above IPL\$_MAILBOX. EXE\$SNDEVMSG returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

EXE\$SNDEVMSG builds a 32-byte message on the stack that includes the following information:

| Bytes | Contents |
|--------------|--|
| 0 and 1 | Low word of R4 (message type) |
| 2 and 3 | Device unit number (UCB\$W_UNIT) |
| 4 through 31 | Counted string of device controller name, formatted as <i>node\$controller</i> for clusterwide devices |

EXE\$SNDEVMSG then calls EXE\$WRTMAILBOX to send the message to a mailbox.

Operating System Routines

EXE\$SNDEVMSG

EXE\$SNDEVMSG can fail for any of the following reasons:

- The message is too large for the mailbox (SS\$_MBTOOSML).
- The message mailbox is full of messages (SS\$_MBFULL).
- The system is unable to allocate memory for the message (SS\$_INSFMEM).
- The caller lacks privilege to write to the mailbox (SS\$_NOPRIV).

EXE\$WRITE

Translates a logical write function into a physical write function, transfers \$QIO system service parameters to the IRP, validates and prepares a user buffer, and proceeds with or aborts a direct-I/O, DMA read/write operation.

module

SYSQIOFDT

input

| Location | Contents |
|-------------|---|
| R3 | Address of IRP. |
| R4 | Address of current PCB. |
| R5 | Address of UCB. |
| R6 | Address of CCB. |
| R7 | Bit number of the I/O function code. |
| R8 | Address of FDT entry for this routine. |
| 00(AP) | Virtual address of buffer (p1). |
| 04(AP) | Number of bytes in transfer (p2). The maximum number of bytes that EXE\$WRITE can transfer is 65,535 (128 pages minus one byte). |
| 12(AP) | Carriage control byte (p4). |
| IRP\$W_FUNC | I/O function code. |

output

| Location | Contents |
|---------------|---|
| R0, R1, R2 | Destroyed |
| IRP\$L_IOST2 | p4 |
| IRP\$W_FUNC | Logical read function code converted to physical |
| IRP\$W_STS | IRP\$V_FUNC clear, indicating a write function |
| IRP\$L_SVAPTE | System virtual address of the process page-table entry (PTE) that maps the first page of the buffer |
| IRP\$W_BOFF | Byte offset to start of transfer in page |
| IRP\$L_BCNT | Size of transfer in bytes |

synchronization

EXE\$WRITE is called as a driver FDT routine at IPL\$_ASTDEL.

DESCRIPTION

A driver uses EXE\$WRITE as an FDT routine when the driver must read from the user-specified buffer. Because EXE\$WRITE transfers control to EXE\$QIODRVPKT if its operations are successful or EXE\$ABORTIO if they are not, it must be the last FDT routine called to perform the preprocessing of write I/O requests. A driver cannot use EXE\$WRITE for buffered I/O operations.

Operating System Routines

EXE\$WRITE

EXE\$WRITE performs the following functions:

- Writes the **p4** argument of the \$QIO request into IRP\$L_IOST2 (IRP\$B_CARCON).
- Translates a logical write function to a physical write function.
- Examines the size of the transfer, as specified in the **p2** argument of the \$QIO request, and takes one of the following actions:
 - If the transfer byte count is zero, EXE\$WRITE transfers control to EXE\$QIODRVPKT to deliver the IRP to the driver's start-I/O routine. The driver start-I/O routine should check for zero-length buffers to avoid mapping them to UNIBUS, Q22 bus, MASSBUS, or VAXBI node space. An attempted mapping can cause a system failure.
 - If the byte count is not zero, EXE\$READ loads the byte count and the starting address of the transfer into R1 and R0, respectively, and calls EXE\$WRITELOCK.

EXE\$WRITELOCK calls EXE\$WRITELOCKR.

EXE\$WRITELOCKR calls EXE\$WRITECHKR, which performs the following tasks:

- Moves the transfer byte count into IRP\$L_BCNT. If the byte count is negative, it returns SS\$_BADPARAM status to EXE\$WRITELOCKR.
- Determines whether the specified buffer is read accessible for a write I/O function, with one of the following results:
 - If the buffer allows read access, EXE\$WRITECHKR returns SS\$_NORMAL to EXE\$WRITELOCKR.
 - If the buffer does not allow read access, EXE\$WRITECHKR returns SS\$_ACCVIO status to EXE\$WRITELOCKR.

If EXE\$WRITECHKR succeeds, EXE\$WRITELOCKR moves into IRP\$W_BOFF the byte offset to the start of the buffer and calls MMG\$IOLOCK. MMG\$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG\$IOLOCK succeeds, EXE\$WRITELOCKR stores in IRP\$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns control to EXE\$WRITE. EXE\$WRITE transfers control to EXE\$QIODRVPKT to deliver the IRP to the driver's start-I/O routine.
- If MMG\$IOLOCK fails, it returns SS\$_ACCVIO, SS\$_INSFWSL, or page fault status to EXE\$WRITELOCKR.

If either EXE\$WRITECHKR or MMG\$IOLOCK returns an error status, EXE\$WRITELOCKR transfers control to EXE\$ABORTIO.

EXE\$WRITECHK, EXE\$WRITECHKR

Verify that a process has read access to the pages in the buffer specified in a \$QIO request.

module

SYSQIOFDT

input

| Location | Contents |
|----------|---------------------------|
| R0 | Virtual address of buffer |
| R1 | Size of transfer in bytes |
| R3 | Address of IRP |

output

| Location | Contents |
|------------|--|
| R0 | Virtual address of buffer (EXE\$WRITECHK), SS\$_NORMAL (EXE\$WRITECHKR), or error status |
| R1 | Size of transfer in bytes |
| R2 | 0, indicating a write function |
| IRP\$_STG | IRP\$_FUNC clear, indicating a write function |
| IRP\$_BCNT | Size of transfer in bytes |

synchronization

EXE\$WRITECHK and EXE\$WRITECHKR are called by a driver FDT routine at IPL\$_ASTDEL.

DESCRIPTION

A driver uses either of these routines to check the read accessibility of a user-specified buffer. A driver typically calls EXE\$WRITECHKR instead of EXE\$WRITECHK when it must regain control before the request is aborted in the event the buffer is inaccessible.

EXE\$WRITECHK calls EXE\$WRITECHKR.

EXE\$WRITECHKR performs the following tasks:

- Moves the transfer byte count into IRP\$_BCNT. If the byte count is negative, it returns SS\$_BADPARAM status to its caller.
- Determines if the specified buffer is read accessible for a write I/O function, with one of the following results:
 - If the buffer allows read access, EXE\$WRITECHKR returns SS\$_NORMAL to its caller.
 - If the buffer does not allow read access, EXE\$WRITECHKR returns SS\$_ACCVIO status to its caller.

Operating System Routines

EXE\$WRITECHK, EXE\$WRITECHKR

If the initial call was to EXE\$WRITECHK, and EXE\$WRITECHKR returns error status, EXE\$WRITECHK transfers control to EXE\$ABORTIO to terminate the I/O request. If the initial call was to EXE\$WRITECHKR, and an error occurs, EXE\$WRITECHKR returns control to the driver. Otherwise, these routines return success status to their callers.

A driver FDT routine that calls EXE\$WRITECHKR must distinguish between successful and unsuccessful status when it resumes, as shown in the following example:

```
        JSB    G^EXE$WRITECHKR
        BLBS   BUF_ACCESS_OK
BUF_ACCESS_FAIL:
;
; clean up this $QIO bookkeeping
;
        JSB    G^EXE$ABORTIO
BUF_ACCESS_OK:
.
.
;
;continue processing this I/O request
;
```

EXE\$WRITELOCK, EXE\$WRITELOCKR

Validate and prepare a user buffer for a direct-I/O, DMA write operation.

module

SYSQIOFDT

input

| Location | Contents |
|-----------------|-------------------------------------|
| R0 | Virtual address of buffer |
| R1 | Number of bytes in transfer |
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| R6 | Address of CCB |
| R7 | Bit number of the I/O function code |

output

| Location | Contents |
|-----------------|---|
| R0 | SS\$_NORMAL |
| R1 | System virtual address of the process page-table entry (PTE) that maps the first page of the buffer |
| R2 | 0, indicating a write function |
| IRP\$_STS | IRP\$_FUNC clear, indicating a write function |
| IRP\$_SVAPTE | System virtual address of the PTE that maps the first page of the buffer |
| IRP\$_BOFF | Byte offset to start of transfer in page |
| IRP\$_BCNT | Size of transfer in bytes |

synchronization

EXE\$WRITELOCK and EXE\$WRITELOCKR are called by a driver FDT routine at IPL\$_ASTDEL.

DESCRIPTION

A driver typically calls EXE\$WRITELOCKR instead of EXE\$WRITELOCK when it must lock multiple areas into memory for a single I/O request and must regain control, if the request is to be aborted, to unlock these areas. A driver uses either of these routines when it must read from the user-specified buffer and it is not desirable to automatically deliver the IRP to the device unit after the buffer has been successfully locked. A driver cannot use EXE\$WRITELOCK or EXE\$WRITELOCKR for buffered I/O operations.

EXE\$WRITELOCK calls EXE\$WRITELOCKR.

EXE\$WRITELOCKR calls EXE\$WRITECHKR, which performs the following tasks:

- Moves the transfer byte count into IRP\$_BCNT. If the byte count is negative, it returns SS\$_BADPARAM status to EXE\$WRITELOCKR.

Operating System Routines

EXE\$WRITELOCK, EXE\$WRITELOCKR

- Determines if the specified buffer is write accessible for a write I/O function, with one of the following results:
 - If the buffer allows read access, EXE\$WRITECHKR returns SS\$_NORMAL to EXE\$WRITELOCKR.
 - If the buffer does not allow read access, EXE\$WRITECHKR returns SS\$_ACCVIO status to EXE\$WRITELOCKR.

If EXE\$WRITECHKR succeeds, EXE\$WRITELOCKR moves into IRP\$W_BOFF the byte offset to the start of the buffer and calls MMG\$IOLOCK. MMG\$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG\$IOLOCK succeeds, EXE\$WRITELOCKR stores in IRP\$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns success status to its caller.
- If MMG\$IOLOCK fails, it returns SS\$_ACCVIO, SS\$_INSFWSL, or page fault status to EXE\$WRITELOCKR.

If the initial call was to EXE\$WRITELOCK and either EXE\$WRITECHKR or MMG\$IOLOCK returns an error status other than a page fault condition, EXE\$WRITELOCKR transfers control to EXE\$ABORTIO. In the event of a page fault, EXE\$WRITELOCKR adjusts direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the \$QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the \$QIO system service will resubmit the I/O request.

If the initial call was to EXE\$WRITELOCKR and an error occurs, EXE\$WRITELOCKR, by means of a coroutine call, returns control to the driver's FDT routine with status in R0. The driver performs whatever device-specific actions are required to abort the request, preserving the contents of R0 and R1. When the driver issues the RSB instruction, control is returned to EXE\$WRITELOCKR. EXE\$WRITELOCKR proceeds to abort the I/O request.

Otherwise, these routines return success status to their callers.

A driver FDT routine that calls EXE\$WRITELOCKR must distinguish between successful and unsuccessful status when it resumes, as shown in the following example:

```
        JSB      G^EXE$WRITELOCKR
        BLBS    BUF_LOCK_OK
BUF_LOCK_FAIL:
;
; clean up this $QIO bookkeeping
;
        RSB
BUF_LOCK_OK:
.
.
;
;continue processing this I/O request
;
```

EXE\$WRTMAILBOX

Sends a message to a mailbox.

module

MBDRIVER

input

| Location | Contents |
|--------------------|------------------------|
| R3 | Message size |
| R4 | Message address |
| R5 | Address of mailbox UCB |
| Mailbox UCB fields | |

output

| Location | Contents |
|-----------|---|
| R0 | SS\$_NORMAL, SS\$_MBTOOSML, SS\$_MBFULL, SS\$_INSFMEM, or SS\$_NOPRIV |
| R1 and R2 | Destroyed |

synchronization

Because EXE\$WRTMAILBOX raises IPL to IPL\$_MAILBOX and obtains the MAILBOX spin lock in a VMS multiprocessing environment, its caller cannot be executing above IPL\$_MAILBOX. EXE\$WRTMAILBOX returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

EXE\$WRTMAILBOX checks fields in the mailbox UCB (UCB\$_BUFQUO, UCB\$_DEVBUFSIZ) to determine whether it can deliver a message of the specified size to the mailbox. It also checks fields in the associated ORB to determine whether the caller is sufficiently privileged to write to the mailbox. Finally, it calls EXE\$ALONONPAGED to allocate a block of nonpaged pool to contain the message. If it fails any of these operations, EXE\$WRTMAILBOX returns error status to its caller.

If it is successful thus far, EXE\$WRTMAILBOX creates a message and delivers it to the mailbox's message queue, adjusts its UCB fields accordingly, and returns success status to its caller.

Operating System Routines

EXE\$ZEROPARM

EXE\$ZEROPARM

Processes an I/O function code that requires no parameters.

module

SYSQIOFDT

input

| | |
|----|---------------------------------------|
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| R6 | Address of CCB |
| R7 | Bit number of the I/O function code |
| R8 | Address of FDT entry for this routine |

output

| Location | Contents |
|--------------|----------|
| IRP\$L_MEDIA | 0 |

synchronization

EXE\$ZEROPARM is called as a driver FDT routine at IPL\$_ASTDEL.

DESCRIPTION

EXE\$ZEROPARM processes an I/O function code that describes an I/O operation completely without any additional function-specific arguments. It clears IRP\$L_MEDIA and transfers control to EXE\$QIODRVPKT to deliver the IRP to the driver.

Operating System Routines

IOC\$ALOALTMAP, IOC\$ALOALTMAPN, IOC\$ALOALTMAPSP

IOC\$ALOALTMAP, IOC\$ALOALTMAPN, IOC\$ALOALTMAPSP

Allocate a set of Q22-bus alternate map registers.

module

[SYSLOA]MAPSUBxxx

input

| Location | Contents |
|--|--|
| R3 | Number of alternate map registers to allocate (IOC\$ALOALTMAPN and IOC\$ALOALTMAPSP only). The value should account for one extra register needed to prevent a transfer overrun. |
| R4 | Number of first alternate map register to allocate (IOC\$ALOALTMAPSP only). |
| R5 | Address of UCB. |
| UCB\$W_BCNT | Transfer byte count (IOC\$ALOALTMAP only). |
| UCB\$W_BOFF | Byte offset in page (IOC\$ALOALTMAP only). |
| UCB\$L_CRB | Address of CRB. |
| CRB\$L_INTD+ VEC\$L_ADP | Address of ADP. |
| CRB\$L_INTD+ VEC\$W_MAPALT | VEC\$V_ALTLOCK set indicates that alternate map registers have been permanently allocated to this controller. |
| ADP\$W_MR2NREGAR, ADP\$W_MR2FREGAR, ADP\$L_MR2ACTMDR | Alternate map register descriptor arrays. |

output

| Location | Contents |
|--|---|
| R0 | SS\$_NORMAL, SS\$_INSFMAPREG, or SS\$_SSFAL |
| R1 | Destroyed |
| R2 | Address of ADP |
| CRB\$L_INTD+ VEC\$W_NUMALT | Number of alternate map registers allocated |
| CRB\$L_INTD+ VEC\$W_MAPALT | Starting alternate map register number |
| ADP\$W_MR2NREGAR, ADP\$W_MR2FREGAR, ADP\$L_MR2ACTMDR | Updated |

Operating System Routines

IOC\$ALOALTMAP, IOC\$ALOALTMAPN, IOC\$ALOALTMAPSP

synchronization

Callers of IOC\$ALOALTMAP, IOC\$ALOALTMAPN, or IOC\$ALOALTMAPSP may be executing at fork IPL or above and must hold the corresponding fork lock in a VMS multiprocessing environment. Each routine returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

IOC\$ALOALTMAP, IOC\$ALOALTMAPN, and IOC\$ALOALTMAPSP allocate a contiguous set of Q22-bus alternate map registers (registers 496 to 8191) and record the allocation in the ADP and CRB. These routines differ in the way in which they determine the number and location of the alternate map registers they allocate:

- IOC\$ALOALTMAP calculates the number of needed map registers using the values contained in UCB\$W_BCNT and UCB\$W_BOFF. It automatically allocates one extra map register. When it is later called by the driver, IOC\$LOADALTMAP marks this register invalid to prevent a transfer overrun.
- IOC\$ALOALTMAPN uses the value in R3 as the number of required registers.
- IOC\$ALOALTMAPSP uses the value in R3 as the number of required registers and attempts to allocate these registers starting at the one indicated by R4.

If an odd number of map registers is required, these routines round this value up to an even multiple.

If alternate map registers have been permanently allocated to the controller, IOC\$ALOALTMAP, IOC\$ALOALTMAPN, or IOC\$ALOALTMAPSP returns successfully to its caller without allocating the requested map registers. Otherwise, it searches the alternate map register descriptor arrays for the required number of map registers. If there are not enough contiguous map registers available, the routine returns SS\$_INSFMAPREG status.

If the VAX system does not support alternate map registers, the routine exits with SS\$_SSFAIL status.

Operating System Routines

IOC\$ALOUBAMAP, IOC\$ALOUBAMAPN

IOC\$ALOUBAMAP, IOC\$ALOUBAMAPN

Allocate a set of UNIBUS map registers or a set of the first 496 Q22-bus map registers.

module

IOSUBNPAG

input

| Location | Contents |
|---|---|
| R3 | Number of map registers to allocate (IOC\$ALOUBAMAPN only). The value should account for one extra register needed to prevent a transfer overrun. |
| R5 | Address of UCB. |
| UCB\$_BCNT | Transfer byte count (IOC\$ALOUBAMAP only). |
| UCB\$_BOFF | Byte offset in page (IOC\$ALOUBAMAP only). |
| UCB\$_CRB | Address of CRB. |
| CRB\$_INTD+ VEC\$_ADP | Address of ADP. |
| CRB\$_INTD+ VEC\$_MAPREG | VEC\$_MAPLOCK set indicates that map registers have been permanently allocated to this controller. |
| ADP\$_MRNREGARY, ADP\$_MRFREGARY, ADP\$_MRACTMDRS | Map register descriptor arrays. |

output

| Location | Contents |
|---|-----------------------------------|
| R0 | SS\$_NORMAL or 0 |
| R1 | Destroyed |
| R2 | Address of ADP |
| CRB\$_INTD+VEC\$_ NUMREG | Number of map registers allocated |
| CRB\$_INTD+VEC\$_ MAPREG | Starting map register number |
| ADP\$_MRNREGARY, ADP\$_MRFREGARY, ADP\$_MRACTMDRS | Updated |

synchronization

The caller of IOC\$ALOUBAMAP or IOC\$ALOUBAMAPN may be executing at fork IPL or above and must hold the corresponding fork lock in a VMS multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Operating System Routines

IOC\$ALOUBAMAP, IOC\$ALOUBAMAPN

DESCRIPTION

IOC\$ALOUBAMAP and IOC\$ALOUBAMAPN allocate a contiguous set of UNIBUS map registers or a set of the first 496 Q22-bus map registers and record the allocation in the ADP and CRB. These routines differ in the way in which they determine the number of the map registers they allocate:

- IOC\$ALOUBAMAP calculates the number of needed map registers using the values contained in UCB\$W_BCNT and UCB\$W_BOFF. It automatically allocates one extra map register. When it is later called by the driver, IOC\$LOADUBAMAP marks this register invalid to prevent a transfer overrun.
- IOC\$ALOUBAMAPN uses the value in R3 as the number of required registers.

If an odd number of map registers is required, both routines round this value up to an even multiple.

If map registers have been permanently allocated to the controller, IOC\$ALOUBAMAP or IOC\$ALOUBAMAPN returns successfully to its caller without allocating the requested map registers. Otherwise, it searches the map register descriptor arrays for the required number of map registers. If there are not enough contiguous map registers available, the routine returns an error status of zero to its caller.

IOC\$APPLYECC

Applies an ECC correction to data transferred from a disk device into memory.

module

IOSUBRAMS

input

| Location | Contents |
|---------------|---|
| R0 | Number of bytes of data that have been transferred, not including the block to be corrected; this must be a multiple of 512 bytes |
| R5 | Address of UCB |
| UCB\$W_BCNT | Length of transfer in bytes |
| UCB\$W_EC1 | Starting bit number of the error burst |
| UCB\$W_EC2 | Exclusive OR correction pattern |
| UCB\$L_SVPN | Address of system PTE for a page that is available for use by driver |
| UCB\$L_SVAPTE | System virtual address of PTE that maps the transfer |

output

| Location | Contents |
|---------------|--|
| R0, R1, R2 | Destroyed |
| UCB\$W_DEVSTS | UCB\$V_ECC set to indicate that an ECC correction was made |

synchronization

IOC\$APPLYECC executes at the caller's IPL, obtains no spin locks, and returns control to its caller at its caller's IPL.

DESCRIPTION

IOC\$APPLYECC corrects data transferred from a disk device to memory by performing an exclusive-OR operation on the data and applying a correction pattern from the UCB. IOC\$APPLYECC also sets a UCB bit (UCB\$V_ECC in UCB\$W_DEVSTS) to indicate that it has made an ECC correction.

Note that, to use this routine, the driver must define the local UCB disk extension, as described in Section A.14.

Operating System Routines

IOC\$CANCELIO

IOC\$CANCELIO

Conditionally marks a UCB so that its current I/O request will be canceled.

module

IOSUBNPAG

input

| Location | Contents |
|-------------|---|
| R2 | Channel index number |
| R3 | Address of IRP |
| R4 | Address of current PCB |
| R5 | Address of UCB |
| IRP\$L_PID | Process identification of the process that queued the I/O request |
| IRP\$W_CHAN | I/O request channel index number |
| PCB\$L_PID | Process identification of the process that requested cancellation |
| UCB\$L_STS | UCB\$V_BSY set if device is busy, clear if device is idle |

output

| Location | Contents |
|------------|---|
| UCB\$L_STS | UCB\$V_CANCEL set if the I/O request should be canceled |

synchronization

IOC\$CANCELIO executes at its caller's IPL, obtains no spin locks, and returns control to its caller at the caller's IPL. It is usually called by EXE\$CANCEL (if specified in the DDT as the driver's cancel-I/O routine) at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment.

DESCRIPTION

IOC\$CANCELIO cancels I/O to a device in the following device-independent manner:

- 1 It confirms that the device is busy by examining the device-busy bit in the UCB status longword (UCB\$V_BSY in UCB\$L_STS).
- 2 It confirms that the IRP in progress on the device originates from the current process (that is, the contents of IRP\$L_PID and PCB\$L_PID are identical).
- 3 It confirms that the specified channel-index number is the same as the value stored in the IRP's channel-index field (IRP\$W_CHAN).
- 4 It sets the cancel-I/O bit in the UCB status longword (UCB\$V_CANCEL in UCB\$L_STS).

IOC\$DIAGBUFILL

Fills a diagnostic buffer if the original \$QIO request specified such a buffer.

module

IOSUBNPAG

input

| Location | Contents |
|-----------------|--|
| R4 | Address of device's CSR |
| R5 | Address of UCB |
| UCB\$L_IRP | Address of current IRP |
| IRP\$W_STS | IRP\$V_DIAGBUF set if a diagnostic buffer exists |
| IRP\$L_DIAGBUF | Address of diagnostic buffer, if one is present |
| UCB\$B_ERTCNT | Final error retry count |
| UCB\$L_DDB | Address of DDB |
| DDT\$L_DDT | Address of DDT |
| DDT\$L_REGDUMP | Address of driver's register dumping routine |
| EXE\$GQ_SYSTIME | Current system time (time at I/O request completion) |

output

| Location | Contents |
|----------|-------------------------|
| R0, R1 | Destroyed |
| R2 | Address of DDT |
| R3 | Address of IRP |
| R4 | Address of device's CSR |
| R5 | Address of UCB |

synchronization

The caller of IOC\$DIAGBUFILL may be executing at or above fork IPL and must hold the corresponding fork lock in a VMS multiprocessing environment. IOC\$DIAGBUFILL returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

A device driver fork process calls IOC\$DIAGBUFILL at the end of I/O processing but before releasing the I/O channel. IOC\$DIAGBUFILL stores the I/O completion time and the final error retry count in the diagnostic buffer. (IOC\$INITIATE has already placed the I/O initiation time in the first quadword of the buffer.) IOC\$DIAGBUFILL then calls the driver's register dumping routine, which fills the remainder of the buffer, and returns to its caller.

Operating System Routines

IOC\$INITIATE

IOC\$INITIATE

Initiates the processing of the next I/O request for a device unit.

module

IOSUBNPAG

input

| Location | Contents |
|------------------|--|
| R3 | Address of IRP |
| R5 | Address of UCB |
| CPU\$L_PHY_CPUID | CPU ID of local processor |
| IRP\$L_SVAPTE | Address of system buffer (buffered I/O) or system virtual address of the PTE that maps process buffer (direct I/O) |
| IRP\$W_BOFF | Byte offset of start of buffer |
| IRP\$L_BCNT | Size in bytes of transfer |
| IRP\$W_STS | IRP\$V_DIAGBUF set if a diagnostic buffer exists |
| IRP\$L_DIAGBUF | Address of diagnostic buffer, if one is present |
| EXE\$GO_SYSTIME | Current system time (when I/O processing began) |
| UCB\$L_DDB | Address of DDB |
| UCB\$L_DDT | Address of DDT |
| UCB\$L_AFFINITY | Device's affinity mask |
| DDT\$L_START | Address of driver start-I/O routine |

output

| Location | Contents |
|-------------------|--|
| RO, R1 | Destroyed |
| UCB\$L_IRP | Address of IRP |
| UCB\$L_SVAPTE | IRP\$L_SVAPTE |
| UCB\$W_BOFF | IRP\$W_BOFF |
| UCB\$W_BCNT | IRP\$L_BCNT (low-order word) |
| UCB\$L_STS | UCB\$V_CANCEL and UCB\$V_TIMEOUT cleared |
| Diagnostic buffer | Current system time (first quadword) |

synchronization

IOC\$INITIATE is called at fork IPL with the corresponding fork lock held in a VMS multiprocessing system. Within this context, it transfers control to the driver's start-I/O routine.

Operating System Routines

IOC\$INITIATE

DESCRIPTION

IOC\$INITIATE creates the context in which a driver fork process services an I/O request. IOC\$INITIATE creates this context and activates the driver's start-I/O routine in the following steps:

- Checks the CPU ID of the local processor against the device's affinity mask to determine whether the local processor can initiate the I/O operation on the device. If it cannot, IOC\$INITIATE takes steps to initiate the I/O function on another processor in a VMS multiprocessing system. It then returns to its caller.
- Stores the address of the current IRP in UCB\$L_IRP.
- Copies the transfer parameters contained in the IRP into the UCB:
 - Copies the address of the system buffer (buffered I/O) or the system virtual address of the PTE that maps process buffer (direct I/O) from IRP\$L_SVAPTE to UCB\$L_SVAPTE
 - Copies the byte offset within the page from IRP\$W_BOFF to UCB\$W_BOFF
 - Copies the low-order word of the byte count from IRP\$L_BCNT to UCB\$W_BCNT
- Clears the cancel-I/O and timeout bits in the UCB status longword (UCB\$V_CANCEL and UCB\$V_TIMEOUT in UCB\$L_STS).
- If the I/O request specifies a diagnostic buffer, as indicated by IRP\$V_DIAGBUF in IRP\$W_STS, stores the system time in the first quadword of the buffer to which IRP\$L_DIAGBUF points (the \$QIO system service having already allocated the buffer).
- Transfers control to the driver's start-I/O routine.

Operating System Routines

IOC\$IOPPOST

IOC\$IOPPOST

Performs device-independent I/O postprocessing and delivers the results of an I/O request to a process.

module

IOCIOPPOST

input

| Location | Contents |
|----------------|--|
| CPU\$_PSFL | Head of the CPU-specific I/O postprocessing queue |
| IRP\$_PID | Process identification of the process that initiated the I/O request |
| IRP\$_UCB | Address of UCB |
| IRP\$_STS | IRP\$_BUFIO set if buffered-I/O request, clear if direct-I/O request; IRP\$_PHYSIO set if physical-I/O function; IRP\$_EXTEND set if an IRPE is linked to this IRP; IRP\$_KEY set if IRP\$_KEYDESC contains the address of an encryption key buffer; IRP\$_FUNC set if read function, clear if write function; IRP\$_DIAGBUF set if diagnostic buffer exists; IRP\$_MBXIO set if mailbox read function |
| IRP\$_DIAGBUF | Address of diagnostic buffer, if one is present |
| IRP\$_SVAPE | Address of system buffer (buffered I/O) or system virtual address of the PTE that maps process buffer (direct I/O) |
| IRP\$_BOFF | Byte offset of start of buffer |
| IRP\$_BCNT | Size in bytes of transfer |
| IRP\$_OBCNT | Original byte count for virtual I/O transfer |
| IRP\$_IOST1 | First I/O status longword |
| IRP\$_CHAN | I/O request channel index number |
| IRP\$_IOSB | Address of I/O status block, if specified |
| IRP\$_RMOD | Access mode of I/O request; ACB\$_QUOTA set if request specified AST |
| IRP\$_EFN | Event flag number |
| UCB\$_W_QLen | Length of pending-I/O queue |
| UCB\$_DEVCHAR | DEV\$_FOD set if file-oriented device |
| PCB\$_W_DIOCNT | Process's direct-I/O count |
| PCB\$_W_BIOCNT | Process's buffered-I/O count |
| JIB\$_BYTCNT | Job byte count quota |
| CCB\$_W_IOC | Number of outstanding I/O requests on channel |
| CCB\$_DIRP | Address of IRP for requested deaccess |

Operating System Routines

IOC\$IOPOST

output

| Location | Contents |
|--------------|--|
| CPU\$_PSFL | Updated |
| UCB\$_QLEN | Decrementd |
| PCB\$_DIOCNT | Incremented for a direct-I/O request |
| PCB\$_BIOCNT | Incremented for a buffered I/O request |
| JIB\$_BYTCNT | Updated for buffered I/O request |
| CCB\$_IOC | Decrementd |
| CCB\$_DIRP | Cleared if channel is idle |

synchronization

IOC\$IOPOST executes in response to an interrupt granted at IPL\$_IOPOST. It performs some of its functions in a special kernel-mode AST that executes within process context at IPL\$_ASTDEL. It obtains and releases the various spin locks required to deallocate nonpaged pool and adjust process quotas.

DESCRIPTION

This interrupt service routine processes IRPs in an I/O postprocessing queue, gaining control when the processor grants a software interrupt at IPL\$_IOPOST. When a processor's I/O postprocessing queue is empty, IOC\$IOPOST dismisses the interrupt with an REI instruction.

IOC\$IOPOST performs several tasks to complete either a direct- or buffered-I/O request:

- For a *buffered-I/O* read request, it copies data from the system buffer to the process buffer. If it cannot write to the process buffer, it returns SSS\$_ACCVIO status. For read and write requests, it releases the system buffer to nonpaged pool.
- For a *direct-I/O* request, it unlocks those process buffer pages that were locked for the I/O transfer. (If an IRPE exists, the unlocked pages include any defined in the IRPE area descriptors.)

IOC\$IOPOST performs the following tasks for *both* direct and buffered I/O requests:

- Decrements the device's pending-I/O queue length
- Adjusts direct-I/O or buffered-I/O quota use
- Sets an event flag if one was specified in the \$QIO system service call
- Copies I/O completion status from the IRP to the process's I/O status block (if one was specified in the \$QIO system service call).
- Queues a user mode AST (if specified) to the process
- Copies the diagnostic buffer (if specified) from system to process space and releases the system buffer
- Deallocates the IRP and any IRPEs

Note that many of these operations are performed within process context by the special kernel-mode AST IOC\$IOPOST queues to the process.

Operating System Routines

IOC\$LOADALTMAP

IOC\$LOADALTMAP

Loads a set of Q22-bus alternate map registers.

module [SYSLOA]MAPSUBxxx

macro LOADALT

input

| Location | Contents |
|-------------------------------|--|
| R5 | Address of UCB |
| UCB\$W_BCNT | Number of bytes in transfer |
| UCB\$W_BOFF | Byte offset in first page of transfer |
| UCB\$L_SVAPTE | System virtual address of PTE for first page of transfer |
| UCB\$L_CRB | Address of CRB |
| CRB\$L_INTD+ VEC\$W_NUMALT | Number of alternate map registers allocated |
| CRB\$L_INTD+ VEC\$W_MAPALT | Number of first alternate map register allocated |
| CRB\$L_INTD+ VEC\$L_ADP | Address of ADP |
| ADP\$L_MR2ADDR | Address of the first Q22-bus alternate map register |

output

| Location | Contents |
|----------|--|
| R0 | SS\$_NORMAL, SS\$_INSFMAPREG, or SS\$_SSFAIL |
| R1, R2 | Destroyed |

synchronization

A driver fork process calls IOC\$LOADALTMAP at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment. IOC\$LOADALTMAP returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

A driver fork process calls IOC\$LOADALTMAP to load a previously-allocated set of alternate map registers with page-frame numbers (PFNs). This enables a device DMA transfer to or from the buffer indicated by the contents of UCB\$L_SVAPTE, UCB\$W_BCNT, and UCB\$W_BOFF.

Operating System Routines

IOC\$LOADALTMAP

IOC\$LOADALTMAP confirms that sufficient alternate map registers have been previously allocated. If not, it issues a UBMAPEXCED bugcheck. Otherwise, it loads the appropriate PFN into each map register and sets the map register valid bit. It clears the last map register. This last invalid register prevents a transfer overrun.

If the VAX system does not support alternate map registers, the routine exits with SS\$_SSFAIL status.

Operating System Routines

IOC\$LOADMBAMAP

IOC\$LOADMBAMAP

Loads MASSBUS map registers.

module LOADMREG

macro LOADMBA

input

| Location | Contents |
|--------------|--|
| R4 | Address of MBA configuration register (MBA\$_CSR) |
| R5 | Address of UCB |
| UCB\$_BCNT | Number of bytes in transfer |
| UCB\$_BOFF | Byte offset in first page of transfer |
| UCB\$_SVAPTE | System virtual address of PTE for first page of transfer |
| MBA\$_MAP | Address of first MASSBUS map register |

output

| Location | Contents |
|------------|-----------|
| R0, R1, R2 | Destroyed |

synchronization A driver fork process calls IOC\$LOADMBAMAP at fork IPL. IOC\$LOADMBAMAP returns control to its caller at the caller's IPL.

DESCRIPTION

Driver fork processes for DMA transfers call IOC\$LOADMBAMAP to load MASSBUS adapter map registers with page-frame numbers (PFNs).

IOC\$LOADMBAMAP uses the contents of UCB\$_SVAPTE, UCB\$_BCNT, and UCB\$_BOFF to determine the number of pages involved in the transfer. It then copies the page frame numbers from the page-table entries associated with this buffer into map registers, starting with map register 0. IOC\$LOADMBAMAP also loads the negated transfer size into the MASSBUS adapter's byte count register (MBA\$_BCR) and the byte offset of the transfer into the MASSBUS adapter's virtual address register (MBA\$_VAR). It clears the last map register. This last invalid register prevents a transfer overrun.

The driver must own the MASSBUS adapter, and thus its map registers, before it calls this routine.

Operating System Routines

IOC\$LOADUBAMAP, IOC\$LOADUBAMAPA

IOC\$LOADUBAMAP, IOC\$LOADUBAMAPA

Load a set of UNIBUS map registers or a set of the first 496 Q22 bus map registers.

module LOADMREG

macro LOADUBA

input

| Location | Contents |
|---------------------------------|---|
| R5 | Address of UCB |
| UCB\$W_BCNT | Number of bytes in transfer |
| UCB\$W_BOFF | Byte offset in first page of transfer |
| UCB\$L_SVAPTE | System virtual address of PTE for first page of transfer |
| UCB\$L_CRB | Address of CRB |
| CRB\$L_INTD+ VEC\$B_NUMREG | Number of map registers allocated |
| CRB\$L_INTD+ VEC\$W_MAPREG | Number of first map register allocated |
| CRB\$L_INTD+ VEC\$B_DATAPATH | Data path specifier; VEC\$V_LWAE set if longword buffering is used, clear if quadword buffering is used |
| CRB\$L_INTD+ VEC\$L_ADP | Address of ADP |
| UBA\$L_MAP | Address of first UNIBUS or Q22 bus map register |
| UCB\$L_SVAPTE | System virtual address of PTE for the first page of the transfer |

output

| Location | Contents |
|------------|-----------|
| R0, R1, R2 | Destroyed |

synchronization

A driver fork process calls IOC\$LOADUBAMAP or IOC\$LOADUBAMAPA at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

A driver fork process calls IOC\$LOADUBAMAP or IOC\$LOADUBAMAPA to load a previously-allocated set of map registers with page-frame numbers (PFNs). This enables a device DMA transfer to or from the buffer indicated by the contents of UCB\$L_SVAPTE, UCB\$W_BCNT, and UCB\$W_BOFF.

Operating System Routines

IOC\$LOADUBAMAP, IOC\$LOADUBAMAPA

Either IOC\$LOADUBAMAP or IOC\$LOADUBAMAPA confirms that sufficient map registers have been previously allocated. If not, it issues a UBMAPEXCED bugcheck. Otherwise, it loads into each map register the appropriate PFN and data-path number. It sets the map register valid bit and, if VEC\$V_LWAE is set in VEC\$B_DATAPATH, the longword-access-enable bit.

IOC\$LOADUBAMAP checks the low bit of UCB\$W_BOFF to determine whether the transfer is byte-aligned or word-aligned. If the low bit is set, it sets the byte-offset bit in each map register. Drivers for byte-aligned UNIBUS devices that must never set the byte-offset bit call IOC\$LOADUBAMAPA. Drivers for Q22-bus-only devices also call IOC\$LOADUBAMAPA as there is no byte-offset bit in a Q22-bus map register.

Both IOC\$LOADUBAMAP and IOC\$LOADUBAMAPA clear the last map register. This last invalid register prevents a transfer overrun.

Operating System Routines

IOC\$MOVFRUSER, IOC\$MOVFRUSER2

IOC\$MOVFRUSER, IOC\$MOVFRUSER2

Move a string from a user buffer to a system buffer.

module

BUFFERCTL

input

| Location | Contents |
|---------------|---|
| R0 | Address of byte to be moved (IOC\$MOVFRUSER2 only) |
| R1 | Address of driver's buffer |
| R2 | Number of bytes to move |
| R5 | Address of UCB |
| DPT\$B_FLAGS | Bit DPT\$V_SVP set (causing a system page-table entry (SPTe) to be allocated to the driver) |
| UCB\$L_SVAPTE | System virtual address of PTE that maps the first page of the buffer |
| UCB\$L_SVPN | System virtual page number of SPTe allocated to driver |
| UCB\$W_BOFF | Byte offset to start of transfer in page |

output

None.

synchronization

The caller of IOC\$MOVFRUSER or IOC\$MOVFRUSER2 may be executing at fork IPL or above and must hold the corresponding fork lock in a VMS multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

IOC\$MOVFRUSER2 is useful for moving blocks of data in several pieces, each piece beginning within a page rather than on a page boundary. To begin, the driver calls IOC\$MOVFRUSER. For each subsequent piece, the driver calls IOC\$MOVFRUSER2.

If an SPTe has not been allocated to the driver, these routines will cause an access violation when they attempt to refer to the location addressed by the contents of the field UCB\$L_SVAPTE. (See the description of the DPTAB macro in Appendix B for information on how to allocate this SPTe.)

Operating System Routines

IOC\$MOVTOUSER, IOC\$MOVTOUSER2

IOC\$MOVTOUSER, IOC\$MOVTOUSER2

Move a string from a system buffer to a user buffer.

module

BUFFERCTL

input

| Location | Contents |
|--------------|--|
| R0 | Address of byte to be moved (IOC\$MOVTOUSER2 only) |
| R1 | Address of driver's buffer |
| R2 | Number of bytes to move |
| R5 | Address of UCB |
| DPT\$_FLAGS | Bit DPT\$_SVP set (causing a system page-table entry (SPTE) to be allocated to the driver) |
| UCB\$_SVApte | System virtual address of PTE that maps the first page of the buffer |
| UCB\$_SVPn | System virtual page number of SPTE allocated to driver |
| UCB\$_W_BOFF | Byte offset to start of transfer in page |

output

None.

synchronization

The caller of IOC\$MOVTOUSER or IOC\$MOVTOUSER2 may be executing at fork IPL or above and must hold the corresponding fork lock in a VMS multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

IOC\$MOVTOUSER2 is useful for moving blocks of data in several pieces, each piece beginning within a page rather than on a page boundary. To begin, the driver calls IOC\$MOVTOUSER. For each subsequent piece, the driver calls IOC\$MOVTOUSER2.

If an SPTE has not been allocated to the driver, these routines will cause an access violation when they attempt to refer to the location addressed by the contents of the field UCB\$_SVApte. (See the description of the DPTAB macro in Appendix B for information on how to allocate this SPTE.)

IOC\$PURGDATAP

Purges the buffered data path and logs memory errors that may have occurred during an I/O transfer.

module [SYSLOA]LIOSUBxxx

macro PURDPR

input

| Location | Contents |
|----------|----------------|
| R5 | Address of UCB |

output

| Location | Contents |
|----------|---|
| R0 | Bit 0 set if success, clear if failure |
| R1 | Contents of data path after purge |
| R2 | Address of start of the I/O bus map registers |
| R3 | Address of CRB |

synchronization

The caller of IOC\$PURGDATAP may be executing at fork IPL or above and must hold the corresponding fork lock in a VMS multiprocessing environment. It returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

All device drivers that support DMA transfers, including those on VAX systems that have no buffered data paths (such as the MicroVAX 3600 series, MicroVAX II, and MicroVAX I), call IOC\$PURGDATAP after a data transfer.

IOC\$PURGDATAP performs the following tasks:

- Obtains the start of adapter register space using the following chain of pointers:
UCB\$_CRB → CRB\$_INTD+VEC\$_ADP → ADP\$_CSR
- Extracts the caller's data path number (buffered or direct) from the CRB.
- Purges the data path if it is a buffered data path. Note that a purge of a direct data path (data path 0) is legal and always results in success status.
- Stores the contents of the data path register in R1. The driver's register dumping routine writes this value to the error message buffer.

Operating System Routines

IOC\$PURGDATAP

- Clears any purge errors in the data path register.
- Places the appropriate return status in R0.
- Determines the base of UNIBUS or Q22-bus map registers and writes the value into R2. The driver's register dumping routine writes this value to the error message buffer.
- In some machine implementations, checks for memory errors that might have occurred during the DMA operation and, if an error is detected, logs it.

IOC\$RELALTMAP

Releases a set of Q22-bus alternate map registers.

module [SYSLOA]MAPSUBxxx

macro RELALT

input

| Location | Contents |
|---|---|
| R5 | Address of UCB |
| UCB\$_CRB | Address of CRB |
| CRB\$_INTD+ VEC\$_ADP | Address of ADP |
| CRB\$_INTD+ VEC\$_MAPALT | Starting alternate map register number; VEC\$_ALTLOCK set indicates that alternate map registers have been permanently allocated to this controller |
| CRB\$_INTD+ VEC\$_NUMALT | Number of allocated alternate map registers |
| ADP\$_MR2QFL | Head of queue of UCBs waiting for alternate map registers |
| ADP\$_MR2NREGAR, ADP\$_MR2FREGAR, ADP\$_MR2ACTMDR | Alternate map register descriptor arrays |

output

| Location | Contents |
|---|----------------------------|
| R0 | SS\$_NORMAL or SS\$_SSFAIL |
| R1, R2 | Destroyed |
| ADP\$_MR2NREGAR, ADP\$_MR2FREGAR, ADP\$_MR2ACTMDR | Updated |

synchronization

A driver fork process calls IOC\$RELALTMAP at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment.

DESCRIPTION

A driver fork process calls IOC\$RELALTMAP to release a previously-allocated set of Q22-bus alternate map registers (registers 496 to 8191) and update the alternate map register descriptor arrays in the ADP. IOC\$RELALTMAPREG assumes that its caller is the current owner of the controller data channel.

Operating System Routines

IOC\$RELALTMAP

IOC\$RELALTMAP obtains the location and number of the allocated map registers from CRB\$L_INTD+VEC\$W_MAPALT and CRB\$L_INTD+VEC\$W_NUMALT, respectively. If VEC\$V_ALTLOCK is set in CRB\$L_INTD+VEC\$W_MAPALT, the alternate map registers have been permanently allocated to the controller and IOC\$RELALTMAP returns successfully to its caller.

After adjusting the alternate map register descriptor arrays, IOC\$RELALTMAP examines the alternate-map-register wait queue. If the queue is empty, IOC\$RELALTMAP returns successfully to its caller. If the queue contains waiting fork processes, IOC\$RELALTMAP dequeues the first process and calls IOC\$ALOALTMAP to attempt to allocate the set of map registers it requires.

If there are sufficient alternate map registers, IOC\$RELALTMAP restores R3 through R5 to the process and reactivates it. When this fork process returns control to IOC\$RELALTMAP, IOC\$RELALTMAP attempts to allocate map registers to the next waiting fork process. IOC\$RELALTMAP continues to allocate map registers in this manner until the alternate-map-register wait queue is empty or it cannot satisfy the requirements of the process at the head of the queue. In the latter event, IOC\$RELALTMAP reinserts the fork process's UCB in the queue and returns successfully to its caller.

If the VAX system does not support alternate map registers, IOC\$RELALTMAP exits with SS\$_SSFAIL status.

IOC\$RELCHAN

Releases device ownership of all controller data channels.

module

IOSUBNPAG

macro

RELCHAN

input

| Location | Contents |
|----------------------|--|
| R5 | Address of UCB |
| UCB\$_CRB | Address of CRB |
| CRB\$_LINK | Address of secondary CRB |
| CRB\$_MASK | CRB\$_BSY set if the channel is busy |
| CRB\$_INTD+VEC\$_IDB | Address of IDB |
| IDB\$_OWNER | Address of UCB of channel owner |
| CRB\$_WQFL | Head of queue of UCBs waiting for the controller channel |

output

| Location | Contents |
|-------------|---|
| R0, R1, R2 | Destroyed |
| IDB\$_OWNER | Cleared if no driver is waiting for the channel |
| CRB\$_MASK | CRB\$_BSY cleared if no driver is waiting for the channel |

synchronization

A driver fork process calls IOC\$RELCHAN at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment. IOC\$RELCHAN returns control to its caller after resuming execution of other fork processes waiting for a controller channel.

DESCRIPTION

A driver fork process calls IOC\$RELCHAN to release all controller data channel assigned to a device; it calls IOC\$RELSCHAN to release only the secondary data channel.

If the channel wait queue contains waiting fork processes, IOC\$RELCHAN dequeues a process, assigns the channel to that process, restores R3 and R5, moves the address of the CSR (IDB\$_CSR) into R4, and reactivates the suspended fork process.

Operating System Routines

IOC\$RELDATAP

IOC\$RELDATAP

Releases a UNIBUS adapter's buffered data path.

module

IOSUBNPAG

macro

RELDPR

input

| Location | Contents |
|-------------------------------|---|
| R5 | Address of UCB |
| UCB\$_CRB | Address of CRB |
| CRB\$_INTD+ VEC\$_ADP | Address of ADP |
| CRB\$_INTD+ VEC\$_DATAPATH | Data path specifier; VEC\$_PATHLOCK set if the data path has been permanently allocated to the controller |
| ADP\$_DPQFL | Head of queue of UCBs waiting for a UNIBUS adapter buffered data path |
| ADP\$_DPBITMAP | Data path bit map |

output

| Location | Contents |
|-------------------------------|--|
| R0, R1, R2 | Destroyed |
| ADP\$_DPBITMAP | Bit representing data path set if the path is not allocated to another driver fork process |
| CRB\$_INTD+ VEC\$_DATAPATH | Bits 0 through 4 cleared if the path is not permanently allocated |

synchronization

A driver fork process calls IOC\$RELDATAP at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment. IOC\$RELDATAP returns control to its caller after resuming execution of any other fork processes waiting for a buffered data path.

DESCRIPTION

A driver fork process must own a UNIBUS buffered data path when it calls IOC\$RELDATAP.

IOC\$RELDATAP obtains the number of the allocated data path from bits 0 through 4 of the data path specifier. If VEC\$_PATHLOCK is set in the specifier, the data path has been permanently allocated to the controller and IOC\$RELDATAP returns to its caller.

Operating System Routines

IOC\$RELDATAP

If the data path wait queue contains waiting fork processes, IOC\$RELDATAP dequeues the first process, allocates the data path to it, restores R3 through R5, and reactivates it. Otherwise, it marks the path available by setting the corresponding bit in the data path bit map (ADP\$W_DPBITMAP), and returns to its caller.

If the bit map has been corrupted, IOC\$RELDATAP issues an INCONSTATE bugcheck.

Operating System Routines

IOC\$RELMAPREG

IOC\$RELMAPREG

Releases a set of UNIBUS map registers or a set of the first 496 Q22-bus map registers.

module

IOSUBNPAG

macro

RELMPR

input

| Location | Contents |
|---|---|
| R5 | Address of UCB |
| UCB\$_CRB | Address of CRB |
| CRB\$_INTD+ VEC\$_ADP | Address of ADP |
| CRB\$_INTD+ VEC\$_MAPREG | Starting map register number; VEC\$_MAPLOCK set indicates that map registers have been permanently allocated to this controller |
| CRB\$_INTD+ VEC\$_NUMREG | Number of allocated map registers |
| ADP\$_MRQFL | Head of queue of UCBs waiting for map registers |
| ADP\$_MRNREGARY, ADP\$_MRFREGARY, ADP\$_MRACTMDRS | Map register descriptor arrays |

output

| Location | Contents |
|---|----------------------------|
| R0 | SS\$_NORMAL or SS\$_SSFAIL |
| R1, R2 | Destroyed |
| ADP\$_MRNREGARY, ADP\$_MRFREGARY, ADP\$_MRACTMDRS | Updated |

synchronization

A driver fork process calls IOC\$RELMAPREG at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment.

DESCRIPTION

A driver fork process calls IOC\$RELMAPREG to release a previously-allocated set of UNIBUS map registers or a set of the first 496 Q22 bus map registers. IOC\$RELMAPREG updates the alternate map register descriptor arrays in the ADP. IOC\$RELMAPREG assumes that its caller is the current owner of the controller data channel.

Operating System Routines

IOC\$RELMAPREG

IOC\$RELMAPREG obtains the location and number of the allocated map registers from CRB\$L_INTD+VEC\$W_MAPREG and CRB\$L_INTD+VEC\$B_NUMREG, respectively. If VEC\$V_MAPLOCK is set in CRB\$L_INTD+VEC\$W_MAPREG, the map registers have been permanently allocated to the controller and IOC\$RELMAPREG returns successfully to its caller.

After adjusting the map register descriptor arrays, IOC\$RELMAPREG examines the standard-map-register wait queue. If the queue is empty, IOC\$RELMAPREG returns successfully to its caller. If the queue contains waiting fork processes, IOC\$RELMAPREG dequeues the first process and calls IOC\$ALOUBAMAP to attempt to allocate the set of map registers it requires.

If there are sufficient map registers, IOC\$RELMAPREG restores R3 through R5 to the process and reactivates it. When this fork process returns control to IOC\$RELMAPREG, IOC\$RELMAPREG attempts to allocate map registers to the next waiting fork process. IOC\$RELMAPREG continues to allocate map registers in this manner until the standard-map-register wait queue is empty or it cannot satisfy the requirements of the process at the head of the queue. In the latter event, IOC\$RELMAPREG reinserts the fork process's UCB in the queue and returns successfully to its caller.

Operating System Routines

IOC\$RELSCHAN

IOC\$RELSCHAN

Releases device ownership of only the secondary controller's data channel.

module

IOSUBNPAG

macro

RELSCHAN

input

| Location | Contents |
|----------------------|--|
| R5 | Address of UCB |
| UCB\$_CRB | Address of CRB |
| CRB\$_LINK | Address of secondary CRB |
| CRB\$_MASK | CRB\$_BSY set if the channel is busy |
| CRB\$_INTD+VEC\$_IDB | Address of IDB |
| IDB\$_OWNER | Address of UCB of channel owner |
| CRB\$_WQFL | Head of queue of UCBs waiting for the controller channel |

output

| Location | Contents |
|-------------|---|
| R0, R1, R2 | Destroyed |
| IDB\$_OWNER | Cleared if no driver is waiting for the channel |
| CRB\$_MASK | CRB\$_BSY cleared if no driver is waiting for the channel |

synchronization

A driver fork process calls IOC\$RELSCHAN at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment. IOC\$RELSCHAN returns control to its caller after resuming execution of other fork processes waiting for the secondary controller's channel.

DESCRIPTION

IOC\$RELSCHAN releases a secondary controller's data channel (for instance, the MASSBUS adapter's controller data channel). The caller retains ownership of the primary controller's data channel. A driver fork process calls IOC\$RELCHAN to release all controller data channels assigned to a device.

If the secondary channel's wait queue contains waiting fork processes, IOC\$RELSCHAN dequeues a process, assigns the channel to that process, restores R3 through R5, and reactivates the suspended process.

IOC\$REQALTMAP

Allocates sufficient Q22-bus alternate map registers to accommodate a DMA transfer and, if unavailable, places the requesting fork process in an alternate-map-register wait queue.

module SYSLOA[MAPSUB]xxx

macro REQALT

input

| Location | Contents |
|--|--|
| R5 | Address of UCB |
| 00(SP) | Return PC of caller |
| 04(SP) | Return PC of caller's caller |
| UCB\$W_BCNT | Transfer byte count |
| UCB\$W_BOFF | Byte offset in page |
| UCB\$L_CRB | Address of CRB |
| CRB\$L_INTD+ VEC\$L_ADP | Address of ADP |
| CRB\$L_INTD+ VEC\$W_MAPALT | VEC\$V_ALTLOCK set indicates that alternate map registers have been permanently allocated to this controller |
| ADP\$W_MR2NREGAR, ADP\$W_MR2FREGAR, ADP\$L_MR2ACTMDR | Alternate map register descriptor arrays |
| ADP\$L_MR2QBL | Tail of queue of UCBs waiting for alternate map registers |

Operating System Routines

IOC\$REQALTMAP

output

| Location | Contents |
|---|---|
| R0 | SS\$_NORMAL or SS\$_SSFAIL |
| R1 | Destroyed |
| R2 | Address of ADP |
| CRB\$_INTD+ VEC\$_NUMALT | Number of alternate map registers allocated |
| CRB\$_INTD+ VEC\$_MAPALT | Starting alternate map register number |
| ADP\$_MR2NREGAR, ADP\$_MR2FREGAR, ADP\$_MR2ACTMDR | Updated |
| ADP\$_MR2QBL | Updated |
| UCB\$_FR3 | R3 of caller |
| UCB\$_FR4 | R4 of caller |
| UCB\$_FPC | 00(SP) |

synchronization

A driver fork process calls IOC\$REQALTMAP at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment.

DESCRIPTION

A driver fork process calls IOC\$REQALTMAP to allocate a contiguous set of Q22 bus alternate map registers (registers 496 to 8191) to service the DMA transfer described by UCB\$_BCNT and UCB\$_BOFF. IOC\$REQALTMAP calls IOC\$ALOALTMAP.

If alternate map registers have been permanently allocated to the controller, IOC\$REQALTMAP returns successfully to its caller without allocating map registers. Otherwise, it searches the alternate map register descriptor arrays for the required number of map registers.

IOC\$ALOALTMAP determines the required number of alternate map registers from the contents of UCB\$_BOFF and UCB\$_BCNT. It allocates one extra map register; this register is marked invalid when the driver fork process subsequently calls IOC\$LOADALTMAP, thus preventing a transfer overrun. If an odd number of map registers is required, IOC\$ALOALTMAP rounds this value up to an even multiple.

If sufficient alternate map registers are available, IOC\$REQALTMAP assigns them to its caller, records the allocation in the ADP and CRB, and returns successfully to its caller.

If IOC\$REQALTMAP cannot allocate a sufficient number of contiguous map registers, it saves process context by placing the contents of R3, R4, and the PC into the UCB fork block and the UCB into the alternate-map-register wait queue (ADP\$_MR2QBL). It then returns to its caller's caller.

If the VAX system does not support alternate map registers, IOC\$REQALTMAP exits with SS\$_SSFAIL status.

IOC\$REQCOM

Completes an I/O operation on a device unit, requests I/O postprocessing of the current request, and starts the next I/O request waiting for the device.

module

IOSUBNPAG

macro

REQCOM

input

| Location | Contents |
|-----------------|---|
| R0 | First longword of I/O status. |
| R1 | Second longword of I/O status. |
| R5 | Address of UCB. |
| UCB\$L_STS | UCB\$V_ERLOGIP set if error logging is in progress. |
| UCB\$B_ERTCNT | Final error count. |
| UCB\$B_ERTMAX | Maximum error retry count. |
| UCB\$L_EMB | Address of error message buffer. |
| UCB\$L_IRP | Address of IRP. |
| UCB\$B_DEVCLASS | DC\$_DISK and DC\$_TAPE devices are subject to mount verification checks. |
| UCB\$L_IOQFL | Device unit's pending-I/O queue. |
| CPU\$L_PSBL | Tail of local processor's I/O postprocessing queue. |

output

| Location | Contents |
|--------------------|---|
| R0 through R3 | Destroyed. Other registers (used by the driver's start-I/O routine) are destroyed if IOC\$INITIATE is called. |
| IRP\$L_IOST1 | First longword of I/O status. |
| IRP\$L_IOST2 | Second longword of I/O status. |
| UCB\$L_OPCNT | Incremented. |
| UCB\$L_IOQFL | Updated. |
| EMB\$W_DV_STS | UCB\$W_STS. |
| EMB\$B_DV_ERTCNT | UCB\$B_ERTCNT. |
| EMB\$B_DV_ERTCNT+1 | UCB\$B_ERTMAX. |
| EMB\$Q_DV_IOSB | Quadword of I/O status. |
| UCB\$L_STS | UCB\$V_BSY and UCB\$V_ERLOGIP cleared. |
| CPU\$L_PSBL | Updated. |

Operating System Routines

IOC\$REQCOM

synchronization

A driver fork process calls IOC\$REQCOM at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment. IOC\$REQCOM transfers control to IOC\$RELCHAN. If the fork process calls IOC\$REQCOM by means of the REQCOM macro (or a JMP instruction), IOC\$RELCHAN returns control to the caller of the driver fork process (for instance, the fork dispatcher).

DESCRIPTION

A driver fork process calls this routine after a device I/O operation and all device-dependent processing of an I/O request is complete.

IOC\$REQCOM performs the following tasks:

- If error logging is in progress for the device (as indicated by UCB\$V_ERLOGIP in UCB\$L_STS), writes into the error message buffer the status of the device unit, the error retry count for the transfer, the maximum error retry count for the driver, and the final status of the I/O operation. It then releases the error message buffer by calling ERL\$RELEASEMB.
- Increments the device unit's operations count (UCB\$L_OPCNT).
- If UCB\$B_DEVCLASS specifies a disk device (DC\$_DISK) or tape device (DC\$_TAPE) and error status is reported, performs a set of checks to determine if mount verification is necessary. Tape end-of-file errors (SS\$_ENDOFFILE) are exempt from these checks. For a tape device with success status, checks to determine if CRC must be generated.
- Writes final I/O status (R0 and R1) into IRP\$L_IOST1 and IRP\$L_IOST2.
- Inserts the IRP into the local processor's I/O postprocessing queue headed by CPU\$L_PSBL.
- Requests a software interrupt from the local processor at IPL\$_IOPOST.
- Attempts to remove an IRP from the device's pending-I/O queue (at UCB\$L_IOQFL). If successful, it transfers control to IOC\$INITIATE to begin driver processing of this I/O request. If the queue is empty, it clears the unit busy bit (UCB\$V_BSY in UCB\$L_STS) to indicate that the device is idle.
- Exits by transferring control to IOC\$RELCHAN.

IOC\$REQDATAP, IOC\$REQDATAPNW

Request a UNIBUS adapter buffered data path and, optionally, if no path is available, place process in data-path wait queue.

module

IOSUBNPAG

macro

REQDPR

input

| Location | Contents |
|-------------------------------|---|
| R5 | Address of UCB |
| 00(SP) | Return PC of caller |
| 04(SP) | Return PC of caller's caller |
| UCB\$_CRB | Address of CRB |
| UCB\$_CRB | Address of CRB |
| CRB\$_INTD+ VEC\$_ADP | Address of ADP |
| CRB\$_INTD+ VEC\$_DATAPATH | Data path specifier; VEC\$_PATHLOCK set if the data path is permanently allocated to the controller |
| ADP\$_DPBITMAP | Data path bit map |

output

| Location | Contents |
|-------------------------------|--|
| R0 | SS\$_NORMAL or bit 0 set (indicating error status) |
| CRB\$_INTD+ VEC\$_DATAPATH | Data path specifier |
| ADP\$_DPBITMAP | Bit corresponding to allocated data path cleared |

synchronization

A driver fork process calls IOC\$REQDATAP or IOC\$REQDATAPNW at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment.

DESCRIPTION

A driver fork process calls IOC\$REQDATAP or IOC\$REQDATAPNW to request a UNIBUS adapter buffered data path for a DMA transfer.

If a buffered data path is already permanently allocated to the controller, IOC\$REQDATAP or IOC\$REQDATAPNW returns successfully to its caller without allocating a data path. Otherwise, it searches the data path bit map for the first available data path.

Operating System Routines

IOC\$REQDATAP, IOC\$REQDATAPNW

If IOC\$REQDATAP or IOC\$REQDATAPNW locates a free data path, it writes the data path number into CRB\$L_INTD+VEC\$B_DATAPATH, updates the data path bit map (ADP\$W_DPBITMAP), and returns successfully to its caller. If the bit map has been corrupted, the routine issues an INCONSTATE bugcheck.

If IOC\$REQDATAP cannot allocate a data path, it saves process context by placing the contents of R3, R4 and the PC into the UCB fork block and the UCB into the data-path wait queue (ADP\$L_DPQBL). It then returns to its caller's caller. By contrast, if IOC\$REQDATAPNW cannot allocate a data path, it returns immediately to its caller with the low bit in R0 clear, indicating an error.

When called from a driver executing in a VAX system that does not provide buffered data paths, IOC\$REQDATAP and IOC\$REQDATAPNW return control after examining the data path bit map in the ADP.

IOC\$REQMAPREG

Allocates sufficient UNIBUS map registers or a sufficient number of the first 496 Q22-bus map registers to accommodate a DMA transfer and, if unavailable, places process in standard-map-register wait queue.

module IOSUBNPAG

macro REQMPR

input

| Location | Contents |
|--|--|
| R5 | Address of UCB |
| 00(SP) | Return PC of caller |
| 04(SP) | Return PC of caller's caller |
| UCB\$W_BCNT | Transfer byte count |
| UCB\$W_BOFF | Byte offset in page |
| UCB\$L_CRB | Address of CRB |
| CRB\$L_INTD+ VEC\$L_ADP | Address of ADP |
| CRB\$L_INTD+ VEC\$W_MAPREG | VEC\$V_MAPLOCK set indicates that map registers have been permanently allocated to this controller |
| ADP\$W_MRNREGARY, ADP\$W_MRFREGARY, ADP\$L_MRACTMDRS | Map register descriptor arrays |
| ADP\$L_MRQBL | Tail of queue of UCBs waiting for map registers |

Operating System Routines

IOC\$REQMAPREG

output

| Location | Contents |
|---|-----------------------------------|
| R0 | SS\$_NORMAL |
| R1 | Destroyed |
| R2 | Address of ADP |
| CRB\$_INTD+ VEC\$_NUMREG | Number of map registers allocated |
| CRB\$_INTD+ VEC\$_MAPREG | Starting map register number |
| ADP\$_MRNREGARY, ADP\$_MRFREGARY, ADP\$_MRACTMDRS | Updated |
| ADP\$_MRQBL | Updated |
| UCB\$_FR3 | R3 of caller |
| UCB\$_FR4 | R4 of caller |
| UCB\$_FPC | 00(SP) |

synchronization

A driver fork process calls IOC\$REQMAPREG at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment.

DESCRIPTION

A driver fork process calls IOC\$REQMAPREG to allocate a contiguous set of UNIBUS map registers or a set of the first 496 Q22-bus map registers to service the DMA transfer described by UCB\$_BCNT and UCB\$_BOFF. IOC\$REQMAPREG calls IOC\$ALOUBAMAP.

If map registers have been permanently allocated to the controller, IOC\$REQMAPREG returns successfully to its caller without allocating map registers. Otherwise, it searches the map register descriptor arrays for the required number of map registers.

IOC\$ALOUBAMAP determines the required number of map registers from the contents of UCB\$_BOFF and UCB\$_BCNT. It allocates one extra map register; this register is marked invalid when the driver fork process subsequently calls IOC\$LOADUBAMAP, thus preventing a transfer overrun. If an odd number of map registers is required, IOC\$ALOUBAMAP rounds this value up to an even multiple.

If sufficient map registers are available, IOC\$REQMAPREG assigns them to its caller, records the allocation in the ADP and CRB, and returns successfully to its caller.

If IOC\$REQMAPREG cannot allocate a sufficient number of contiguous map registers, it saves process context by placing the contents of R3, R4, and the PC into the UCB fork block and R5 into the standard-map-register wait queue (ADP\$_MRQBL). It then returns to its caller's caller.

Operating System Routines

IOC\$REQPCHANH, IOC\$REQPCHANL, IOC\$REQSCHANH, IOC\$REQSCHANL

IOC\$REQPCHANH, IOC\$REQPCHANL, IOC\$REQSCHANH, IOC\$REQSCHANL

Request a controller's primary or secondary data channel and, if unavailable, place process in channel wait queue.

module

IOSUBNPAG

macro

REQPCHAN, REQCHAN

input

| Location | Contents |
|----------------------|---|
| R5 | Address of UCB |
| 00(SP) | Return PC of caller |
| 04(SP) | Return PC of caller's caller |
| UCB\$_CRB | Address of CRB |
| CRB\$_LINK | Address of secondary CRB (IOC\$REQSCHANH and IOC\$REQSCHANL only) |
| CRB\$_MASK | CRB\$_BSY set if the channel is busy |
| CRB\$_INTD+VEC\$_IDB | Address of IDB |
| CRB\$_WQFL | Head of queue of UCBs waiting for the controller channel |
| CRB\$_WQBL | Tail of queue of UCBs waiting for the controller channel |
| IDB\$_CSR | Address of device CSR |

output

| Location | Contents |
|-------------|-----------------------|
| R0, R1, R2 | Destroyed |
| R4 | Address of device CSR |
| IDB\$_OWNER | Address of UCB |
| CRB\$_WQFL | Updated |
| CRB\$_WQBL | Updated |

synchronization

A driver fork process calls IOC\$REQPCHANH, IOC\$REQPCHANL, IOC\$REQSCHANH, or IOC\$REQSCHANL holding the corresponding fork lock in a VMS multiprocessing environment.

Operating System Routines

IOC\$REQPCHANH, IOC\$REQPCHANL, IOC\$REQSCHANH, IOC\$REQSCHANL

DESCRIPTION

A driver fork process calls IOC\$REQPCHANH or IOC\$REQPCHANL to acquire ownership of the primary controller's data channel; it calls IOC\$REQSCHANH or IOC\$REQSCHANL to request the secondary controller's data channel (for instance, the MASSBUS adapter's controller data channel).

Each routine examines CRB\$V_BSY in CRB\$B_MASK. If the selected controller's data channel is idle, the routine grants the channel to the fork process, placing its UCB address in IDB\$L_OWNER and returning successfully with the device's CSR address in R4.

If the data channel is busy, the routine saves process context by placing the contents of R3 and the PC into the UCB fork block. (Note that IOC\$RELCHAN moves the contents of IDB\$L_CSR into R4 before resuming execution of a waiting fork process.) IOC\$REQPCHANH and IOC\$REQSCHANH then insert the UCB at the head of the channel wait queue (CRB\$L_WQFL); IOC\$REQPCHANL and IOC\$REQSCHANL insert the UCB at the tail of the queue (CRB\$L_WQBL). Finally, the routine returns control to its caller's caller.

IOC\$RETURN

Returns to its caller.

module

None.

input

None.

output

None.

synchronization

IOC\$RETURN executes at its caller's IPL and returns control to the caller at that IPL.

DESCRIPTION

IOC\$RETURN is a universal executive routine vector in the fixed portion of the VMS executive. It contains a single RSB instruction. When a driver invokes the DDTAB macro, the macro writes the address of IOC\$RETURN into routine address fields of the DDT that are not supplied in the macro invocation.

Operating System Routines

IOC\$VERIFYCHAN

IOC\$VERIFYCHAN

Verifies an I/O channel number and translates it to a CCB address.

module

IOSUBPAGD

input

| Location | Contents |
|----------------|--|
| R0 | Channel number (in low word) |
| CTL\$GL_CCBASE | Base address of process CCB table |
| CCB\$_AMOD | Access mode (plus 1) of process owning the channel |

output

| Location | Contents |
|----------|--|
| R0 | SS\$_NORMAL, SS\$_IVCHAN, or SS\$_NOPRIV |
| R1 | Address of CCB |
| R2 | Channel index number |
| R3 | Destroyed |

synchronization

Because IOC\$VERIFYCHAN gains access to information stored in user process virtual address space, it should only be called from code originating at IPL\$_ASTDEL or below.

DESCRIPTION

Drivers call IOC\$VERIFYCHAN to validate a user-supplied channel number, construct a channel index, and obtain the address of the CCB to which the channel number points.

If the channel number is invalid or zero, or if the channel is unowned, IOC\$VERIFYCHAN returns SS\$_IVCHAN status to its caller.

If the access mode of the current process is less privileged than that indicated in CCB\$_AMOD, IOC\$VERIFYCHAN returns SS\$_NOPRIV status to its caller with the address of the CCB in R1.

Otherwise, IOC\$VERIFYCHAN returns successfully to its caller with the address of the CCB in R1.

IOC\$WFIKPCH, IOC\$WFIRLCH

Suspend a driver fork thread and fold its context into a fork block in anticipation of a device interrupt or timeout.

module

IOSUBNPAG

macro

WFIKPCH, WFIRLCH

input

| Location | Contents |
|----------------|---|
| R5 | Address of UCB |
| 00(SP) | Address following the JSB to IOC\$WFIKPCH or IOC\$WFIRLCH |
| 04(SP) | Timeout value in seconds |
| 08(SP) | IPL to which to lower before returning to the caller's caller |
| 12(SP) | Return PC of caller's caller |
| EXE\$GL_ABSTIM | Absolute time |

output

| Location | 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 device I/O is being timed |
| UCB\$V_TIMOUT | Cleared to indicate that unit is not timed out |
| UCB\$L_FR3 | R3 |
| UCB\$L_FR4 | R4 |
| UCB\$L_FPC | 00(SP)+2 |

synchronization

When it is called, IOC\$WFIKPCH or IOC\$WFIRLCH assumes that the local processor has obtained the appropriate synchronization with the device database:

- In a *uniprocessing* environment, the processor must be executing at device IPL or above.
- In a *multiprocessing* environment, the processor must own the appropriate device lock, as recorded in the unit control block (UCB\$L_DLCK) of the device unit from which the interrupt is expected. This requirement also presumes that the local processor is executing at the device IPL associated with the lock.

Operating System Routines

IOC\$WFIKPCH, IOC\$WFIRLCH

Before exiting, IOC\$WFIKPCH or IOC\$WFIRLCH achieves the following synchronization:

- In a *uniprocessing* environment, it lowers the local processor's IPL to the IPL saved on the stack.
- In a *multiprocessing* environment, it conditionally releases the device lock, so that if the caller of the driver fork thread (the caller's caller) previously owned the device lock, it will continue to hold it when the routine exits. IOC\$WFIKPCH or IOC\$WFIRLCH also lowers the local processor's IPL to the IPL saved on the stack.

DESCRIPTION

A driver fork process calls IOC\$WFIKPCH to wait for an interrupt while keeping ownership of the controller's data channel; IOC\$WFIRLCH, by contrast, releases the channel.

Either routine performs the following operations:

- Adds 2 to the address on the top of the stack to determine the address of the next instruction in the driver fork thread after the invocation of the WFIKPCH or WFIRLCH macro. (Note that the macro places the relative offset to the timeout handling routine in the word following the JSB to IOC\$WFIKPCH or IOC\$WFIRLCH.) It pops this address into the UCB fork block (UCB\$L_FPC) so that the driver's interrupt service routine can resume execution of the driver fork thread with a JSB instruction.
- Pops R3 and R4 from the top of the stack into the UCB fork block.
- Sets UCB\$V_INT to indicate an expected interrupt from the device unit.
- Sets UCB\$V_TIM to indicate that VMS should check for timeouts from the device unit.
- Determines the timeout due time from the timeout value, now at the top of the stack, and EXE\$GL_ABSTIM, and stores the result in UCB\$L_DUETIM.
- Clears UCB\$V_TIMEOUT to indicate that the unit has not timed out.
- In a multiprocessing environment, issues a DEVICEUNLOCK to conditionally release the device lock associated with the device unit and to lower IPL to the IPL saved on the stack. These actions presume that the DEVICELock macro has been issued prior to the wait-for-interrupt invocation.
- Returns to the caller of the driver fork thread (that is, its caller's caller) whose address is now at the top of the stack.

In the course of processing, IOC\$WFIKPCH or IOC\$WFIRLCH explicitly removes the longwords at 00(SP) through 08(SP) from the stack and implicitly removes the longword at 12(SP) by exiting with an RSB instruction.

Note that IOC\$WFIRLCH exits by transferring control to IOC\$RELCHAN. IOC\$RELCHAN releases the controller data channel and executes the RSB instruction. Because the release of the channel occurs at fork IPL, an interrupt service routine cannot reliably distinguish between operations initiated by IOC\$WFIKPCH and IOC\$WFIRLCH by examining the ownership of the CRB.

LDR\$ALLOC_PT

Allocates the specified number of system page-table entries (SPTes).

module

PTALLOC

input

| Location | Contents |
|-----------------|---------------------------------|
| R2 | Number of SPTes to be allocated |
| LDR\$GL_SPTBASE | Base of system page table |
| LDR\$GL_FREE_PT | Offset to first free SPTe |

output

| Location | Contents |
|----------|---|
| R0 | SS\$_NORMAL, SS\$_INSFSPTS, or SS\$_BADPARAM |
| R1 | Address of first allocated SPTe |
| R2 | Number of allocated system page-table entries |

synchronization

Because LDR\$ALLOC_PT executes at IPL\$_SYNCH and obtains the MMG spin lock in a VMS multiprocessing environment, its caller cannot be executing above IPL\$_SYNCH or hold any higher ranked spin locks. (For instance, a driver fork process executing at IPL\$_SYNCH holding the IOLOCK8 fork lock can call LDR\$ALLOC_PT.) LDR\$ALLOC_PT returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

LDR\$ALLOC_PT allocates the number of system page-table entries (SPTes) specified in R2. LDR\$ALLOC_PT adjusts the pool of free SPTes to reflect the allocation of the SPTes.

A generic VAXBI device driver calls LDR\$ALLOC_PT if it must map the device's node window space. It is the caller's responsibility to fill in each allocated SPTe with a page-frame number (PFN), set its valid bit, and otherwise initialize it.

If R2 contains a zero, LDR\$ALLOC_PT returns SS\$_BADPARAM status in R0 and clears R1. If there are no free SPTes, it returns SS\$_INSFSPTS status to its caller.

Operating System Routines

LDR\$DEALLOC_PT

LDR\$DEALLOC_PT

Deallocates the specified system page-table entries (SPTes).

module

PTALLOC

input

| Location | Contents |
|-----------------|---|
| R1 | Address of first SPTe to be deallocated |
| R2 | Number of SPTes to be deallocated |
| LDR\$GL_SPTBASE | Base of system page table |
| LDR\$GL_FREE_PT | Offset to first free SPTe |

output

| Location | Contents |
|----------|---|
| R0 | SS\$_NORMAL, SS\$_BADPARAM, or LOADER\$_PTE_NOT_EMPTY |
| R1 | Address of first allocated SPTe |
| R2 | Destroyed |

synchronization

Because LDR\$DEALLOC_PT executes at IPL\$_SYNCH and obtains the MMG spin lock in a VMS multiprocessing environment, its caller cannot be executing above IPL\$_SYNCH or hold any higher ranked spin locks. (For instance, a driver fork process executing at IPL\$_SYNCH holding the IOLOCK8 fork lock can call LDR\$DEALLOC_PT.) LDR\$DEALLOC_PT returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

DESCRIPTION

LDR\$DEALLOC_PT deallocates the number of system page-table entries (SPTes) specified in R2, starting at the one indicated by the contents of R1. LDR\$DEALLOC_PT adjusts the pool of free SPTes to reflect the addition of the deallocated SPTes.

If R2 contains a zero, LDR\$DEALLOC_PT returns SS\$_BADPARAM status in R0 and clears R1.

It is the caller's responsibility to ensure that the SPTes to be deallocated are empty.⁵ If they are not, LDR\$DEALLOC_PT returns LOADER\$_PTE_NOT_EMPTY status in R0.

⁵ Modifications to valid SPTes require that these SPTes be flushed from the system's translation buffers. See the description of the INVALIDATE_TB macro in Appendix B.

MMG\$UNLOCK

Unlocks process pages previously locked for a direct-I/O operation.

module

IOLOCK

input

| Location | Contents |
|----------|---|
| R1 | Number of buffer pages to unlock |
| R3 | System virtual address of PTE for the first buffer page |

output

None.

synchronization

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

DESCRIPTION

Drivers rarely use MMG\$UNLOCK. At the completion of a direct-I/O transfer, IOC\$IOPOST automatically unlocks the pages of both the user buffer and any additional buffers specified in region 1 (if defined) and region 2 (if defined) for all the IRPEs linked to the packet undergoing completion processing.

However, driver FDT routines do use MMG\$UNLOCK when an attempt to lock IRPE buffers for a direct-I/O transfer fails. The buffer-locking routines called by such a driver—EXE\$READLOCKR, EXE\$WRITELOCKR, and EXE\$MODIFYLOCKR—all perform coroutine calls back to the driver if an error occurs. When called as a coroutine, the driver must unlock all previously locked regions using MMG\$UNLOCK, and deallocate the IRPE (using EXE\$DEANONPAGED), before returning to the buffer-locking routine.

Operating System Routines

SMP\$ACQNOIPL

SMP\$ACQNOIPL

Acquires a device lock, assuming the local processor is already running at the IPL appropriate for acquisition of the lock.

module

SPINLOCKS

macro

DEVICELock

input

| Location | Contents |
|----------|------------------------|
| R0 | Address of device lock |

output

| Location | Contents |
|----------|------------------------|
| R0 | Address of device lock |

synchronization

Upon entry, the local processor *must* be executing at the synchronization IPL of the device lock, as it is, for instance, when responding to a device interrupt. SMP\$ACQNOIPL exits with the IPL unchanged and the device lock held.

DESCRIPTION

The DEVICELock macro calls SMP\$ACQNOIPL when NOSETIPL is specified as its **condition** argument.

SMP\$ACQNOIPL attempts to acquire the requested device lock, allowing the acquisition to succeed if the local processor already holds the lock or if the lock is unowned.

If the lock is unowned, the routine increments by 1 a counter that records the acquisition level. Each additional (or nested) acquisition of this lock by the owning processor again increments this counter.

If the lock is owned by another processor, the local processor spin waits until the lock is released.

SMP\$ACQUIRE

Acquires a fork lock or spin lock and enforces the appropriate IPL synchronization on the local processor.

module

SPINLOCKS

macro

FORKLOCK, LOCK

input

| Location | Contents |
|----------|------------------------------|
| RO | Fork lock or spin lock index |

output

| Location | Contents |
|----------|------------------------------|
| RO | Fork lock or spin lock index |

synchronization

When calling SMP\$ACQUIRE, the local processor should be executing at an IPL less than or equal to the synchronization IPL of the lock. The routine, if necessary, immediately raises IPL to the synchronization IPL of the lock. Violations of IPL synchronization in a full-checking multiprocessing environment result in a SPLIPLHIGH bugcheck.

In a full-checking multiprocessing environment, if it must spin wait for the requested lock to be released by another processor, SMP\$ACQUIRE temporarily restores the original IPL for the duration of the wait. If the original IPL was less than IPL\$_RESCHED, the spin wait occurs at IPL\$_RESCHED.

SMP\$ACQUIRE exits with IPL at the synchronization IPL of the lock and the fork lock or spin lock held.

DESCRIPTION

The FORKLOCK and LOCK macros call SMP\$ACQUIRE.

In a full-checking multiprocessing environment, SMP\$ACQUIRE, having ensured that IPL has been set to the lock's synchronization IPL, verifies that the local processor does not currently hold any higher-ranked locks. If a higher-ranked lock is held, SMP\$ACQUIRE issues an SPLACQERR bugcheck.

Otherwise SMP\$ACQUIRE attempts to acquire the requested lock, allowing the acquisition to succeed if the local processor already holds the lock or if the lock is unowned.

If the lock is unowned, the routine increments by 1 a counter that records the acquisition level. Each additional (or nested) acquisition of this lock by the owning processor again increments this counter.

If the lock is owned by another processor, the local processor spin waits until the lock is released.

Operating System Routines

SMP\$ACQUIREL

SMP\$ACQUIREL

Acquires a device lock and enforces the appropriate IPL synchronization on the local processor.

module

SPINLOCKS

macro

DEVICELOCK

input

| Location | Contents |
|----------|------------------------|
| R0 | Address of device lock |

output

| Location | Contents |
|----------|------------------------|
| R0 | Address of device lock |

synchronization

When calling SMP\$ACQUIREL, the local processor should be executing at an IPL less than or equal to the synchronization IPL of the device lock. The routine, if necessary, immediately raises IPL to the synchronization IPL of the device lock. Violations of IPL synchronization result in a SPLIPLHIGH bugcheck if full-checking multiprocessing is enabled.

In a full-checking multiprocessing environment, if it must spin wait for the requested lock to be released by another processor, SMP\$ACQUIREL temporarily restores the original IPL for the duration of the wait. If the original IPL was less than IPL\$_RESCHED, the spin wait occurs at IPL\$_RESCHED. SMP\$ACQUIREL exits with IPL at the device lock's synchronization IPL and the device lock held.

DESCRIPTION

The DEVICELOCK macro calls SMP\$ACQUIREL when NOSETIPL is *not* specified as its **condition** argument.

SMP\$ACQUIREL, having ensured that IPL has been set to the device lock's synchronization IPL, attempts to acquire the requested device lock, allowing the acquisition to succeed if the local processor already holds the lock or if the lock is unowned.

If the lock is unowned, the routine increments by 1 a counter that records the acquisition level. Each additional (or nested) acquisition of this lock by the owning processor again increments this counter.

If the lock is owned by another processor, the local processor spin waits until the lock is released.

SMP\$RELEASE

Releases all acquisitions of a fork lock or spin lock by the local processor and makes the lock available for acquisition by other processors.

module

SPINLOCKS

macro

FORKUNLOCK, UNLOCK

input

| Location | Contents |
|----------|------------------------------|
| R0 | Fork lock or spin lock index |

output

| Location | Contents |
|----------|------------------------------|
| R0 | Fork lock or spin lock index |

synchronization

Upon entry, the local processor must be executing at or above the IPL at which the lock was originally obtained. This IPL must be greater than IPL\$_ASTDEL. Violations of IPL synchronization in a full-checking multiprocessing environment result in a SPLIPLLOW bugcheck. At exit, IPL is unchanged and the lock is released.

DESCRIPTION

The FORKUNLOCK and UNLOCK macros call SMP\$RELEASE when the **condition=RESTORE** argument is not specified.

SMP\$RELEASE first verifies that the local processor owns the specified lock. If this is not the case, the procedure issues an SPLRELERR bugcheck. Otherwise, SMP\$RELEASE initializes the ownership count of the lock and releases the lock.

Operating System Routines

SMP\$RELEASEL

SMP\$RELEASEL

Releases all acquisitions of a device lock by the local processor and makes the lock available for acquisition by other processors.

module

SPINLOCKS

macro

DEVICEUNLOCK

input

| Location | Contents |
|----------|------------------------|
| RO | Address of device lock |

output

| Location | Contents |
|----------|------------------------|
| RO | Address of device lock |

synchronization

Upon entry, the local processor must be executing at or above the IPL at which the device lock was originally obtained. This IPL must be greater than IPL\$_ASTDEL. Violations of IPL synchronization in a full-checking multiprocessing environment result in a SPLIPLLOW bugcheck. At exit, IPL is unchanged and the device lock is released.

DESCRIPTION

The DEVICEUNLOCK macro calls SMP\$RELEASEL when the **condition=RESTORE** argument is not specified.

SMP\$RELEASEL first verifies that the local processor owns the specified device lock. If this is not the case, the procedure issues an SPLRELERR bugcheck. Otherwise, SMP\$RELEASEL initializes the ownership count of the device lock and releases the lock.

SMP\$RESTORE

Releases a single acquisition of a fork lock or spin lock held by the local processor.

module

SPINLOCKS

macro

FORKUNLOCK, UNLOCK

input

Location

Contents

R0

Fork lock or spin lock index

output

Location

Contents

R0

Fork lock or spin lock index

synchronization

Upon entry, the local processor must be executing at or above the IPL at which the lock was originally obtained. This IPL must be greater than IPL\$_ASTDEL. Violations of IPL synchronization in a full-checking multiprocessing environment result in a SPLIPLLOW bugcheck. At exit, IPL is unchanged and the lock may or may not be still held.

DESCRIPTION

The FORKUNLOCK and UNLOCK macros call SMP\$RESTORE when RESTORE is specified as the **condition** argument.

SMP\$RESTORE first verifies that the local processor owns the specified lock. If this is not the case, the procedure issues an SPLRSTERR bugcheck. Otherwise, SMP\$RESTORE proceeds to decrement the ownership count of the lock. If the ownership count of the lock drops to its initial state, the procedure releases the lock and makes it available to other processors.

Operating System Routines

SMP\$RESTOREL

SMP\$RESTOREL

Releases a single acquisition of a device lock held by the local processor.

module

SPINLOCKS

macro

DEVICEUNLOCK

input

| Location | Contents |
|----------|------------------------|
| RO | Address of device lock |

output

| Location | Contents |
|----------|------------------------|
| RO | Address of device lock |

synchronization

Upon entry, the local processor must be executing at or above the IPL at which the device lock was originally obtained. This IPL must be greater than IPL\$_ASTDEL. Violations of IPL synchronization in a full-checking multiprocessing environment result in a SPLIPLLOW bugcheck. At exit, IPL is unchanged and the device lock may or may not be still held.

DESCRIPTION

The DEVICEUNLOCK macro calls SMP\$RESTOREL when RESTORE is specified as its **condition** argument.

SMP\$RESTOREL first verifies that the local processor owns the specified device lock. If this is not the case, the procedure issues an SPLRSTERR bugcheck. Otherwise, SMP\$RESTOREL proceeds to decrement the ownership count of the device lock. If the ownership count of the device lock drops to its initial state, the procedure releases the lock and makes it available to other processors.

D

Device Driver Entry Points

This appendix describes the entry points VMS uses to activate a device driver.

Device Driver Entry Points

Alternate Start-I/O Routine

Alternate Start-I/O Routine

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

specified in

Specify the address of the alternate start-I/O routine in the **altstart** argument to the DDTAB macro. This macro places the address into DDT\$L_
ALTSTART.

called by

Called by routine EXE\$ALTQUEPKT in module SYSQIOREQ. A driver FDT routine generally is the caller of EXE\$ALTQUEPKT.

synchronization

An alternate start-I/O routine begins execution at fork IPL, holding the corresponding fork lock in a VMS multiprocessing environment. It must return control to its EXE\$ALTQUEPKT in this context.

context

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

register usage

An alternate start-I/O routine must preserve the contents of all registers except R0 through R5.

input

| Location | Contents |
|----------|----------------|
| R3 | Address of IRP |
| R5 | Address of UCB |

exit

The alternate start-I/O routine completes I/O requests by calling the routine COM\$POST. This routine places each IRP in the I/O postprocessing queue and returns control to the driver. The driver can then fetch another IRP from an internal queue. If no IRPs remain, the driver returns control to EXE\$ALTQUEPKT, which relinquishes fork level synchronization and returns to the driver FDT routine that called it. The FDT routine performs any postprocessing and transfers control to the routine EXE\$QIORETURN.

DESCRIPTION

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

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

Cancel-I/O Routine

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

specified in

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

called by

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

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

synchronization

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

context

A cancel-I/O routine executes in kernel mode in process context.

register usage

A cancel-I/O routine must preserve the contents of all registers except R0 through R5.

Device Driver Entry Points

Cancel-I/O Routine

input

| Location | Contents |
|----------|---|
| R2 | Channel index number |
| R3 | Address of IRP |
| R4 | Address of PCB of the process for which the I/O request is being canceled |
| R5 | Address of UCB |
| R8 | Reason for cancellation, one of the following: |

| Code | Meaning |
|---------------|---|
| CAN\$C_CANCEL | Called by \$CANCEL system service |
| CAN\$C_DASSGN | Called by \$DASSGN or \$DALLOC system service |

exit

The cancel-I/O routine issues an RSB instruction to return to its caller.

DESCRIPTION

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

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

Cloned UCB Routine

Performs device-specific initialization and verification of a cloned UCB.

specified in

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

called by

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

synchronization

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

context

A cloned UCB routine executes in kernel mode in process context.

register usage

A cloned UCB routine must preserve the contents of R2.

input

| Location | Contents |
|-------------------|---|
| R0 | SS\$_NORMAL |
| R2 | Address of cloned UCB |
| R3 | Address of DDT |
| R4 | Address of current PCB |
| R5 | Address of template UCB |
| UCB\$L_FQFL(R2) | Address of UCB\$L_FQFL(R2) |
| UCB\$L_FQBL(R2) | Address of UCB\$L_FQFL(R2) |
| UCB\$L_FPC(R2) | 0 |
| UCB\$L_FR3(R2) | 0 |
| UCB\$L_FR4(R2) | 0 |
| UCB\$W_BUFQUO(R2) | 0 |
| UCB\$L_ORB(R2) | Address of cloned ORB |
| UCB\$L_LINK(R2) | Address of next UCB in DDB chain |
| UCB\$L_IOQFL(R2) | Address of UCB\$L_IOQFL(R2) |
| UCB\$L_IOQBL(R2) | Address of UCB\$L_IOQFL(R2) |
| UCB\$W_UNIT(R2) | Device unit number |
| UCB\$W_CHARGE(R2) | Mailbox byte quota charge (UCB\$W_SIZE) |
| UCB\$W_REFC(R2) | 0 |

Device Driver Entry Points

Cloned UCB Routine

| | |
|------------------------------------|---|
| UCB\$_STS(R2) | UCB\$_DELETEUCB set, UCB\$_ONLINE set |
| UCB\$_DEVSTS(R2) | UCB\$_DELMBX set if DEV\$_MBX is set in UCB\$_DEVCHAR(R2) |
| UCB\$_OPCNT(R2) | 0 |
| UCB\$_SVAPTE(R2) | 0 |
| UCB\$_BOFF(R2) | 0 |
| UCB\$_BCNT(R2) | 0 |
| UCB\$_ORB(R2) | Address of cloned ORB |
| ORB\$_OWNER of template ORB | UIC of current process |
| ORB\$_ACL_MUTEX of template ORB | FFFF ₁₆ |
| ORB\$_FLAGS of template ORB | ORB\$_PROT_16 set |
| ORB\$_PROT of template ORB | 0 |
| ORB\$_ACL_COUNT of template ORB | 0 |
| ORB\$_ACL_DESC of template ORB | 0 |
| ORB\$_MIN_CLASS of template ORB | 0 in first longword |

exit

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

DESCRIPTION

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

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

Device Driver Entry Points

Controller Initialization Routine

Controller Initialization Routine

Prepares a controller for operation.

| specified in | Use the DPT_STORE macro to place the address of the controller initialization routine into CRB\$L_INTD+VEC\$L_INITIAL. | | | | | | | | | | |
|------------------------|---|----------|----------|----|-------------------------|----|---|----|---|----|-----------------------------|
| called by | SYSGEN calls a driver's controller initialization routine when processing a CONNECT command. Also, VMS calls this routine if the device, controller, processor, or adapter to which the device is connected experiences a power failure. | | | | | | | | | | |
| synchronization | <p>VMS calls a controller initialization routine at IPL\$_POWER. If it must lower IPL, the controller initialization routine cannot explicitly do so. Rather, it must fork. Because SYSGEN calls the unit initialization routine immediately after the controller initialization returns control to it, the driver's initialization routines must synchronize their activities. If the controller initialization routine forks, the unit initialization routine must be prepared to execute before the controller initialization routine completes.</p> <p>The portion of the controller initialization that services power failure cannot acquire any spin locks. As a result, the routine cannot fork to perform power failure servicing.</p> | | | | | | | | | | |
| context | Because a controller initialization routine executes within system context, it can refer only to those virtual addresses that reside in system (S0) space. | | | | | | | | | | |
| register usage | A controller initialization routine must preserve the contents of all registers except R0, R1, and R2. | | | | | | | | | | |
| input | <table><thead><tr><th>Location</th><th>Contents</th></tr></thead><tbody><tr><td>R4</td><td>Address of device's CSR</td></tr><tr><td>R5</td><td>Address of IDB associated with the controller</td></tr><tr><td>R6</td><td>Address of DDB associated with the controller</td></tr><tr><td>R8</td><td>Address of controller's CRB</td></tr></tbody></table> | Location | Contents | R4 | Address of device's CSR | R5 | Address of IDB associated with the controller | R6 | Address of DDB associated with the controller | R8 | Address of controller's CRB |
| Location | Contents | | | | | | | | | | |
| R4 | Address of device's CSR | | | | | | | | | | |
| R5 | Address of IDB associated with the controller | | | | | | | | | | |
| R6 | Address of DDB associated with the controller | | | | | | | | | | |
| R8 | Address of controller's CRB | | | | | | | | | | |
| exit | The controller initialization routine returns control to its caller with an RSB instruction. | | | | | | | | | | |

Device Driver Entry Points

Controller Initialization Routine

DESCRIPTION

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

- Determine whether it is being called as a result of a power failure by examining the power bit (UCB\$V_POWER in UCB\$L_STS) in the UCB. A controller initialization routine may want to perform or avoid specific tasks when servicing a power failure.
- Clear error-status bits in device registers.
- Enable controller interrupts.
- Allocate resources that must be permanently allocated to the controller.
- If the controller is dedicated to a single-unit device, such as a printer, fill in IDB\$L_OWNER and set the online bit (UCB\$V_ONLINE in UCB\$L_STS).
- For generic VAXBI devices, initialize BIIC and device hardware.

Driver Unloading Routine

A driver specifies a driver unloading routine if there is any device-specific work to do when the driver is unloaded and reloaded.

specified in

Specify the address of the driver unloading routine in the **unload** argument of the DPTAB macro. The driver-loading procedure puts the relative address of this routine in DPT\$W_UNLOAD.

called by

SYSGEN calls the driver unloading routine, if it exists, when executing a RELOAD command.

synchronization

SYSGEN calls a driver unloading routine at IPL\$_POWER. The driver unloading routine cannot lower IPL.

context

The driver unloading routine executes in process context.

register usage

The driver unloading routine can use all registers.

input

| Location | Contents |
|----------|----------------|
| R6 | Address of DDB |
| R10 | Address of DPT |

exit

The driver unloading routine returns exits with an RSB instruction. If it returns a success code (bit 0 set) in R0, SYSGEN proceeds to load the new version of the driver. If it returns a failure code (bit 0 clear), SYSGEN neither unloads the old version of the driver nor loads the new version.

DESCRIPTION

Because the driver unloading routine cannot lower IPL from IPL\$_POWER or obtain spin locks, it is of limited usefulness. It cannot safely modify I/O database fields, but can use COM\$DRVDEALMEM to return system buffers allocated by the driver to nonpaged pool.

Device Driver Entry Points

FDT Routines

FDT Routines

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

specified in

Use the FUNCTAB macro to specify the set of FDT routines that preprocess requests for I/O activity of a given type. Specify the names of the routines in the order in which you want them to execute for each type of I/O operation.

called by

The \$QIO system service calls a driver's FDT routines from the module SYSQIOREQ.

synchronization

FDT routines are called at IPL\$_ASTDEL and must exit at IPL\$_ASTDEL. FDT routines must not lower IPL below IPL\$_ASTDEL. If they raise IPL, they must lower it to IPL\$_ASTDEL before passing control to any other code. Similarly, before exiting they must release any spin locks they may acquire in a VMS multiprocessing environment.

context

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

register usage

FDT routines must preserve the contents of R3 through R8, the AP, and the FP.

input

| Location | Contents |
|----------|---|
| R0 | Address of FDT routine being called |
| R3 | Address of IRP |
| R4 | Address of PCB of the requesting process |
| R5 | Address of UCB of the device on which I/O activity is requested |
| R6 | Address of CCB that describes the user-specified process-I/O channel |
| R7 | Number of the bit that specifies the code for the requested I/O function |
| R8 | Address of entry in the function decision table that dispatched control to this FDT routine |
| AP | Address of first function-dependent argument (p1) specified in the \$QIO request |

Device Driver Entry Points

FDT Routines

exit

In a set of FDT routines associated with an I/O function, each, except the last, must return control to its caller by means of an RSB instruction. The last must exit using one of the following mechanisms:

| Exit Mechanism | Function |
|--------------------|--|
| JMP EXE\$ABORTIO | Aborts an I/O request and returns status to the caller of the \$QIO system service in R0. |
| JSB EXE\$ALTQUEPKT | Queues an IRP to the driver's alternate start-I/O routine without checking the status of the device. |
| JMP EXE\$FINISHIO | Completes the processing of an I/O request, returning status to the caller of the \$QIO system service. (EXE\$FINISHIO takes the status information from R0 and R1 and returns it in the IOSB specified in the call to \$QIO.) |
| JMP EXE\$FINISHIOC | Completes the I/O processing of an I/O request, returning status to the caller of the \$QIO system service. (EXE\$FINISHIOC takes the status information from R0 and returns it in the IOSB specified in the call to \$QIO, clearing the second longword of the IOSB.) |
| JMP EXE\$QIODRVPKT | Inserts an IRP into a device's pending-I/O queue if the device is busy, or starts I/O activity if the device is idle. |

DESCRIPTION

FDT routines validate the function-dependent arguments to a \$QIO system service request and prepare the I/O database to service the request. For each function that a device supports, a set of FDT routines must provide preprocessing of requests for that function. For a function that does not involve an I/O transfer, a set of FDT routines may complete its processing. Otherwise FDT routines can abort the request, pass it to the next FDT routine in the set, or pass it to a VMS routine that delivers it to the driver.

Device Driver Entry Points

Interrupt Service Routine

Interrupt Service Routine

Processes interrupts generated by a device.

specified in

UNIBUS, Q22-bus, and generic VAXBI devices require an interrupt service routine for each interrupt vector the device has. Use the `DPT_STORE` macro to place the address of the interrupt service routine into `CRB$L_INTD+VEC$L_ISR`.

If the device has two interrupt vectors, use the `DPT_STORE` macro to place the address of the second interrupt service routine into `CRB$L_INTD2+VEC$L_ISR`.

Tape devices on the MASSBUS require an interrupt service routine that interrogates the tape formatter (the controller) to determine which drive needs attention and whether the interrupt is unsolicited.

Disk devices on the MASSBUS use the interrupt service routine provided by VMS and do not need to provide their own interrupt service routine.

called by

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

synchronization

A driver's interrupt service routine is called, executes, and returns at device IPL. In a VMS multiprocessing environment, the interrupt service routine must obtain the device lock associated with its device IPL. It performs this acquisition as soon as it obtains the address of the UCB of the interrupting device. It must release this device lock before dismissing the interrupt.

context

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

register usage

If an interrupt service routine uses R6 through R11, the AP, or the FP, it must first save the contents of those registers, restoring their contents before exiting by means of the REI instruction. MASSBUS drivers must also preserve the contents of R0 and R1.

Device Driver Entry Points

Interrupt Service Routine

input

| Location | Contents |
|------------------|---|
| 00(SP) | Address of longword that contains the address of the IDB |
| 04(SP) to 24(SP) | <i>For UNIBUS, Q22-bus, and generic VAXBI devices, the contents of R0 through R5 at the time of the interrupt</i> |
| 28(SP) | <i>For UNIBUS, Q22-bus, and generic VAXBI devices, PC at the time of the interrupt</i> |
| 32(SP) | <i>For UNIBUS, Q22-bus, and generic VAXBI devices, PSL at the time of the interrupt</i> |
| 04(SP) to 16(SP) | <i>for MASSBUS devices, the contents of R2 through R5 at the time of the interrupt</i> |
| 20(SP) | <i>For MASSBUS devices, PC at the time of the interrupt</i> |
| 24(SP) | <i>For MASSBUS devices, PSL at the time of the interrupt</i> |

exit

Before an interrupt service routine transfers control to the suspended driver, it must restore the contents of R3 and R4 from the UCB. It then transfers control to the address saved in UCB\$L_FPC.

When it regains control (after the suspended driver forks), an interrupt service routine removes the address of the pointer to the IDB from the top of the stack and restores the registers VMS saved when dispatching the interrupt (R0 through R5 for UNIBUS, Q22-bus, and generic VAXBI interrupt service routines, R2 through R5 for MASSBUS interrupt service routines). Finally, an interrupt service routine dismisses the interrupt with an REI instruction.

DESCRIPTION

An interrupt service routine performs the following functions:

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

For MASSBUS devices, a VMS interrupt service routine performs these functions.

Device Driver Entry Points

Register Dumping Routine

Register Dumping Routine

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

specified in

Specify the name of the register dumping routine in the **regdmp** argument of the DDTAB macro. This macro places the address of the routine into DDT\$L_REGDUMP.

called by

The VMS error logging routines (ERL\$DEVICERR, ERL\$DEVICTMO, and ERL\$DEVICEATTN) and diagnostic buffer filling routine (IOC\$DIAGBUFILL) call the register dumping routine.

synchronization

VMS calls a register dumping routine at the same IPL at which the driver called the VMS routine ERL\$DEVICERR, ERL\$DEVICTMO, ERL\$DEVICEATTN, or IOC\$DIAGBUFILL. A register dumping routine must not change IPL.

context

A register dumping routine executes within the context of an interrupt service routine or a driver fork process, using the kernel-mode stack. As a result, it can only refer to those virtual addresses that reside in system (S0) space.

register usage

The register dumping routine preserves the contents of all registers except R0 through R2. If it uses the stack, the register dumping routine must restore the stack before passing control to another routine, waiting for an interrupt, or returning control to its caller.

input

| Location | Contents |
|----------|---|
| R0 | Address of buffer into which a register dumping routine copies the contents of device registers |
| R4 | Address of device's CSR (if the driver invoked the WFIKPCH macro to wait for an interrupt or timeout) |
| R5 | Address of UCB |

exit

The register dumping routine issues an RSB instruction to return to its caller.

DESCRIPTION

A register dumping routine fills the indicated buffer as follows:

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

Start-I/O Routine

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

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

called by The start-I/O routine is called by IOC\$INITIATE and IOC\$REQCOM in module IOSUBNPAG.

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

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

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

register usage A start-I/O routine must preserve the contents of all registers except R0, R1, R2, and R4. If the start-I/O routine uses the stack, it must restore the stack before completing the request, waiting for an interrupt, or requesting system resources.

input

| Location | Contents |
|---------------|---|
| R3 | Address of IRP |
| R5 | Address of UCB |
| UCB\$W_BCNT | Number of bytes to be transferred, copied from the low-order word of IRP\$L_BCNT |
| UCB\$W_BOFF | Byte offset into first page of direct-I/O transfer; for buffered-I/O transfers, number of bytes to be charged to the process allocating the buffer. |
| UCB\$L_SVAPTE | For a <i>direct-I/O</i> transfer, virtual address of first page-table entry (PTE) of I/O-transfer buffer; for <i>buffered-I/O</i> transfer, address of buffer in system address space |

Device Driver Entry Points

Start-I/O Routine

exit

The start-I/O routine suspends itself whenever it must wait for a required resource, such as a controller data channel or UNIBUS/Q22 bus map registers. To do so, it invokes a VMS macro (such as REQPCAN or REQMPR) that saves its context in the UCB fork block, places the UCB in a resource wait queue, and returns control to the caller of the start-I/O routine.

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

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

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

DESCRIPTION

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

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

Timeout Handling Routine

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

specified in

Specify the address of the timeout handling routine in the **except** argument to the WFIKPCH or the WFIRLCH macro.

called by

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

Routines in the VMS module TIMESCHDL call the timeout handling routine at the request of the WFIKPCH and WFIRLCH macros.

synchronization

A timeout handling routine is called at device IPL and must return to its caller at device IPL. In a VMS multiprocessing environment, the processor holds both the fork lock and device lock associated with the device at the time of the call.

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

- Issue a DEVICEUNLOCK macro to lower to fork level
- Perform timeout handling activities possible at the lower IPL
- Issue a DEVICELOCK macro to again obtain the device lock and raise to device IPL
- Issue an RSB instruction to return to its caller

context

Because a timeout handling routine executes in the context of a fork process, it can access only those virtual addresses that refer to system (S0) space.

register usage

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

Device Driver Entry Points

Timeout Handling Routine

input

| Location | Contents |
|------------|--|
| R3 | Contents of R3 when the last invocation of WFIKPCH or WFIRLCH took place |
| R4 | Contents of R4 when the last invocation of WFIKPCH or WFIRLCH took place |
| R5 | Address of UCB of the device |
| UCB\$L_STS | UCB\$V_INT and UCB\$V_TIM clear; UCB\$V_TIMOUT set |

exit

The timeout handling routine issues an RSB instruction to return to its caller.

DESCRIPTION

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

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

Unit Delivery Routine

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

specified in

Specify the name of the unit delivery routine in the **deliver** argument to the DPTAB macro. The macro puts the relative address of this routine in DPT\$W_DELIVER.

called by

SYSGEN's AUTOCONFIGURE command calls the unit delivery routine once for each unit the controller is capable of controlling. This value is specified in the **defunits** argument to the DPTAB macro.

synchronization

The unit delivery routine is called at IPL\$_POWER. It must not lower IPL.

context

The unit delivery routine executes in the context of the process within which SYSGEN executes.

register usage

The unit delivery routine can use R0, R1, and R2 freely, but must preserve the contents of all other registers.

input

| Location | Contents |
|----------|---|
| R3 | Address of IDB; 0 if none exists |
| R4 | Address of device's CSR |
| R5 | Number of unit that the unit delivery routine must decide to configure or not to configure |
| R6 | Address of start of the UNIBUS adapter's or Q22-bus's I/O space (UNIBUS/Q22-bus devices); address of MBA configuration register (MASSBUS devices) |
| R7 | Address of AUTOCONFIGURE command's configuration control block (ACF) |
| R8 | Address of ADP |

Device Driver Entry Points

Unit Delivery Routine

exit

A unit delivery routine issues an RSB instruction to return control to the SYSGEN autoconfiguration facility. If the routine returns error status in R0, SYSGEN does not configure the unit.

Note that, because generic VAXBI devices are not recognized by SYSGEN's autoconfiguration facility, their drivers do not contain a unit delivery routine.

DESCRIPTION

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

Unit Initialization Routine

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

specified in

You can specify a unit initialization routine in two ways, either of which will suffice for all but a few specific devices.

- Specify the address of the unit initialization routine **unitinit** argument of the DDTAB macro. This macro places the address of the routine into DDT\$L_UNITINIT. MASSBUS device drivers *must* use this method.
- Use the DPT_STORE macro to place the address of the unit initialization routine into CRB\$L_INTD+VEC\$L_UNITINIT.

called by

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

synchronization

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

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

context

Because VMS calls it in system context, a unit initialization routine can only refer to those virtual addresses that reside in system (S0) space.

register usage

A unit initialization routine must preserve the contents of all registers except R0, R1, and R2.

input

| Location | Contents |
|----------|---|
| R3 | Address of primary CSR. |
| R4 | Address of secondary CSR, if it exists. (If it does not, the contents of R4 are the same as those of R3.) |
| R5 | Address of UCB. |

exit

The unit initialization routine returns control to its caller with an RSB instruction.

Device Driver Entry Points

Unit Initialization Routine

DESCRIPTION

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

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

Device Driver Entry Points

Unsolicited Interrupt Service Routine

Unsolicited Interrupt Service Routine

Services an interrupt from a MASSBUS disk that is not the result of a driver's request.

specified in

Specify the name of the unsolicited interrupt service routine in the **unsolic** argument to the DDTAB macro. This macro places the address of the routine into DDT\$L_UN SOLINT.

called by

The MASSBUS adapter's interrupt service routine (MBA\$INT in module ADPERRSUB of the SYSLOA facility) calls a driver's unsolicited interrupt service routine.

synchronization

An unsolicited interrupt service routine is called, executes, and returns at device IPL.

context

Because the unsolicited interrupt service routine executes in kernel mode on the interrupt stack, it can only refer to those addresses that reside in system (S0) space.

register usage

The unsolicited interrupt service routine must not alter the contents of registers R6 through R11, the AP, or the FP.

input

| Location | Contents |
|----------|---|
| R4 | Address of MBA's configuration register |
| R5 | Address of UCB |

exit

An unsolicited interrupt service routine issues an RSB instruction to return control to the MASSBUS adapter's interrupt service routine.

DESCRIPTION

Only drivers of MASSBUS disks must provide unsolicited interrupt service routines. All other devices detect unsolicited interrupts in their interrupt service routines.

The routine that handles these unsolicited interrupts must determine the nature of the interrupt and act accordingly, depending on the characteristics of the device and controller. Examples of such unsolicited interrupts include disks being placed on line or taken off line.

E

Sample Driver for the RL11, RL01, and RL02

This example driver, DLDRIVER, drives a disk device on both the UNIBUS and the Q22 bus.

```
.TITLE DLDRIVER - VAX/VMS RL11/RL01,RL02 DISK DRIVER
.IDENT 'X-7'
;
;*****
;*
;* COPYRIGHT (c) 1978, 1980, 1982, 1984 BY
;* DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.
;* ALL RIGHTS RESERVED.
;*
;* THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
;* ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
;* INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER
;* COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
;* OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
;* TRANSFERRED.
;*
;* THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
;* AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
;* CORPORATION.
;*
;* DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
;* SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
;*
;*****
;
; FACILITY:
;
;     VAX/VMS RL11/RL01,RL02 DISK DRIVER
;
; **
; PAGE
; ABSTRACT:
;
;     THIS MODULE CONTAINS THE TABLES AND ROUTINES NECESSARY TO
;     PERFORM ALL DEVICE-DEPENDENT PROCESSING OF AN I/O REQUEST
;     FOR RL11/RL01,RL02 DISK TYPES ON A VAX/VMS SYSTEM.
;
;     THE DISKS HAVE THE FOLLOWING PHYSICAL GEOMETRY:
;
;           # CYL      TRACKS/   SECTORS/   BYTES/   MAXIMUM
;           #         CYLINDER  TRACK      SECTOR   BLOCKS
; RL01      256         2         40        256      10240
; RL02      512         2         40        256      20480
;
;     SINCE THE SECTOR SIZE IS ONLY 1/2 BLOCK, LOGICAL TO PHYSICAL
;     CONVERSION OF THE DISK ADDRESS IS DONE IN THE DRIVER STARTIO
;     ROUTINE RATHER THAN IN THE IOC$CVTLOGPHY FDT ROUTINE.
;
;     OVERLAPPED SEEKS ARE NOT ATTEMPTED BECAUSE THE DEVICE DOES
```

Sample Driver for the RL11, RL01, and RL02

```
NOT INTERRUPT AT THE COMPLETION OF A SEEK.

ALSO, THE DEVICE DOES NOT PERFORM AN IMPLICIT SEEK WHEN PERFORMING
A READ OR WRITE FUNCTION, SO SEEK FUNCTIONS ARE ISSUED BY THIS
DRIVER WHERE NECESSARY PRIOR TO ISSUING A READ OR WRITE FUNCTION.
THE READ OR WRITE FUNCTION IS THEN ISSUED AS SOON AS THE RL11
CONTROLLER BECOMES READY (WHILE THE SEEK IS IN PROGRESS), AND A
WAIT FOR INTERRUPT (UPON COMPLETION OF THE READ OR WRITE) IS
ISSUED. IF A SEEK FUNCTION IS REQUESTED SEPARATELY FROM A READ OR
WRITE, A DUMMY READ HEADER FUNCTION IS ISSUED FOLLOWING THE SEEK
FUNCTION AND A WAIT FOR INTERRUPT (UPON COMPLETION OF THE READ
HEADER) IS ISSUED.

THE IO$X_INHSEEK FUNCTION MODIFIER IS TREATED AS A NO-OP BY
THIS DRIVER, SINCE AN EXPLICIT SEEK IS NECESSARY FOR THE RL02
TO TRANSFER DATA PROPERLY.

THE RL'S DO NOT READ OR WRITE BEYOND THE END OF TRACK (THEY DO NOT
AUTOMATICALLY SEEK THE NEXT TRACK), SO ALL READ AND WRITE FUNCTIONS
ARE BROKEN UP BY THIS DRIVER INTO PARTIAL TRANSFERS TO THE END OF
TRACK, FOLLOWED BY A SEEK TO THE NEXT TRACK, THEN ANOTHER READ OR
WRITE FUNCTION UNTIL THE TOTAL DATA TRANSFER IS COMPLETE.

--
.PAGE
.SBTTL EXTERNAL AND LOCAL DEFINITIONS

; EXTERNAL SYMBOLS

$ADPDEF          ;DEFINE ADAPTER CONTROL BLOCK
$CRBDEF          ;DEFINE CHANNEL REQUEST BLOCK
$DCDEF           ;DEFINE DEVICE CLASS
$DDBDEF          ;DEFINE DEVICE DATA BLOCK
$DEVDEF          ;DEFINE DEVICE CHARACTERISTICS
$DPTDEF          ;DEFINE DRIVER PROLOGUE TABLE
$DYNDEF          ;DEFINE DYNAMIC DATA STRUCTURE TYPES
$EMBDEF          ;DEFINE ERROR MESSAGE BUFFER
$IIDDEF          ;DEFINE INTERRUPT DATA BLOCK
$IODEF           ;DEFINE I/O FUNCTION CODES
$IRPDEF          ;DEFINE I/O REQUEST PACKET
$PRDEF           ;DEFINE PROCESSOR REGISTERS
$PTEDEF          ;DEFINE SYSTEM PTES
$SSDEF           ;DEFINE SYSTEM STATUS CODES
$UCBDEF          ;DEFINE UNIT CONTROL BLOCK
$VADEF           ;DEFINE VIRTUAL ADDRESS BITS
$VECDEF          ;DEFINE INTERRUPT VECTOR BLOCK

; LOCAL MACROS

; EXFUNCL
; BRANCH TO SUBROUTINE WHICH REQUESTS CHANNEL (IF NOT ALREADY OWNED),
; EXECUTES FCODE (OR R3) FUNCTION, AND BRANCHES TO BDST ON ERROR

.MACRO EXFUNCL BDST,FCODE
  .IF NB FCODE          ;IS FCODE NONBLANK?
  MOVZBL #CD'FCODE,R3  ;IF NB - SPECIFY FCODE FUNCTION
  .ENDC                 ;IF B - SPECIFY FMTN IN EXISTING R3
  BSBW FEXL             ;EXECUTE FUNCTION
  .BYTE BDST--1         ;WHERE TO GO IF ERROR
.ENDM
```

Sample Driver for the RL11, RL01, and RL02

```
;
GENF
; GENERATE FUNCTION TABLE ENTRY AND CASE TABLE INDEX SYMBOL
;
.MACRO GENF FCODE
CD'FCODE=-.FTAB/2
.WORD FCODE!RL_CS_M_IE ;FCODE WITH INT ENABLE BIT
.ENDM

;
CKPWR
; DISABLE INTERRUPTS, CHECK IF POWER HAS FAILED,
; AND PUT DEVICE UNIT NUMBER IN R2<9:8>
;
.MACRO CKPWR SAVE_RO=YES,?L1
CLRL R2 ;CLEAR R2 FOR UNIT NUMBER
INSV UCB$W_UNIT(R5),- ;PUT UNIT # IN R2<9:8>
#8,#2,R2 ;...
DEVICELOCK -
LOCKADDR=UCB$L_DLCK(R5),- ; LOCK DEVICE ACCESS
LOCKIPL=UCB$B_DIPL(R5),- ; RAISE IPL
SAVIPL=- (SP),- ;SAVE CURRENT IPL
PRESERVE='SAVE_RO
SETIPL #31,- ;DISABLE ALL INTERRUPTS
ENVIRON=UNIPROCESSOR
BBC #UCB$V_POWER,- ;IF CLR - NO POWER FAILURE
UCB$W_STS(R5),L1 ;...
; POWERFAILURE!
DEVICEUNLOCK -
LOCKADDR=UCB$L_DLCK(R5),- ; UNLOCK DEVICE ACCESS
NEWIPL=(SP)+,- ;RESTORE IPL
PRESERVE='SAVE_RO
BRW RETREG ;EXIT
L1: ;RETURN FOR NO POWER FAILURE
.ENDM

;
; LOCAL SYMBOLS
;
RL_NUM_REGS =4 ;NUMBER OF DEVICE REGISTERS
RL_SLM =5 ;STATE=SEEK LINEAR MODE (READY TO GO)
UCB$B_DL_DCHEK =UCB$W_OFFSET+1 ;REDEFINE FOR DATA CHECK USE
;
; UCB OFFSETS WHICH FOLLOW THE STANDARD UCB FIELDS
;
$DEFINI UCB ;START OF UCB DEFINITIONS
```

Sample Driver for the RL11, RL01, and RL02

```

.=UCB$K_LCL_DISK_LENGTH
$DEF UCB$W_DL_PBCR .BLKW 1 ;BEGIN DEFINITIONS AT END OF UCB
$DEF UCB$W_DL_CS .BLKW 1 ;PARTIAL BYTE COUNT
$DEF UCB$W_DL_BA .BLKW 1 ;CONTROL STATUS REGISTER
$DEF UCB$W_DL_DA .BLKW 1 ;BUS ADDRESS REGISTER
$DEF UCB$W_DL_DA .BLKW 1 ;DISK ADDRESS REGISTER
$DEF UCB$W_DL_MP .BLKW 1 ;MULTIPURPOSE REGISTER
$DEF UCB$W_DL_DPN .BLKW 1 ;DATA PATH NUMBER
$DEF UCB$L_DL_SVAPTE .BLKW 1 ;SAVED SVAPTE OF THE USER'S BUFFER
$DEF UCB$L_DL_DPR .BLKL 1 ;DATAPATH REGISTER
$DEF UCB$L_DL_BUFADR .BLKW 1 ;USER BUFFER ADDRESS
$DEF UCB$L_DL_FMPR .BLKL 1 ;FINAL MAP REGISTER
$DEF UCB$A_DL_MOVRTN .BLKW 1 ;BUFFER MOVE ROUTINE ADDRESS
$DEF UCB$L_DL_MMPR .BLKL 1 ;PREVIOUS MAP REGISTER
$DEF UCB$E_DL_DPPE .BLKB 1 ;DATAPATH PURGE ERROR
$DEF UCB$W_DL_DB .BLKW 3 ;DATA BUFFER REGISTER
$DEF UCB$E_DL_XBA .BLKB 1 ;BUS ADDRESS EXTENSION BITS
$DEF UCB$W_DL_SBA .BLKW 1 ;SAVED BUFFER ADDRESS
$DEF UCB$A_DL_BUF_VA .BLKL 1 ;PHYSICAL BUFFER VIRTUAL ADDRESS
$DEF UCB$A_DL_BUF_PA .BLKL 1 ;PHYSICAL BUFFER PHYSICAL ADDRESS
$DEF UCB$W_DL_FLAGS .BLKW 1 ;FLAGS
$VIELD UCB,0,<- ;START THE FLAG DEFINITIONS
      <DL_22BIT,,M>,- ;22 BIT ADDRESSING
      <DL_MAPPING,,M>,- ;ADAPTER MAPPING
      > ;END OF FLAG DEFINITIONS
$DEF UCB$K_DL_LEN .BLKW 1 ;LENGTH OF UCB
$EQU UCB$K_DL_BUFSZ 20 ;BUFFER SIZE = 40 SECTORS *
                                ;256 BYTES/SECTOR / 512 BYTES/PAGE
$DEFEND UCB ;END OF UCB DEFINITIONS

;
; RL11/RL01 REGISTER OFFSETS FROM CSR ADDRESS
;
$DEFINI RL ; START OF REGISTER DEFINITIONS
$DEF RL_CS .BLKW 1 ;CONTROL STATUS REGISTER (CSR)
  _VIELD RL_CS,0,<- ;START OF CSR BIT DEFINITIONS
    <DRDY,,M>,- ; DRIVE READY
    <FCODE,3>,- ; FUNCTION CODE
    <XBA,2>,- ; BUS ADDRESS EXTENSION BITS
    <IE,,M>,- ; INTERRUPT ENABLE
    <CRDY,,M>,- ; CONTROLLER READY
    <DS,2>,- ; DRIVE SELECT
    <OPI,,M>,- ; OPERATION INCOMPLETE
    <CRC,,M>,- ; DATA CRC OR HEADER CRC
    <DLT,,M>,- ; DATA LATE OR HEADER NOT FOUND
    <NXM,,M>,- ; NONEXISTENT MEMORY
    <DE,,M>,- ; DRIVE ERROR
    <CE,,M>- ; COMPOSITE ERROR
  > ;END CSR BIT DEFINITIONS
$DEF RL_BA .BLKW 1 ;BUS ADDRESS REGISTER (BAR)
$DEF RL_DA .BLKW 1 ;DISK ADDRESS REGISTER (DAR)
  _VIELD RL_DA,0,<- ;START OF DAR BIT DEFINITIONS
    <MRK,,M>,- ; MARK (ALWAYS 1)
    <STS,,M>,- ; GET STATUS
    <,1>,- ; RESERVED BIT
    <RST,,M>,- ; RESET
    <,12>,- ; RESERVED BITS
  > ;END OF DAR BIT DEFINITIONS

```

Sample Driver for the RL11, RL01, and RL02

```

$DEF  RL_MP          .BLKW  1      ;MULTIPURPOSE REGISTER (MPR)
      _VIELD RL_MP,0,<-          ;START OF MPR BIT DEFINITIONS
      <STA,3>,-                ; DRIVE STATE
      <BH,,M>,-                ; BRUSH HOME
      <HO,,M>,-                ; HEADS OUT
      <CO,,M>,-                ; COVER OPEN
      <HS,,M>,-                ; HEAD SELECT
      <TYP,,M>,-               ; DRIVE TYPE
      <DSE,,M>,-               ; DRIVE SELECT ERROR
      <VC,,M>,-                ; VOLUME CHECK
      <WGE,,M>,-               ; WRITE GATE ERROR
      <SPE,,M>,-               ; SPIN ERROR
      <SKTO,,M>,-              ; SEEK TIME OUT
      <WL,,M>,-                ; WRITE LOCK
      <CHE,,M>,-               ; CURRENT HEAD ERROR
      <WDE,,M>,-               ; WRITE DATA ERROR
      >                          ;END MPR BIT DEFINITIONS

$DEF  RL_BAE          .BLKW  1      ; BUS ADDRESS EXTENSION REGISTER(BAE)
      $DEFEND RL                ;END RL11/RL01 REGISTER DEFINITIONS

;
; HARDWARE FUNCTION CODES
;
F_NOP=0*2                        ;NO OPERATION
F_UNLOAD=F_NOP                   ;NO OPERATION
F_SEEK=3*2                       ;SEEK CYLINDER
F_RECAL=F_NOP                    ;NO OPERATION
F_DRVCLR=2*2                     ;DRIVE CLEAR (GET STATUS)
F_RELEASE=F_NOP                  ;NO OPERATION
F_OFFSET=F_NOP                   ;NO OPERATION
F_RETCENTER=F_NOP                ;NO OPERATION
F_PACKACK=2*2                    ;PACK ACKNOWLEDGE (SET VOLUME VALID)
F_SEARCH=F_NOP                   ;NO OPERATION
F_WRITECHECK=1*2                 ;WRITE CHECK
F_WRITEDATA=5*2                  ;WRITE DATA
F_WRITEHEAD=F_NOP                ;NO OPERATION
F_READDATA=6*2                   ;READ DATA
F_READHEAD=4*2                   ;READ HEADER
F_AVAILABLE=F_NOP                ;NO OPERATION
F_GETSTATUS=2*2                  ;GET STATUS (DRIVER INTERNAL USE)

      .PAGE
      .SBTTL STANDARD TABLES

;
; DRIVER PROLOGUE TABLE
;
; THE DPT DESCRIBES DRIVER PARAMETERS AND I/O DATABASE FIELDS
; THAT ARE TO BE INITIALIZED DURING DRIVER LOADING AND RELOADING
;
;
DPTAB  -                          ;DPT CREATION MACRO
      END=DL_END,-                ;END OF DRIVER LABEL
      ADAPTER=UBA,-               ;ADAPTER TYPE = UNIBUS
      FLAGS=DPT$M_SVP,-           ;SYSTEM PAGE-TABLE ENTRY REQUIRED
      UCBSIZE=UCB$K_DL_LEN,-      ;LENGTH OF UCB
      NAME=DLDRIVER               ;DRIVER NAME

```

Sample Driver for the RL11, RL01, and RL02

```

DPT_STORE INIT ;START CONTROL BLOCK INIT VALUES
DPT_STORE DDB,DCB,DCB$L_ACPD,L,<^A\F11> ;DEFAULT ACP NAME
DPT_STORE DDB,DCB,DCB$L_ACPD+3,B,DCB$K_CART ;ACP CLASS
DPT_STORE UCB,UCB$B_FLCK,B,SPL$C_IOLOCK8 ;FORK LOCK INDEX
DPT_STORE UCB,UCB$L_DEVCHAR,L,- ;DEVICE CHARACTERISTICS
    <DEV$M_FOD- ;FILES ORIENTED
    !DEV$M_DIR- ;DIRECTORY STRUCTURED
    !DEV$M_AVL- ;AVAILABLE
    !DEV$M_ELG- ;ERROR LOGGING
    !DEV$M_SHR- ;SHAREABLE
    !DEV$M_IDV- ;INPUT DEVICE
    !DEV$M_ODV- ;OUTPUT DEVICE
    !DEV$M_RND> ;RANDOM ACCESS
DPT_STORE UCB,UCB$L_DEVCHAR2,L,- ;DEVICE CHARACTERISTICS
    <DEV$M_NNM> ;PREFIX NAME WITH "node$"
DPT_STORE UCB,UCB$B_DEVCLASS,B,DCB$DISK ;DEVICE CLASS
DPT_STORE UCB,UCB$W_DEVBUFSIZ,W,512 ;DEFAULT BUFFER SIZE
DPT_STORE UCB,UCB$B_SECTORS,B,40 ;NUMBER OF SECTORS PER TRACK
DPT_STORE UCB,UCB$B_TRACKS,B,2 ;NUMBER OF TRACKS PER CYLINDER
DPT_STORE UCB,UCB$B_DIPL,B,21 ;DEVICE IPL
DPT_STORE UCB,UCB$B_ERTMAX,B,8 ;MAX ERROR RETRY COUNT
DPT_STORE UCB,UCB$W_DEVSTS,W,- ;INHIBIT LOG TO PHYS CONVERSION IN FDT
    <UCB$M_NOCNVRT> ;...

DPT_STORE REINIT ;START CONTROL BLOCK RE-INIT VALUES
DPT_STORE CRB,CRB$L_INTD+4,D,DL_INT ;INTERRUPT SERVICE ROUTINE ADDRESS
DPT_STORE CRB,CRB$L_INTD+VEC$L_INITIAL,- ;CONTROLLER INIT ADDRESS
    D,DL_RL11_INIT ;...
DPT_STORE CRB,CRB$L_INTD+VEC$L_UNITINIT,- ;UNIT INIT ADDRESS
    D,DL_RLOX_INIT ;...
DPT_STORE DDB,DCB,DCB$L_DDT,D,DL$DDT ;DDT ADDRESS

DPT_STORE END ;END OF INITIALIZATION TABLE

;
; DRIVER DISPATCH TABLE
;
; THE DDT LISTS ENTRY POINTS FOR DRIVER SUBROUTINES WHICH ARE
; CALLED BY THE OPERATING SYSTEM.
;

DDTAB - ;DDT CREATION MACRO
    DEVNAM=DL,- ;NAME OF DEVICE
    START=DL_STARTIO,- ;START I/O ROUTINE
    UNSOLIC=DL_UNSOINT,- ;UNSOLICITED INTERRUPT
    FUNCTB=DL_FUNCTABLE,- ;FUNCTION DECISION TABLE
    CANCEL=0,- ;CANCEL=NO-OP FOR FILES DEVICE
    REGDMP=DL_REGDUMP,- ;REGISTER DUMP ROUTINE
    DIAGBF=<<<RL_NUM_REGS+5+5+3+1>*4>,- ;BYTES IN DIAG BUFFER
    ERLGBF=<<<<RL_NUM_REGS+5+1>*4>+EMB$L_DV_REGSAV> ;BYTES IN
    ;ERROR LOG BUFFER

; DIAGNOSTIC BUFFER SIZE = <<<4 RLO2 REGISTER LONGWORDS + 5 UCB FIELD LONGWORDS
; + 5 IOC$DIAGBUFILL LONGWORDS + 3 BUFFER ALLOCATION
; LONGWORDS + 1 LONGWORD FOR # REGISTERS IN DL_REGDUMP>
; * 4 BYTES/LONGWORD>
;
; ERROR LOG BUFFER SIZE = <<<<4 RLO2 REGISTER LONGWORDS + 5 UCB FIELD LONGWORDS
; + 1 LONGWORD FOR # REGISTERS IN DL_REGDUMP>
; * 4 BYTES/LONGWORD> + BYTES NEEDED FOR ERROR LOGGER
; TO SAVE SOFTWARE REGISTERS>

```

Sample Driver for the RL11, RL01, and RL02

```

;
; HARDWARE FUNCTION CODE TABLE
;
; THIS TABLE MERGES THE FUNCTION CODE BITS WITH THE
; INTERRUPT ENABLE BIT AND GENERATES THE CASE TABLE
; INDEX SYMBOL.
;
FTAB:  GENF    F_NOP                ;NO-OP
        GENF    F_UNLOAD            ;UNLOAD VOLUME (NOP)
        GENF    F_SEEK              ;SEEK
        GENF    F_RECAL             ;RECALIBRATE (NOP)
        GENF    F_DRVCLR            ;DRIVE CLEAR (RESET & GET STATUS)
        GENF    F_RELEASE           ;RELEASE PORT (NOP)
        GENF    F_OFFSET            ;OFFSET HEADS (NOP)
        GENF    F_RETCENTER         ;RETURN HEADS TO CENTERLINE (NOP)
        GENF    F_PACKACK           ;PACK ACKNOWLEDGE (RESET & GET STATUS)
        GENF    F_SEARCH            ;SEARCH (NOP)
        GENF    F_WRITECHECK        ;WRITE CHECK
        GENF    F_WRITEDATA         ;WRITE DATA
        GENF    F_READDATA          ;READ DATA
        GENF    F_WRITEHEAD         ;WRITE HEADERS (NOP)
        GENF    F_READHEAD          ;READ HEADERS
        GENF    F_NOP               ;place holder
        GENF    F_NOP               ;place holder
        GENF    F_AVAILABLE        ;AVAILABLE

```

.PAGE

```

;
; FUNCTION DECISION TABLE
;
; THE FDT LISTS VALID FUNCTION CODES, SPECIFIES WHICH
; CODES ARE BUFFERED, AND DESIGNATES SUBROUTINES TO
; PERFORM PREPROCESSING FOR PARTICULAR FUNCTIONS.
;

```

```

DL_FUNCTABLE:
FUNCTAB , -                ;LIST LEGAL FUNCTIONS
        <NOP, -            ; NO-OP
        UNLOAD, -         ; UNLOAD
        SEEK, -           ; SEEK
        DRVCLR, -        ; DRIVE CLEAR
        PACKACK, -       ; PACK ACKNOWLEDGE
        SENSECHAR, -     ; SENSE CHARACTERISTICS
        SETCHAR, -       ; SET CHARACTERISTICS
        SENSEMODE, -     ; SENSE MODE
        SETMODE, -       ; SET MODE
        WRITECHECK, -    ; WRITE CHECK
        READHEAD, -      ; READ HEADER
        READLBLK, -      ; READ LOGICAL BLOCK
        WRITELBLK, -     ; WRITE LOGICAL BLOCK
        READPBLK, -      ; READ PHYSICAL BLOCK
        WRITEPBLK, -     ; WRITE PHYSICAL BLOCK
        READVBLK, -      ; READ VIRTUAL BLOCK
        WRITEVBLK, -     ; WRITE VIRTUAL BLOCK
        AVAILABLE, -     ; AVAILABLE
        ACCESS, -        ; ACCESS FILE / FIND DIRECTORY ENTRY
        ACPCONTROL, -    ; ACP CONTROL FUNCTION
        CREATE, -        ; CREATE FILE AND/OR DIRECTORY ENTRY
        DEACCESS, -      ; DEACCESS FILE
        DELETE, -        ; DELETE FILE AND/OR DIRECTORY ENTRY
        MODIFY, -        ; MODIFY FILE ATTRIBUTES
        MOUNT, -         ; MOUNT VOLUME
        >
FUNCTAB , -                ;BUFFERED FUNCTIONS
        <NOP, -            ; NO-OP

```

Sample Driver for the RL11, RL01, and RL02

```

UNLOAD,- ; UNLOAD
SEEK,- ; SEEK
DRVCLR,- ; DRIVE CLEAR
PACKACK,- ; PACK ACKNOWLEDGE
SENSECHAR,- ; SENSE CHARACTERISTICS
SETCHAR,- ; SET CHARACTERISTICS
SENSEMODE,- ; SENSE MODE
SETMODE,- ; SET MODE
AVAILABLE,- ; AVAILABLE
ACCESS,- ; ACCESS FILE / FIND DIRECTORY ENTRY
ACPCONTROL,- ; ACP CONTROL FUNCTION
CREATE,- ; CREATE FILE AND/OR DIRECTORY ENTRY
DEACCESS,- ; DEACCESS FILE
DELETE,- ; DELETE FILE AND/OR DIRECTORY ENTRY
MODIFY,- ; MODIFY FILE ATTRIBUTES
MOUNT- ; MOUNT VOLUME
>
FUNCTAB DL_ALIGN,- ; TEST ALIGNMENT FUNCTIONS
<READHEAD,- ; READ HEADER
READBLK,- ; READ LOGICAL BLOCK
READPBLK,- ; READ PHYSICAL BLOCK
READVBLK,- ; READ VIRTUAL BLOCK
WRITECHECK,- ; WRITE CHECK
WRITELBK,- ; WRITE LOGICAL BLOCK
WRITEPBLK,- ; WRITE PHYSICAL BLOCK
WRITEVBLK- ; WRITE VIRTUAL BLOCK
>
FUNCTAB +ACP$READBLK,- ; READ FUNCTIONS
<READHEAD,- ; READ HEADER
READBLK,- ; READ LOGICAL BLOCK
READPBLK,- ; READ PHYSICAL BLOCK
READVBLK- ; READ VIRTUAL BLOCK
>
FUNCTAB +ACP$WRITEBLK,- ; WRITE FUNCTIONS
<WRITECHECK,- ; WRITE CHECK
WRITELBK,- ; WRITE LOGICAL BLOCK
WRITEPBLK,- ; WRITE PHYSICAL BLOCK
WRITEVBLK- ; WRITE VIRTUAL BLOCK
>
FUNCTAB +ACP$ACCESS,- ; ACCESS FUNCTIONS
<ACCESS,- ; ACCESS FILE / FIND DIRECTORY ENTRY
CREATE- ; CREATE FILE AND/OR DIRECTORY ENTRY
>
FUNCTAB +ACP$DEACCESS,- ; DEACCESS FUNCTION
<DEACCESS- ; DEACCESS FILE
>
FUNCTAB +ACP$MODIFY,- ; MODIFY FUNCTIONS
<ACPCONTROL,- ; ACP CONTROL FUNCTION
DELETE,- ; DELETE FILE AND/OR DIRECTORY ENTRY
MODIFY- ; MODIFY FILE ATTRIBUTES
>
FUNCTAB +ACP$MOUNT,- ; MOUNT FUNCTION
<MOUNT- ; MOUNT VOLUME
>
FUNCTAB +EXE$LCLDSKVALID,- ; LOCAL DISK VALID FUNCTIONS
<UNLOAD,- ; UNLOAD VOLUME
AVAILABLE,- ; UNIT AVAILABLE
PACKACK- ; PACK ACKNOWLEDGE
>
FUNCTAB +EXE$ZEROPARM,- ; ZERO PARAMETER FUNCTIONS
<NOP,- ; NO-OP
UNLOAD,- ; UNLOAD
DRVCLR,- ; DRIVE CLEAR

```


Sample Driver for the RL11, RL01, and RL02

```

        PACKACK,-          ; PACK ACKNOWLEDGE
        AVAILABLE,-       ; AVAILABLE
    >
    FUNCTAB +EXE$ONEPARM,- ; ONE PARAMETER FUNCTION
    <SEEK-                ; SEEK
    >
    FUNCTAB +EXE$SENSEMODE,- ; SENSE FUNCTIONS
    <SENSECHAR,-          ; SENSE CHARACTERISTICS
    SENSEMODE-           ; SENSE MODE
    >
    FUNCTAB +EXE$SETCHAR,- ; SET FUNCTIONS
    <SETCHAR,-            ; SET CHARACTERISTICS
    SETMODE-              ; SET MODE
    >

.PAGE

.SBTTL CONTROLLER INITIALIZATION ROUTINE
; ++
;
; FUNCTIONAL DESCRIPTION:
;
; THIS ROUTINE IS A NO-OP FOR THE RL11 BUT MUST BE INCLUDED
; SINCE IT IS CALLED WHEN THE RLO2 IS BOOTED AS A SYSTEM DEVICE.
;
; THE OPERATING SYSTEM CALLS THIS ROUTINE:
;   - AT SYSTEM STARTUP
;   - DURING DRIVER LOADING
;   - DURING RECOVERY FROM POWER FAILURE
;
; INPUTS:
;
;   R4   - CSR ADDRESS (DEVICE CONTROL STATUS REGISTER)
;   R5   - IDB ADDRESS (INTERRUPT DATA BLOCK)
;   R6   - DDB ADDRESS (DEVICE DATA BLOCK)
;   R8   - CRB ADDRESS (CHANNEL REQUEST BLOCK)
;   ALL INTERRUPTS ARE LOCKED OUT
;
; OUTPUTS:
;
;   ALL REGISTERS EXCEPT R0-R3 ARE PRESERVED.
;   CONTROL IS RETURNED TO THE CALLER.
;
; --
DL_RL11_INIT:                ;CONTROLLER INITIALIZATION
;
; FOR MICROVAX I, ALLOCATE A PHYSICALLY CONTIGUOUS BUFFER
; AREA FOR PERFORMING I/O.
;
ADPDISP SELECT=ADAP_MAPPING,- ; Allocate a physically contiguous
    ADDRLIST=<<YES,20$>>,- ; buffer for those adapters that
    CRBADDR=R8,-          ; don't support mapping.
    SCRATCH=R0

10$:  MOVZWL #UCB$K_DL_BUFSZ,R1 ;LOAD SIZE OF BUFFER
      JSB   G^EXE$ALOPHYCNTG ;ALLOCATE PHYSICALLY CONTIGUOUS MEMORY
      BLBC  RO,20$           ;EXIT ON ERROR
      MOVL  R2,CRB$L_AUXSTRUC(R8) ;GET BUFFER VIRTUAL ADDRESS
      RSB   ;RETURN TO CALLER

20$:  CLRL  CRB$L_AUXSTRUC(R8) ;INDICATE MEMORY ALLOCATION FAILURE
      RSB   ;RETURN TO CALLER
.PAGE
.SBTTL UNIT INITIALIZATION ROUTINE

```

Sample Driver for the RL11, RL01, and RL02

```

; ++
;
; DL_RLOX_INIT - UNIT INITIALIZATION ROUTINE
;
; FUNCTIONAL DESCRIPTION:
;
;     THIS ROUTINE READIES THE RLO1/RLO2 UNITS FOR I/O OPERATIONS.
;
;     THE OPERATING SYSTEM CALLS THIS ROUTINE:
;     - AT SYSTEM STARTUP
;     - DURING DRIVER LOADING
;     - DURING RECOVERY FROM POWER FAILURE
;
; INPUTS:
;
;     R4 - CSR ADDRESS (CONTROLLER STATUS REGISTER)
;     R5 - UCB ADDRESS (UNIT CONTROL BLOCK)
;
; OUTPUTS:
;
;     THE DRIVE UNIT IS RESET, UCB FIELDS ARE INITIALIZED, AND THE
;     ROUTINE WAITS FOR ONLINE UNITS TO SPIN UP. ALL REGISTERS
;     EXCEPT R0-R3 ARE PRESERVED.
;
; --
DL_RLOX_INIT:                                ;RLO1/RLO2 UNIT INITIALIZATION
MOVW    #1@UCB$V_DL_MAPPING,-                ; DEFAULT TO ADAPTER MAPPING
        UCB$W_DL_FLAGS(R5)                  ; AND 18 BIT ADDRESSING
ADPDISP SELECT=ADAP_MAPPING,-
        ADDRLIST=<<YES,2$>>,-
        UCBADDR=R5,-
        SCRATCH=R0
2$:     CLRW    UCB$W_DL_FLAGS(R5)            ; Clear adapter mapping bit
ADPDISP SELECT=ADDR_BITS,-
        ADDRLIST=<<18,3$>>,-
        ADPADDR=R0
        BISW    #1@UCB$V_DL_22BIT,-         ; FOR MICROVAX II 22-BIT
        UCB$W_DL_FLAGS(R5)                 ; ADDRESSING AS WELL AS ADAPTER MAPPING
3$:
10$:    MOVZWL UCB$W_STS(R5),R3              ;SAVE CURRENT UNIT STATUS
        BICW    #UCB$M_ONLINE!UCB$M_VALID,- ;ASSUME OFFLINE/INVALID
        UCB$W_STS(R5)                      ;...
;
; WAIT FOR CONTROLLER (6 SECONDS MAX) IF CHANNEL IS BUSY WITH ANOTHER UNIT
;
        MOVL    UCB$L_CRB(R5),R0            ;GET CRB ADDRESS
        BBC     #CRB$V_BSY,CRB$B_MASK(R0),20$ ;IF CLEAR - CHANNEL NOT BUSY
        TIMEDWAIT TIME=#600*1000,-         ;6 SECOND WAIT LOOP
        INS1=<TSTB    RL_CS(R4)>,-          ;IS CONTROLLER READY
        INS2=<BLSS    15$>,-              ;IF LSS - YES
        DONELBL=15$                        ;LABEL TO EXIT WAIT LOOP
        BLBC    R0,25$                     ;TIME EXPIRED - EXIT
;
; GET CURRENT DRIVE STATUS AND RESET DRIVE
;

```

Sample Driver for the RL11, RL01, and RL02

```

20$:  MOVW    #RL_DA_M_RST!-          ;PUT RESET AND GET STATUS IN DAR
      RL_DA_M_STS!RL_DA_M_MRK,RL_DA(R4) ;...
      CLRL   R1                      ;CLEAR R1 FOR UNIT NUMBER
      INSV  UCB$W_UNIT(R5),#8,#8,R1 ;GET UNIT NUMBER
      BISW3 R1,#F_GETSTATUS,RL_CS(R4) ;EXECUTE GET STATUS FUNCTION
      BSBW  DL_WAIT                  ;WAIT FOR CONTROLLER
      TSTB  RL_CS(R4)                ;WAS CONTROLLER READY?
      BGEQ  25$                      ;IF GEQ - NO

;
; CLASSIFY DRIVE TYPE
;
      MOVL  #^X2324C001,-
          UCB$L_MEDIA_ID(R5)        ;SET MEDIA IDENT "DL RL01"
      BITW  #RL_MP_M_TYP,RL_MP(R4)   ;IS DRIVE TYPE = RL02?
      BNEQ  30$                      ;IF NEQ - YES
      MOVB  S^#DT$_RL01,-
          UCB$B_DEVTYPE(R5)        ;SET RL01 DEVICE TYPE
      MOVW  #256,UCB$W_CYLINDERS(R5);SET NUMBER OF RL01 CYLINDERS
      MOVZWL #10240,UCB$L_MAXBLOCK(R5) ;SET MAX RL01 BLOCK NUMBER
      BRB   40$

25$:  BRB    70$                      ;BRANCH TO COMMON EXIT

30$:  MOVB  S^#DT$_RL02,-
          UCB$B_DEVTYPE(R5)        ;SET RL02 DEVICE TYPE
      MOVW  #512,UCB$W_CYLINDERS(R5);SET NUMBER OF RL02 CYLINDERS
      MOVZWL #20480,UCB$L_MAXBLOCK(R5) ;SET MAX RL02 BLOCK NUMBER
      INCL  UCB$L_MEDIA_ID(R5)       ;SET MEDIA IDENT "DL RL02"
40$:  BBC   #UCB$V_VALID,R3,60$      ; Branch around wait for drive to spin up
          ; if the drive did NOT have a VALID
          ; volume on it before POWER failure.

;
; INITIALIZE UCB FIELDS AND WAIT FOR ONLINE UNITS TO SPIN UP
;
45$:  BITW  #RL_CS_M_DRDY,RL_CS(R4) ; Is drive ready?
      BNEQ  50$                      ;IF NEQ - YES
      JSB  G^EXE$PWRTIMCHK          ;IS MAX TIME EXCEEDED?
      BLBS R0,45$                   ;IF LBS - NO, STILL MORE TIME NEEDED
      BRB  60$                      ;POWER UP TIME EXCEEDED

50$:  BISW  #UCB$M_VALID,UCB$W_STS(R5) ;SET UCB STATUS VOLUME VALID

```

Sample Driver for the RL11, RL01, and RL02

```

60$:   BBS      #UCB$V_DL_MAPPING, -      ;ADAPTER MAPPING?
        UCB$W_DL_FLAGS(R5),65$         ;IF BS YES
        MOVL   UCB$L_CRB(R5),R1         ;GET CRB ADDRESS
        MOVL   CRB$L_AUXSTRUC(R1),R2    ;MEMORY ALLOC FAILURE DURING CTL INIT?
        BEQL   70$                      ;IF EQL YES, LEAVE OFFLINE
        MOVL   R2,UCB$A_DL_BUF_VA(R5)  ;SAVE BUFFER'S VIRTUAL ADDRESS
        EXTZV  #VA$V_VPN,#VA$$_VPN,R2,R1;GET VIRTUAL PAGE NUMBER OF BUFFER
        MOVL   G^MMG$GL_SPTBASE,RO     ;GET BASE ADDRESS OF SPTS
        MOVL   (RO)[R1],RO              ;GET THE PTE CONTENTS
        BICL3  #^C<VA$M_BYTE>,R2,R1    ;GET BUFFER OFFSET (BA00-BA08)
        ASSUME PTE$$_PFN GE 13
        INSV   RO,#9,#13,R1             ;COPY BAO9-BA21
        MOVL   R1,UCB$A_DL_BUF_PA(R5)  ;SAVE PHYSICAL ADDRESS OF BUFFER
65$:   BISW    #UCB$M_ONLINE,UCB$W_STS(R5) ;SET UCB STATUS VOLUME VALID
70$:   RSB
        .PAGE
        .SBTTL  DRIVER SPECIFIC SUBROUTINES
;
; DL_WAIT - WAIT FOR CONTROLLER READY
;
; INPUTS:
;   R4      - DEVICE CSR ADDRESS
;
; FUNCTIONAL DESCRIPTION:
;
;   THIS ROUTINE IS CALLED FROM THE DRIVER UNIT INITIALIZATION ROUTINE
;   TO WAIT UNTIL THE RL11 CONTROLLER IS READY. TO PREVENT HANGING UP
;   AT HIGH IPL, A MAXIMUM OF 30 USEC ELAPSES BEFORE CONTROL IS
;   RETURNED TO THE CALLER.
;
DL_WAIT:
;   ;WAIT FOR CONTROLLER READY
        MOVQ   RO,-(SP)                  ;SAVE RO, R1
        TIMWAIT #3,#RL_CS_M_CRDY,RL_CS(R4),W
        MOVQ   (SP)+,RO                  ;RESTORE RO, R1
        RSB                                ;RETURN TO UNIT INIT OR STARTIO
        .PAGE
        .SBTTL  FDT ROUTINE - TEST TRANSFER BYTE COUNT ALIGNMENT
;
; ++
;
; DL_ALIGN - FDT ROUTINE TO TEST XFER BYTE COUNT
;
; FUNCTIONAL DESCRIPTION:
;
;   THIS ROUTINE IS CALLED FROM THE FUNCTION DECISION TABLE DISPATCHER
;   TO CHECK THE BYTE COUNT PARAMETER SPECIFIED BY THE USER PROCESS
;   FOR AN EVEN NUMBER OF BYTES (WORD BOUNDARY).
;
; INPUTS:
;
;   R3      - IRP ADDRESS (I/O REQUEST PACKET)
;   R4      - PCB ADDRESS (PROCESS CONTROL BLOCK)
;   R5      - UCB ADDRESS (UNIT CONTROL BLOCK)
;   R6      - CCB ADDRESS (CHANNEL CONTROL BLOCK)
;   R7      - BIT NUMBER OF THE I/O FUNCTION CODE
;   R8      - ADDRESS OF FDT TABLE ENTRY FOR THIS ROUTINE
;   4(AP)   - ADDRESS OF FIRST FUNCTION DEPENDENT QIO PARAMETER
;
; OUTPUTS:
;
;   IF THE QIO BYTE COUNT PARAMETER IS ODD, THE I/O OPERATION IS
;   TERMINATED WITH AN ERROR. IF IT IS EVEN, CONTROL IS RETURNED
;   TO THE FDT DISPATCHER.

```

Sample Driver for the RL11, RL01, and RL02

```

;
;--
DL_ALIGN:                                ;CHECK BYTE COUNT AT P1(AP)
      BLBS      4(AP),10$                ;IF LBS - ODD BYTE COUNT
      RSB                                ;EVEN - RETURN TO CALLER
10$:   MOVZWL   #SS$_IVBUFLN,RO          ;SET BUFFER ALIGNMENT STATUS
      JMP      G^EXE$ABORTIO           ;ABORT I/O
      .PAGE
      .SBTTL   START I/O ROUTINE

;++
;
; DL_STARTIO - START I/O ROUTINE
;
; FUNCTIONAL DESCRIPTION:
;
; THIS FORK PROCESS IS ENTERED FROM THE EXECUTIVE AFTER AN I/O REQUEST
; PACKET HAS BEEN DEQUEUED, AND PERFORMS THE FOLLOWING:
;
; - ACTIVATES THE DISK AFTER SETTING UCB FIELDS, OBTAINING
;   UBA AND CONTROLLER RESOURCES, AND SETTING RL11 REGISTERS
;
; - WAITS FOR AN INTERRUPT
;
; - REGAINS CONTROL AFTER THE ISR SERVICES THE INTERRUPT, AND
;   - REACTIVATES THE DISK IF THE ORIGINAL FUNCTION
;     IS NOT YET COMPLETE, OR
;   - COMPLETES THE I/O REQUEST BY RELEASING RESOURCES,
;     SETTING STATUS CODES, AND RETURNING TO THE EXECUTIVE.
;
; INPUTS:
;
; R3      - IRP ADDRESS (I/O REQUEST PACKET)
; R5      - UCB ADDRESS (UNIT CONTROL BLOCK)
; IRP$L_MEDIA - PARAMETER LONGWORD (LOGICAL BLOCK NUMBER)
;
; OUTPUTS:
;
; R0      - FIRST I/O STATUS LONGWORD: STATUS CODE & BYTES XFERED
; R1      - SECOND I/O STATUS LONGWORD: 0 FOR DISKS
;
; THE I/O FUNCTION IS EXECUTED.
;
; ALL REGISTERS EXCEPT R0-R4 ARE PRESERVED.
;
;--
DL_STARTIO:                                ;START I/O OPERATION
;
; COMPUTE PHYSICAL MEDIA ADDRESS
;
; LBN = LBN * (SECTORS/BLOCK)
; LBN/(SECTORS/TRACK) = D + SECTOR
; D/(TRACKS/CYLINDER) = CYLINDER + TRACK
;
;
; PREPROCESS UCB FIELDS
;

```

Sample Driver for the RL11, RL01, and RL02

```

PREPROCESS:
    MOVL    IRP$L_MEDIA(R3),-      ; Copy given MEDIA address (logical)
           UCB$L_MEDIA(R5)        ; to the UCB.
    BBS     #IRP$V_PHYSIO,-        ; IF SET - PHYSICAL I/O
           IRP$W_STS(R3),10$
    MULL3   #2,UCB$L_MEDIA(R5),R0   ; SCALE LBN IN R0
    MOVZBL  UCB$B_SECTORS(R5),R2    ; GET NUMBER OF SECTORS PER TRACK
    CLRL   R1                       ; CLEAR HIGH PART OF DIVIDEND
    EDIV    R2,R0,R0,UCB$L_MEDIA(R5); CALCULATE SECTOR NUMBER AND STORE
    MOVZBL  UCB$B_TRACKS(R5),R2    ; GET NUMBER OF TRACKS PER CYLINDER
    EDIV    R2,R0,R0,R1            ; CALCULATE TRACK AND CYLINDER
    MOVB    R1,UCB$L_MEDIA+1(R5)   ; STORE TRACK NUMBER
    MOVW    RO,UCB$L_MEDIA+2(R5)   ; STORE CYLINDER NUMBER
10$:
    MOVB    UCB$B_ERTMAX(R5),-      ; INITIALIZE ERROR RETRY COUNT
           UCB$B_ERTCNT(R5)       ; ...
    MNEGW   UCB$W_BCNT(R5),UCB$W_BCR(R5) ; INIT NEG BYTES LEFT TO XFER
    CLRW    UCB$W_DL_DPN(R5)       ; CLEAR DATA PATH NO. FOR USE AS-
                                   ; UBA RESOURCE ALLOCATION FLAG
    CLRB    UCB$B_DL_DPPE(R5)      ; CLEAR DATAPATH PURGE ERROR REGISTER
    MOVW    IRP$W_FUNC(R3),UCB$W_FUNC(R5) ; SAVE FUNCTION CODE
    EXTZV   #IRP$V_FCODE,-         ; EXTRACT I/O FUNCTION CODE
           #IRP$S_FCODE,IRP$W_FUNC(R3),R1 ; ...
    MOVB    R1,UCB$B_FEX(R5)       ; STORE FUNCTION DISPATCH INDEX
    CMPB    #TC$_SEEK,R1           ; SEEK FUNCTION?
    BNEQ    20$                    ; IF NEQ - NO
    MOVW    IRP$L_MEDIA(R3),-      ; STORE CYLINDER ADDRESS
           UCB$W_DC(R5)           ; ...
20$:
    BICW    #UCB$M_DIAGBUF,-       ; CLR DIAGNOSTIC BUFFER PRESENT
           UCB$W_DEVSTS(R5)
    BBC     #IRP$V_DIAGBUF,-       ; IF CLR - NO DIAG BUFFER
           IRP$W_STS(R3),FDISPATCH ; ...
    BISW    #UCB$M_DIAGBUF,UCB$W_DEVSTS(R5) ; SET DIAG BUFFER PRESENT
;
;   CENTRAL FUNCTION DISPATCH
;
FDISPATCH:
           ; FUNCTION DISPATCH
    MOVL    UCB$L_IRP(R5),R3        ; GET IRP ADDRESS
    BBS     #IRP$V_PHYSIO,-        ; IF SET - PHYSICAL I/O FUNCTION
           IRP$W_STS(R3),10$
    BBS     #UCB$V_VALID,-         ; IF SET - VOLUME SOFTWARE VALID
           UCB$W_STS(R5),10$
    MOVZWL  #SS$_VOLINV,RO         ; SET VOLUME INVALID STATUS
    BRW    RESETXFR                ; RESET BYTE COUNT AND EXIT
10$:
    CLRB    UCB$B_DL_DCHEK(R5)     ; CLEAR DATA CHECK IN PROGRESS
    MOVZBL  UCB$B_FEX(R5),R3       ; GET FUNCTION DISPATCH INDEX
    CASE    R3,<-
           UNLOAD,-                ; UNLOAD
           SEEK,-                  ; SEEK
           NOP,-                   ; RECALIBRATE (unsupported)
           DRVCLR,-                ; DRVCLR
           NOP,-                   ; RELEASE PORT (unsupported)
           NOP,-                   ; OFFSET HEADS (unsupported)
           NOP,-                   ; RETURN TO CENTER (unsupported)
           PACKACK,-               ; PACK ACKNOWLEDGE
           NOP,-                   ; SEARCH (unsupported)
           WRITECHECK,-            ; WRITE CHECK
           WRITEDATA,-             ; WRITE DATA
           READDATA,-              ; READ DATA
           NOP,-                   ; WRITE HEADER (unsupported)
           READHEAD,-              ; READ HEADER

```

Sample Driver for the RL11, RL01, and RL02

```

NOP,- ; place holder
NOP,- ; place holder
AVAILABLE- ; AVAILABLE
>,LIMIT=#CDF_UNLOAD ;

NOP: ;NO-OP
SEEK: ;SEEK
DRVCLR: ;DRIVE CLEAR (GET STATUS & RESET)
DO_FUNCTION:
    EXFUNCL RETRYERR ;EXECUTE FUNCTION - RETRY IF FAILURE
    BRB NORMAL ;SUCCESSFUL - EXIT WITH NORMAL STATUS

PACKACK: ;PACK ACKNOWLEDGE (GET STATUS & RESET)
    BISW #UCB$M_VALID, - ;Set software volume valid bit.
        UCB$W_STS(R5)
    BRB DO_FUNCTION ;Then go do hardware function.

UNLOAD: ;UNLOAD
AVAILABLE: ;AVAILABLE
    BICW #UCB$M_VALID, - ;Clear software volume valid bit.
        UCB$W_STS(R5) ;and go complete operation without
    BRB NORMAL ;any hardware interaction.

WRITECHECK: ;WRITE CHECK
READHEAD: ;READ HEADER
    BICW #IO$M_DATACHECK, - ;CLEAR DATA CHECK REQUEST-
        UCB$W_FUNC(R5) ;TO PREVENT EXTRA WRITE CHECK

WRITEDATA: ;WRITE DATA
READDATA: ;READ DATA
    EXFUNCL RETRYERR,F_SEEK ;EXECUTE EXPLICIT SEEK - RETRY IF FAIL
    MOVZBL UCB$B_FEX(R5),R3 ;GET FUNCTION DISPATCH INDEX
    EXFUNCL RETRYERR ;EXECUTE TRANSFER FUNCTION

;
; OPERATON COMPLETION
;

NORMAL: ;SUCCESSFUL OPERATION COMPLETE
    MOVZWL #SS$NORMAL,RO ;SET NORMAL COMPLETION STATUS
    BRW FUNCXT ;FUNCTION EXIT

RETRYERR: ;RETRIABLE ERROR
    DECB UCB$B_ERTCNT(R5) ;ANY RETRIES LEFT?
    BEQL FATALERR ;IF EQL - NO
    BRW FDISPATCH ;RETRY FUNCTION

FATALERR: ;UNRECOVERABLE ERROR
    MOVZWL #SS$VOLINV,RO ;ASSUME VOLUME INVALID STATUS
    BBS #RL_MP_V_VC,- ;IF SET - VOLUME INVALID
        UCB$W_DL_MP(R5),FUNCXT ;...

    MOVZWL #SS$WRITLCK,RO ;ASSUME WRITE LOCK ERROR STATUS
    BBC #RL_MP_V_WL,- ;IF CLR - VOLUME NOT WRITE LOCKED
        UCB$W_DL_MP(R5),5$ ;...
    BBS #RL_MP_V_WGE,- ;IF SET - WRITE GATE ERROR
        UCB$W_DL_MP(R5),FUNCXT ;IF WL & WGE SET - WRITE LOCK ERROR

5$: MOVZWL #SS$DATACHECK,RO ;ASSUME DATA CHECK ERROR STATUS
    TSTB UCB$B_DL_DCHEK(R5) ;WRITE CHECK IN PROGRESS?
    BEQL 10$ ;IF EQL - NO
    BBS #RL_CS_V_OPI,- ;IF SET - NOT WRITE CHECK ERROR
        UCB$W_DL_CS(R5),10$ ;...
    BBS #RL_CS_V_CRC,- ;IF SET - WRITE CHECK ERROR
        UCB$W_DL_CS(R5),FUNCXT ;...

```

Sample Driver for the RL11, RL01, and RL02

```

10$:  MOVZWL  #SS$_PARITY,RO          ;ASSUME PARITY ERROR STATUS
      BBS     #RL_CS_V_CRC,-         ;IF SET - CRC ERROR
      UCB$W_DL_CS(R5),FUNCXT        ;OR DATAPATH PURGE ERROR

20$:  MOVZWL  #SS$_DRVERR,RO         ;ASSUME DRIVE ERROR STATUS
      BBS     #RL_CS_V_DE,-         ;IF SET - DRIVE ERROR
      UCB$W_DL_CS(R5),FUNCXT        ;...

      MOVZWL  #SS$_CTRLERR,RO       ;ASSUME CONTROLLER ERROR STATUS

FUNCXT:                                ;FUNCTION EXIT
      PUSHL  RO                      ;SAVE FINAL REQUEST STATUS
      JSB    G^IOC$DIAGBUFILL        ;FILL DIAGNOSTIC BUFFER IF PRESENT
      CMPB   #CDF_WRITECHECK,UCB$B_FEX(R5) ;DRIVE RELATED FUNCTION?
      BGTRU  10$                     ;IF GTRU - YES
      CMPB   #CDF_AVAILABLE,UCB$B_FEX(R5) ;DRIVE RELATED FUNCTION?
      BEQL  10$                       ;IF EQL - YES
      MOVL  UCB$L_IRP(R5),R3          ;RETRIEVE ADDRESS OF IRP
      ADDW3  UCB$W_BCR(R5),-         ;CALCULATE BYTES TRANSFERRED
      IRP$W_BCNT(R3),2(SP)          ;...
      TSTW   UCB$W_DL_DPN(R5)        ;ARE UBA RESOURCES ALLOCATED?
      BEQL  20$                       ;IF EQL -- NO
      BBC    #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
      UCB$W_DL_FLAGS(R5),10$        ;IF BC NO
      RELDPR                                ;RELEASE DATA PATH
      RELMPR                                ;RELEASE MAP REGISTERS
      BRB    20$                       ;JOIN COMMON CODE
10$:  MOVL   UCB$L_DL_SVAPTE(R5),-    ;RESTORE ORIGINAL SVAPTE
      UCB$L_SVAPTE(R5)              ;
20$:  RELCHAN                            ;RELEASE CHANNEL IF OWNED

      CLRL   R1                        ;CLEAR SECOND STATUS LONGWORD
      POPL  RO                          ;RETRIEVE FINAL REQUEST STATUS
      REQCOM                                ;COMPLETE REQUEST
      .PAGE

;
; FEXL - RL11 HARDWARE FUNCTION EXECUTION
;
; THIS ROUTINE IS CALLED VIA A BSB WITH A BYTE IMMEDIATELY FOLLOWING THAT
; SPECIFIES THE ADDRESS OF AN ERROR ROUTINE. ALL DATA IS ASSUMED TO HAVE BEEN
; SET UP IN THE UCB BEFORE THE CALL. THE APPROPRIATE PARAMETERS ARE LOADED
; INTO DEVICE REGISTERS AND THE FUNCTION IS INITIATED. THE RETURN ADDRESS
; IS STORED IN THE UCB AND A WAIT FOR INTERRUPT IS EXECUTED. WHEN THE
; INTERRUPT OCCURS, CONTROL IS RETURNED TO THE CALLER.
;
; INPUTS:
;
; R3 = FUNCTION TABLE DISPATCH INDEX
; R5 = DEVICE UNIT UCB ADDRESS
;
; 00(SP) = RETURN ADDRESS OF CALLER
; 04(SP) = RETURN ADDRESS OF CALLER'S CALLER
;
; IMMEDIATELY FOLLOWING INLINE AT THE CALL SITE IS A BYTE WHICH CONTAINS
; A BRANCH DESTINATION TO AN ERROR RETRY ROUTINE.
;
; OUTPUTS:
;
; THERE ARE FOUR EXITS FROM THIS ROUTINE:
;
; 1. SPECIAL CONDITION - THIS EXIT IS TAKEN IF A POWER FAILURE OCCURS
; OR THE OPERATION TIMES OUT. IT IS A JUMP TO THE APPROPRIATE
; ERROR ROUTINE.
;
; 2. FATAL ERROR - THIS EXIT IS TAKEN IF A FATAL CONTROLLER OR DRIVE

```


Sample Driver for the RL11, RL01, and RL02

```

;
; ERROR OCCURS OR IF ANY ERROR OCCURS AND ERROR RETRY IS EITHER
; INHIBITED OR EXHAUSTED. IT IS A JUMP TO THE FATAL ERROR EXIT
; ROUTINE.
;
;
; 3. RETRIABLE ERROR - THIS EXIT IS TAKEN IF A RETRIABLE CONTROLLER
; OR DRIVE ERROR OCCURS AND ERROR RETRY IS NEITHER INHIBITED
; NOR EXHAUSTED. IT CONSISTS OF TAKING THE ERROR BRANCH EXIT
; SPECIFIED AT THE CALL SITE.
;
; 4. SUCCESSFUL OPERATION - THIS EXIT IS TAKEN IF NO ERRORS OCCUR
; DURING THE OPERATION. IT CONSISTS OF A RETURN INLINE.
;
; IN ALL CASES IF AN ERROR OCCURS, AN ATTEMPT IS MADE TO LOG THE ERROR.
;
; IN ALL CASES FINAL DEVICE REGISTERS ARE RETURNED VIA THE UCB.
;
; UCB$W_BCR(R5) = NEGATIVE BYTES REMAINING TO TRANSFER
; .PAGE
FEXL:
;FUNCTION EXECUTOR
POPL UCB$L_DPC(R5) ;SAVE DRIVER PC VALUE
MOVB R3,UCB$B_CEX(R5) ;SAVE CASE INDEX
MOVL UCB$L_CRB(R5),R0 ;GET ADDRESS OF PRIMARY CRB
MOVL CRB$L_INTD+VEC$L_IDB(R0),R1 ;GET ADDRESS OF IDB
CMLP R5,IDB$L_OWNER(R1) ;DOES THIS PROCESS OWN CHANNEL?
BNEQ 10$ ;IF NEQ - NO
MOVL IDB$L_CSR(R1),R4 ;SET ASSIGNED CHANNEL CSR ADDRESS
BRB 20$ ;
10$: REQPCCHAN ;REQUEST CHANNEL (RETURNS R4 = CSR ADR)
20$: CASE R3,<- ;DISPATCH TO PROPER FUNCTION ROUTINE
IMMED,- ;NO OPERATION
IMMED,- ;UNLOAD VOLUME (NOP)
POSIT,- ;SEEK CYLINDER
IMMED,- ;RECALIBRATE (NOP)
DRCLR,- ;DRIVE CLEAR (GET STATUS & RESET)
IMMED,- ;RELEASE DRIVE (NOP)
IMMED,- ;OFFSET HEADS (NOP)
IMMED,- ;RETURN TO CENTERLINE (NOP)
DRCLR,- ;PACK ACKNOWLEDGE
IMMED,- ;SEARCH (NOP)
> ;
BRW XFER ;TRANSFER FUNCTION
.PAGE
; IMMEDIATE FUNCTION EXECUTION
;
; FUNCTIONS INCLUDE:
;
; NO OPERATION,
; DRIVE CLEAR, AND
; PACK ACKNOWLEDGE
;
; INPUTS:
; R3 - CASE INDEX
; R4 - CSR ADDRESS
; R5 - UCB ADDRESS
;
; FUNCTIONAL DESCRIPTION:
;
; INTERRUPTS ARE LOCKED OUT, THE APPROPRIATE FUNCTION IS INITIATED WITH
; INTERRUPT ENABLE, AND A WAIT FOR INTERRUPT AND KEEP CHANNEL IS EXECUTED.
;

```

Sample Driver for the RL11, RL01, and RL02

```

DRCLR:                                     ;DRIVE CLEAR
      BISW  #RL_DA_M_STS!-                 ;SET GETSTATUS,RESET,AND MARK IN DAR
      RL_DA_M_RST!RL_DA_M_MRK,RL_DA(R4) ;...

IMMED:                                     ;IMMEDIATE FUNCTION EXECUTION
      CKPWR  SAVE_R0=NO                    ;DISABLE INTERRUPTS, CHECK POWER,-
      BISW3  R2,FTAB[R3],RL_CS(R4)        ;AND PUT UNIT NUMBER IN R2<9:8>
      WFIKPC RETREG,#2                     ;MERGE UNIT WITH FNTN AND EXECUTE
      IOFORK                                     ;WAIT FOR INTERRUPT
      BRW    RETREG                         ;RETURN FROM ISR-
      .PAGE                                  ;CREATE FORK PROCESS (&JSB BACK TO ISR)
;
; POSITIONING FUNCTION EXECUTION
;
;   FUNCTIONS INCLUDE:
;
;       SEEK CYLINDER
;
; INPUTS:
;   R3    - CASE INDEX
;   R4    - DEVICE CSR ADDRESS
;   R5    - UCB ADDRESS
;
; FUNCTIONAL DESCRIPTION:
;
; THE CYLINDER DIFFERENCE WORD IS CALCULATED AND LOADED INTO THE DISK
; ADDRESS REGISTER, INTERRUPTS ARE LOCKED OUT, AND THE SEEK FUNCTION
; IS INITIATED WITHOUT INTERRUPT ENABLE. THE CONTROLLER IS THEN POLLED
; FOR READY, AND DEVICE INTERRUPTS ARE ENABLED.
;
; SINCE THE RLO1/RLO2 DO NOT ISSUE AN INTERRUPT UPON COMPLETION OF A
; SEEK, OVERLAPPED SEEKS ARE NOT ATTEMPTED, AND ONE OF THE FOLLOWING IS
; PERFORMED.
;
;   IF ONLY A SEEK FUNCTION IS BEING REQUESTED, A DUMMY READ HEADER
;   FUNCTION IS ISSUED AND A WAITFOR INTERRUPT IS INITIATED.
;   THE READ HEADER IS USED TO SIGNAL THE END OF THE SEEK, SINCE IT
;   WILL ISSUE AN INTERRUPT SHORTLY (315 USEC AVG) AFTER THE SEEK IS
;   COMPLETE. IT WILL ALSO SENSE FOR A TIMEOUT DURING THE SEEK.
;
;   IF THE SEEK IS ASSOCIATED WITH A DATA TRANSFER REQUEST (RLO1/RLO2
;   TRANSFER FUNCTIONS REQUIRE EXPLICIT SEEKS), THE PROGRAM KEEPS THE
;   CHANNEL AND RETURNS TO FDISPATCH TO ISSUE THE TRANSFER REQUEST
;   WHILE THE SEEK IS STILL IN PROGRESS. WHEN THE SEEK COMPLETES, THE
;   RL11 CONTROLLER WILL BEGIN THE TRANSFER.
;
;

```

Sample Driver for the RL11, RL01, and RL02

```

POSIT:                                ;POSITIONING FUNCTION
;
; OBTAIN CURRENT DISK ADDRESS
;
; IF THERE HAS NOT BEEN A PREVIOUS TRANSFER DURING THIS REQUEST,
; A READ HEADER IS EXECUTED TO DETERMINE THE CURRENT DISK ADDRESS.
;
TSTW   UCB$W_DL_DPN(R5)                ;WAS THERE A PREVIOUS TRANSFER?
BEQL   10$                              ;IF EQL - NO, READ HEADER
BICW3  #^077,UCB$W_DL_DA(R5),R1        ;PUT CURRENT CYL & SURFACE IN R1
BRW    60$                              ;CALCULATE DIFFERENCE WORD
5$:    BRW    50$                        ;CONTINUE
10$:   MOVZBL #8,R3                      ;SET READ HEADER RETRY COUNT IN R3
20$:   CKPWR  SAVE_RO=NO                 ;DISABLE INTERRUPTS, CHECK POWER,-
;AND PUT UNIT NUMBER IN R2<9:8>
BISW3  R2,#F_READHEAD!RL_CS_M_IE,-     ;EXECUTE READ HEADER
RL_CS(R4)                               ;...
WFIKPCH 40$,#2                          ;WAIT FOR INTERRUPT OR TIMEOUT
IOFORK   ;CREATE FORK PROCESS
BBC      #RL_CS_V_CE,UCB$W_DL_CS(R5),5$ ;BR ON NO ERRORS
DECB    R3                              ;DECREMENT READ HEADER RETRY COUNT
BNEQ    20$                              ;IF NEQ - RETRY READ HEADER
;IF EQL - READ HEADER RETRY EXHAUSTED -
;TRY PREVIOUS TRACK
MOVZBW  #^X80!RL_DA_M_MRK,-            ;LOAD REVERSE SEEK DIFFERENCE WORD
RL_DA(R4)                               ;...
CKPWR   SAVE_RO=NO                      ;DISABLE INTERRUPTS, CHECK POWER,-
;AND PUT UNIT NUMBER IN R2<9:8>
BISW3  R2,#F_SEEK!RL_CS_M_IE,-         ;EXECUTE REVERSE SEEK
RL_CS(R4)                               ;...
WFIKPCH 40$,#2                          ;WAIT FOR SEEK TO BEGIN (INTERRUPT)
IOFORK   ;CREATE FORK PROCESS
CKPWR   SAVE_RO=NO                      ;DISABLE INTERRUPTS, CHECK POWER,-
;AND PUT UNIT NUMBER IN R2<9:8>
BISW3  R2,#F_READHEAD!RL_CS_M_IE,-     ;TRY READ HEADER ON NEW TRACK
RL_CS(R4)                               ;...
WFIKPCH 40$,#2                          ;WAITFOR INTERRUPT OR TIMEOUT
IOFORK   ;CREATE FORK PROCESS
BBC      #RL_CS_V_CE,UCB$W_DL_CS(R5),50$ ;BR IF NO HEADER ERROR
40$:   ;CANNOT READ CURRENT DISK ADDRESS
;
CLRB    UCB$B_ERTCNT(R5)                ;CLEAR RETRY COUNT
BRW     RETREG                          ;
50$:   ;FOUND CURRENT DISK ADDRESS
BICW3  #^077,UCB$W_DL_MP(R5),R1        ;PUT CURRENT CYL & SURFACE IN R1
;
; CALCULATE CYLINDER DIFFERENCE WORD
;
60$:   CLRL   RO                        ;CLEAR RO FOR DESIRED ADDRESS
INSV   UCB$W_DA+1(R5),#6,#1,RO         ;INSERT DESIRED SURFACE IN RO<6>
INSV   UCB$W_DC(R5),#7,#9,RO          ;INSERT DESIRED CYLINDER IN RO<15:7>
CMPW   RO,R1                            ;IS A SEEK NEEDED?
BEQL   80$                              ;IF EQL - NO
BICB   #^0177,R1                        ;REMOVE SURFACE BIT
BICB   #^0177,RO                        ;REMOVE SURFACE BIT
SUBW   RO,R1                            ;SUBTRACT DESIRED FROM ACTUAL
BEQL   70$                              ;IF EQL - ONLY CHANGE SURFACE
BCC    70$                              ;IF CC - ACTUAL>=DESIRED
MNEGW  R1,R1                            ;ACTUAL<DESIRED, MAKE POSITIVE DIFF
BISW   #4,R1                            ;SET SIGN FOR MOVE TO CENTER OF DISK
70$:   INSV   UCB$W_DA+1(R5),#4,#1,R1  ;INSERT SURFACE BIT
BISW3  #RL_DA_M_MRK,R1,RL_DA(R4)      ;SET MARKER AND LOAD DIFFERENCE WORD

```

Sample Driver for the RL11, RL01, and RL02

```
;
; EXECUTE SEEK
;
      CKPWR   SAVE_RO=NO                ;DISABLE INTERRUPTS, CHECK POWER,-
      ;AND PUT UNIT NUMBER IN R2<9:8>
      BISW3   R2,#F_SEEK!RL_CS_M_IE,-  ;EXECUTE SEEK FUNCTION
      RL_CS(R4)
      ;...
      WFIKPCH 40$,#2                    ;WAIT FOR SEEK TO BEGIN (INTERRUPT)
      IOFORK
      ;CREATE FORK PROCESS
80$:   CMPB   #IO$_SEEK,UCB$_FEX(R5)    ;IS SEEK ASSOCIATED WITH A TRANSFER?
      BEQL   90$                        ;IF EQL - NO, SEEK ONLY
;
; RETURN FOR SEEK ASSOCIATED WITH A TRANSFER REQUEST
;
      INCL   UCB$_L_DPC(R5)             ;ADJUST TO CORRECT RETURN ADDRESS
      JMP    @UCB$_L_DPC(R5)           ;RETURN TO DRIVER FOR TRANSFER
;
; RETURN FOR SEEK ONLY REQUEST
;
90$:   CKPWR   SAVE_RO=NO                ;DISABLE INTERRUPTS, CHECK POWER,-
      ;AND PUT UNIT NUMBER IN R2<9:8>
      BISW3   R2,#F_READHEAD!RL_CS_M_IE,- ;EXECUTE DUMMY READ HEADER
      RL_CS(R4)
      ;...
      WFIKPCH RETREG,#2                 ;WAIT FOR SEEK TO COMPLETE (INTERRUPT)
      IOFORK
      ;CREATE FORK PROCESS
      BRW    RETREG
      .PAGE
;
; TRANSFER FUNCTION EXECUTION
;
      FUNCTIONS INCLUDE:
;
;         WRITE CHECK
;         WRITE DATA
;         READ DATA, AND
;         READ HEADER
;
; INPUTS:
;         R3   - CASE INDEX
;         R4   - DEVICE CSR ADDRESS
;         R5   - UCB ADDRESS
;
; FUNCTIONAL DESCRIPTION:
;
; A UNIBUS DATAPATH IS REQUESTED FOLLOWED BY THE APPROPRIATE NUMBER OF MAP
; REGISTERS REQUIRED FOR THE TRANSFER. THE TRANSFER PARAMETERS ARE LOADED
; INTO THE DEVICE REGISTERS, INTERRUPTS ARE LOCKED OUT, THE FUNCTION IS
; INITIATED, AND A WAITFOR INTERRUPT AND KEEP CHANNEL IS EXECUTED.
;
; UPON RETURN FROM THE INTERRUPT SERVICE ROUTINE, IF THE TRANSFER IS
; COMPLETE, THE APPROPRIATE EXIT IS TAKEN. IF THE FUNCTION IS NOT COMPLETE
; TRANSFER PARAMETERS ARE UPDATED AND A RETURN TO FDISPATCH IS EXECUTED TO
; REISSUE SEEK AND TRANSFER FUNCTIONS WHILE KEEPING CHANNEL AND UBA
; RESOURCES. IF A DATA CHECK HAS BEEN REQUESTED, IT IS PERFORMED
; BEFORE RETURNING TO FDISPATCH.
;
```

Sample Driver for the RL11, RL01, and RL02

```

XFER:                                ;TRANSFER FUNCTION EXECUTION
      BBS      #UCB$V_DL_MAPPING, -   ;ADAPTER MAPPING?
      UCB$W_DL_FLAGS(R5), 2$         ;BRANCH IF ADAPTER MAPPING.
      MOVW    UCB$A_DL_BUF_PA(R5), UCB$W_DL_SBA(R5);GET 1ST WORD OF BUFFER ADDR
      MOVZWL  UCB$A_DL_BUF_PA+2(R5), RO;GET BITS 16:21 OF BUFFER ADDRESS
      MOVW    RO, RL_BAE(R4)         ;SET MEMORY EXTENSION BITS IN BAE
      ASHL    #4, RO, RO             ;PUT MEMORY EXTENSION BITS IN <5:4>
      MOVB    RO, UCB$B_DL_XBA(R5)   ;OF CSR
;
; FIRST TRANSFER OF THIS I/O REQUEST - ALLOCATE RESOURCES
;
      TSTW    UCB$W_DL_DPN(R5)       ;RESOURCES ALREADY ALLOCATED?
      BNEQ    5$                     ;IF NEQ - YES
      CLRL    UCB$A_DL_MOVRTN(R5)    ;ASSUME READ
      CMPB    #CDF_WRITEDATA, R3     ;WRITE DATA?
      BNEQ    1$                     ;IF NEQ NO
      MOVAB   G^IOC$MOVFRUSER, -     ;SET MOVE ROUTINE ADDRESS FOR
      UCB$A_DL_MOVRTN(R5)           ;1ST PARTIAL WRITE
1$:    MOVL    UCB$L_SVAPTE(R5), UCB$L_DL_SVAPTE(R5);SAVE SVAPTE FOR BUFFER COPY
      MNEGW   #1, UCB$W_DL_DPN(R5)   ;SET FIRST XFER FLAG
      BRB     5$                     ;JOIN COMMON CODE
;
; FIRST TRANSFER OF THIS I/O REQUEST - ALLOCATE RESOURCES
;
2$:    TSTW    UCB$W_DL_DPN(R5)       ;UBA RESOURCES ALREADY ALLOCATED?
      BNEQ    5$                     ;IF NEQ - YES
      REQDPR                      ;REQUEST DATAPATH
      REQMPR                      ;REQUEST MAP REGISTERS
      LOADUBA                       ;LOAD UNIBUS MAP REGISTERS
      MOVL    UCB$L_CRB(R5), R1       ;GET CRB ADDRESS
      EXTZV   #VEC$V_DATAPATH, #VEC$S_DATAPATH, - ;EXTRACT DATAPATH NUMBER -
      CRB$L_INTD+VEC$B_DATAPATH(R1), RO ;FOR UBA RESOURCE FLAG
      MOVW    RO, UCB$W_DL_DPN(R5)   ;INDICATE UBA RESOURCES ALLOCATED
;
      MOVZWL  UCB$W_BOFF(R5), RO     ;GET BYTE OFFSET IN PAGE
      INSV    CRB$L_INTD+VEC$W_MAPREG(R1), - ;INSERT HIGH 7 BITS OF ADDRESS
      #9, #7, RO                     ;...
      MOVW    RO, UCB$W_DL_SBA(R5)   ;SET BUFFER ADDRESS
      EXTZV   #7, #2, CRB$L_INTD+VEC$W_MAPREG(R1), RO ;GET MEMORY EXTENSION BITS
      MULB3   #16, RO, UCB$B_DL_XBA(R5) ;POSITION MEMORY EXTENSION BITS TO <5:4>
;
; COMMON TRANSFER POINT
;
; FOR A READ OPERATION WHEN NO ADAPTER MAPPING IS PRESENT EMPTY THE
; INTERNAL PHYSICALLY CONTIGUOUS BUFFER FROM THE PREVIOUS READ TO THE
; USER'S BUFFER.
;
5$:    BSBW    DL_MOVE_TO_BUFFER     ;COPY TO USER BUFFER
;
; PUT BUFFER ADDRESS, WORD COUNT, AND DISK ADDRESS IN DEVICE REGISTERS
;

```

Sample Driver for the RL11, RL01, and RL02

```

MOVW   UCB$W_DL_SBA(R5),RL_BA(R4) ;SET BUFFER ADDRESS
MNEGW  UCB$W_BCR(R5),-           ;GET BYTES LEFT TO TRANSFER AND -
      UCB$W_DL_PBCR(R5)         ;ASSUME ONLY ONE TRANSFER NEEDED
MOVZBL UCB$B_SECTORS(R5),R2      ;GET SECTORS/SURFACE
MOVZBL UCB$W_DA(R5),R1          ;GET DESIRED SECTOR
SUBW   R1,R2                    ;CALCULATE SECTORS LEFT ON SURFACE
MULW   #256,R2                  ;CONVERT TO BYTES LEFT ON SURFACE
CMPW   UCB$W_DL_PBCR(R5),R2     ;ARE ADDITIONAL TRANSFERS REQUIRED?
BLEQU  10$                      ;IF LEQU - NO
MOVW   R2,UCB$W_DL_PBCR(R5)     ;SET BYTE COUNT FOR THIS TRANSFER
;
; FOR A WRITE OPERATION WHEN NO ADAPTER MAPPING IS PRESENT
; FILL INTERNAL PHYSICALLY CONTIGUOUS BUFFER FROM THE USER'S BUFFER.
;
10$:   BSBW   DL_MOVE_FROM_BUFFER ;COPY FROM USER BUFFER
      MOVZBL UCB$B_DL_XBA(R5),R0  ;SET MEMORY EXTENSION BITS
      BISW   FTAB[R3],R0         ;MERGE XBA BITS WITH FUNCTION
      DIVW3  #2,UCB$W_DL_PBCR(R5),R2 ;CALCULATE TRANSFER WORD COUNT
      MNEGW  R2,RL_MP(R4)        ;SET TRANSFER WORD COUNT
      MOVZBL UCB$W_DA(R5),R1     ;PUT DESIRED SECTOR IN R1<5:0>
      INSV  UCB$W_DA+1(R5),#6,#1,R1 ;INSERT DESIRED SURFACE IN R1<6>
      INSV  UCB$W_DC(R5),#7,#9,R1 ;INSERT DESIRED CYLINDER IN R1<15:7>
      MOVW   R1,RL_DA(R4)       ;SET DESIRED DISK ADDRESS
;
; EXECUTE THE TRANSFER FUNCTION
;
      CKPWR ;DISABLE INTERRUPTS, CHECK POWER,-
      ;AND PUT UNIT NUMBER IN R2<9:8>
      BISW3  R2,R0,RL_CS(R4)    ;EXECUTE FUNCTION
      WFIKPCH RETREG,#6        ;WAITFOR INTERRUPT AND KEEP CHANNEL
      IOFORK ;RETURN HERE FROM ISR SAVING REGISTERS
      ;CREATE FORK PROCESS (RETURN TO ISR)
      ;RETURN HERE FROM ISR REI ROUTINE
;
; PURGE DATAPATH
;
      CLRB  UCB$B_DL_DPPE(R5)   ;CLEAR DATAPATH PURGE ERROR
      JSB  G^IOC$PURGDATAP     ;PURGE DATAPATH
      BLBS R0,20$              ;IF SET - NO PURGE ERRORS
      INCB UCB$B_DL_DPPE(R5)   ;SET DATAPATH PURGE ERROR
;
; SAVE UBA REGISTERS FOR UPDATE AND REGDUMP ROUTINES
;

```

Sample Driver for the RL11, RL01, and RL02

```

20$:   BBC      #UCB$V_DL_MAPPING, -      ;ADAPTER MAPPING?
        UCB$W_DL_FLAGS(R5), 30$      ;IF BC NO
        MOVL   R1,UCB$L_DL_DPR(R5)      ;SAVE DATAPATH REGISTER
        EXTZV  #9,#7,UCB$W_DL_BA(R5),R0 ;EXTRACT LOW BITS OF FINAL MAP REG NO.
        EXTZV  #4,#2,UCB$W_DL_CS(R5),R1 ;EXTRACT HI BITS OF FINAL MAP REG NO.
        INSV   R1,#7,#2,R0             ;INSERT HIGH BITS OF FINAL MAP REGISTER
        CMPW   #495,R0                 ;LEGAL MAP REGISTER NUMBER?
        BGEQ   25$                     ;IF GEQ - YES
        MOVZWL #495,R0                 ;RESTRICT MAP REGISTER NUMBER
25$:   MOVL   (R2)[R0],UCB$L_DL_FMPR(R5) ;SAVE FINAL MAP REGISTER NUMBER
        CLRL  UCB$L_DL_PMPR(R5)        ;CLEAR PREVIOUS MAP REGISTER CONTENTS
        DECL  R0                       ;CALCULATE PREVIOUS MAP REGISTER NUMBER
        CMPV  #VEC$V_MAPREG,#VEC$$_MAPREG,- ;ANY PREVIOUS MAP REGISTER?
        CRB$L_INTD+VEC$W_MAPREG(R3),R0 ;...
        BGTR  30$                      ;IF GTR - NO
        MOVL  (R2)[R0],UCB$L_DL_PMPR(R5) ;SAVE PREVIOUS MAP REGISTER
30$:   BBC      #RL_CS_V_CE,UCB$W_DL_CS(R5),40$ ;IF CLR - NO RL ERRORS
        BRW   RETREG                   ;DEVICE ERROR
40$:   BLBC   UCB$B_DL_DPPE(R5),45$    ;IF CLR - NO PURGE ERROR
        BRW   RETREG                   ;PURGE ERROR

;
; RETURN HEADER INFORMATION FOR READ HEADER FUNCTION
;
45$:   CMPB   #CDF_READHEAD,UCB$B_CEX(R5) ;READ HEADER FUNCTION?
        BNEQ  DATACHECK                ;IF NEQ - NO
        PUSHL UCB$W_BCR(R5)            ;SAVE NEG BYTES REMAINING
        PUSHL UCB$L_SVAPTE(R5)         ;SAVE ADDRESS OF PTE
        MOVAB UCB$W_DL_DB(R5),R1       ;SET ADDRESS OF INTERNAL BUFFER
        MOVL  #6,R2                    ;SET NUMBER OF BYTES TO MOVE
        CMPW  R2,UCB$W_BCNT(R5)        ;ROOM FOR FULL HEADER?
        BLSSU 50$                      ;IF LSSU - YES
        MOVZWL UCB$W_BCNT(R5),R2       ;SET LENGTH OF PARTIAL HEADER
50$:   SUBW3  UCB$W_BCNT(R5),R2,UCB$W_BCR(R5) ;CALCULATE TRANSFER BYTE COUNT
        JSB  G^IOC$MOVTOUSER           ;MOVE HEADER TO USER BUFFER
        POPL UCB$L_SVAPTE(R5)         ;RESTORE ADDRESS OF PTE
        POPL UCB$W_BCR(R5)            ;RESTORE NEG BYTES REMAINING

;
; PERFORM DATA CHECK, IF REQUESTED
;
DATACHECK:
        BBC      #IO$V_DATACHECK, -      ;DATACHECK AFTER PARTIAL TRANSFER
        UCB$W_FUNC(R5),UPDATE          ;IF CLR - DATA CHECK NOT REQUESTED
        BBSC    #0,UCB$B_DL_DCHEK(R5),- ;IF SET - DATA CHECK ALREADY PERFORMED
        UPDATE  ;...
        INCB   UCB$B_DL_DCHEK(R5)       ;SET DATA CHECK IN PROGRESS
        MOVZBL #IO$_WRITECHECK,R3      ;SET CASE INDEX TO WRITE CHECK
        BRW   XFER                      ;BRANCH TO PERFORM WRITE CHECK

;
; UPDATE BUFFER ADDRESS, CURRENT DISK ADDRESS, AND BYTES REMAINING
; FOR NEXT TRANSFER
;
UPDATE:
        BBC      #UCB$V_DL_MAPPING, -      ;UPDATE TRANSFER PARAMETERS
        UCB$W_DL_FLAGS(R5),10$        ;ADAPTER MAPPING?
        BICB3  #^XCF,UCB$W_DL_CS(R5),-   ;IF BC NO
        UCB$B_DL_XBA(R5)              ;SAVE MEMORY EXTENSION BITS
        MOVW  UCB$W_DL_BA(R5),-        ;...
        UCB$W_DL_SBA(R5)              ;UPDATE SAVED BUFFER ADDRESS
        ;...

```

Sample Driver for the RL11, RL01, and RL02

```

10$:   CLRB      UCB$W_DA(R5)           ;UPDATE DESIRED SECTOR TO ZERO
      ADDL3    #^0100,UCB$W_DL_DA(R5),R1 ;INCREMENT CYLINDER & SURFACE
      EXTZV    #6,#1,R1,R2           ;EXTRACT DESIRED DISK SURFACE
      MOV      R2,UCB$W_DA+1(R5)      ;UPDATE DESIRED DISK SURFACE
      EXTZV    #7,#9,R1,R2           ;EXTRACT DESIRED DISK CYLINDER
      MOVW     R2,UCB$W_DC(R5)        ;UPDATE DESIRED DISK CYLINDER
      ADDW     UCB$W_DL_PBCR(R5),-    ;UPDATE NEG BYTES REMAINING TO XFER
      UCB$W_BCR(R5)                   ;...
      BEQL     RETREG                 ;IF EQL - TRANSFER COMPLETE
      BRW      FDISPATCH             ;MORE BYTES REMAINING - CONTINUE

;
; GET STATUS AND RESET ERRORS
;
;
RETREG:                                ;GET STATUS AND RESET ERRORS
;
; FOR A READ OPERATION WHEN NO ADAPTER MAPPING IS PRESENT
; EMPTY INTERNAL BUFFER INTO USER'S BUFFER FOR LAST READ
;
;
      BSBW     DL_MOVE_TO_BUFFER      ;MOVE LAST READ INTO USER'S BUFFER
      BITW     #UCB$M_TIMEOUT!UCB$M_POWER,- ; TIMEOUT OR POWERFAIL?
      UCB$L_STS(R5)                   ;
      BEQL     0$                     ;BR IF NO
      IOFORK                                ;ELSE, FORK
0$:   MOVW     #RL_DA_M_STS!-          ;PUT GET STATUS IN DAR
      RL_DA_M_MRK,RL_DA(R4)           ;...
      CLRL     R2                     ;CLEAR R2 FOR UNIT NUMBER
      INSV     UCB$W_UNIT(R5),#8,#8,R2 ;GET UNIT NUMBER
      BISW3    R2,#F_GETSTATUS,RL_CS(R4) ;EXECUTE GET STATUS
      DEVICELOCK -
      LOCKADDR=UCB$L_DLCK(R5),- ;LOCK DEVICE ACCESS
      LOCKIPL=UCB$B_DIPL(R5),- ;RAISE IPL
      SAVIPL=- (SP),- ;SAVE CURRENT IPL
      PRESERVE=NO ;DON'T PRESERVE RO
      BSBW     DL_WAIT                 ;WAIT FOR CONTROLLER
      MOVW     RL_MP(R4),UCB$W_DL_MP(R5) ;RETRIEVE ERROR REGISTER
      MOVW     #RL_DA_M_RST!-         ;PUT GET STATUS & RESET IN DAR
      RL_DA_M_STS!RL_DA_M_MRK,RL_DA(R4) ;...
      BISW3    R2,#F_GETSTATUS,RL_CS(R4) ;EXECUTE RESET
      BSBW     DL_WAIT                 ;WAIT FOR CONTROLLER
      DEVICEUNLOCK -
      LOCKADDR=UCB$L_DLCK(R5),- ;UNLOCK DEVICE ACCESS
      NEWIPL=(SP)+,- ;RESTORE IPL
      PRESERVE=NO ;DON'T PRESERVE RO

;
; DETERMINE EXIT - SPECIAL CONDITION, FATAL ERROR, RETRIABLE ERROR, OR SUCCESS
;
;
      CMPZV    #0,#5,UCB$W_DL_MP(R5),- ;HEADS, BRUSHES, STATE OK?
      #RL_MP_M_BH!RL_MP_M_HO!RL_SLM ;...
      BEQL     1$                     ;IF EQL - YES, ONLINE
      BICW     #UCB$M_TIMEOUT,UCB$W_STS(R5) ;CLEAR DEVICE TIME OUT
      MOVZWL   #SS$_MEDOFL,RO         ;SET MEDIUM OFFLINE STATUS
      BRW      FUNCXT                 ;RETURN
1$:   BITW     #UCB$M_POWER!-         ;POWER FAIL OR DEVICE TIMEOUT?
      UCB$M_TIMEOUT,UCB$W_STS(R5) ;...
      BNEQ     SPECOND                ;IF NEQ - YES, SPECIAL CONDITION

```


Sample Driver for the RL11, RL01, and RL02

```

BBS      #RL_MP_V_VC,UCB$W_DL_MP(R5),20$ ;IF SET - VOLUME INVALID
BBS      #RL_CS_V_CE,UCB$W_DL_CS(R5),2$  ;IF SET - RL ERROR
BLBC     UCB$B_DL_DPPE(R5),10$           ;IF CLR - NO PURGE ERROR
2$:      JSB      G^ERL$DEVICERR          ;ALLOCATE AND FILL ERROR MESSAGE BUFFER
BBS      #IO$V_INHRETRY,UCB$W_FUNC(R5),20$ ;IF SET - RETRY INHIBITED
BBS      #RL_CS_V_NXM,UCB$W_DL_CS(R5),20$ ;IF SET - NONEXISTENT MEMORY
BBC      #RL_CS_V_DE,UCB$W_DL_CS(R5),5$  ;IF CLR - NO DRIVE ERRORS
BBC      #RL_MP_V_WL,UCB$W_DL_MP(R5),4$  ;IF CLR - NOT WRITE LOCKED
BBS      #RL_MP_V_WGE,UCB$W_DL_MP(R5),20$ ;IF WL & WGE SET - WL ERROR
4$:      BITW     #RL_MP_M_WDE!-          ;WRITE DATA ERROR, OR
          RL_MP_M_CHE!-                  ;CURRENT HEAD ERROR, OR
          RL_MP_M_WGE!-                  ;WRITE GATE ERROR, OR
          RL_MP_M_DSE,UCB$W_DL_MP(R5)    ;DRIVE SELECT ERROR?
BNEQ     20$                               ;IF NEQ - YES

;
; RETRIABLE ERROR EXIT
;
5$:      CVTBL   @UCB$L_DPC(R5),-(SP)      ;GET BRANCH DISPLACEMENT
          ADDL   (SP)+,UCB$L_DPC(R5)      ;CALCULATE RETURN ADDRESS - 1

;
; SUCCESSFUL OPERATION EXIT
;
10$:     INCL   UCB$L_DPC(R5)              ;ADJUST TO CORRECT RETURN ADDRESS
          JMP   @UCB$L_DPC(R5)            ;RETURN TO DRIVER

;
; FATAL ERROR EXIT
;
20$:     BRW    FATALERR                  ;FATAL ERROR EXIT

;
; SPECIAL CONDITION EXIT (POWER FAILURE OR DEVICE TIMEOUT)
;
SPECOND:
BBS      #UCB$V_POWER,UCB$W_STS(R5),PWRFAIL ;IF SET - POWER FAILURE
          ;IF CLR - DEVICE TIMEOUT
JSB      G^ERL$DEVICTMO                  ;LOG DEVICE TIMEOUT
BICW     #UCB$M_TIMEOUT,UCB$W_STS(R5)    ;CLEAR TIMEOUT STATUS
MOVZWL   #SS$_TIMEOUT,R0                 ;SET DEVICE TIMEOUT STATUS
DECB     UCB$B_ERTCNT(R5)                ;ANY ERROR RETRIES REMAINING?
BEQL     RESETXFR                        ;IF EQL - NO
BRW      FDISPATCH                      ;RETURN

RESETXFR:
          ;RESET TRANSFER BYTE COUNT
MOVL     UCB$L_IRP(R5),R3                 ;GET ADDRESS OF I/O PACKET
MNEGW    IRP$W_BCNT(R3),UCB$W_BCR(R5)    ;RESET BYTE COUNT
BRW      FUNCXT                          ;EXIT

```

Sample Driver for the RL11, RL01, and RL02

```

PWRFAIL:                                     ;POWER FAILURE
      BICW   #UCB$M_POWER,UCB$W_STS(R5)   ;CLEAR POWER FAILURE BIT
      TSTW   UCB$W_DL_DPN(R5)             ;ARE UCB RESOURCES ALLOCATED?
      BEQL   50$                          ;IF EQL - NO
      BBC    #UCB$V_DL_MAPPING,-          ;ADAPTER MAPPING?
           UCB$W_DL_FLAGS(R5),50$        ;IF BC NO
      RELDPR                                     ;RELEASE DATA PATH
      RELMPR                                     ;RELEASE MAP REGISTERS
50$:   RELCHAN                                ;RELEASE CHANNEL IF OWNED
      MOVL   UCB$L_IRP(R5),R3              ;GET ADDRESS OF I/O PACKET
      MOVQ   IRP$L_SVAPTE(R3),-          ;RESTORE TRANSFER PARAMETERS
           UCB$L_SVAPTE(R5)             ;...
      BRW    PREPROCESS                    ;RETURN TO PREPROCESS UCB FIELDS
      .PAGE
      .SBTTL  INTERRUPT SERVICE ROUTINE
;++;
; DL$INT - RL11 INTERRUPT SERVICE ROUTINE
;
; FUNCTIONAL DESCRIPTION:
;
;   THIS ROUTINE IS ENTERED VIA A JSB INSTRUCTION WHEN AN INTERRUPT
;   OCCURS ON AN RL11 DISK CONTROLLER. IF THE INTERRUPT IS NOT EXPECTED,
;   THE UNSOLICITED INTERRUPT ROUTINE DISMISSES THE INTERRUPT. IF
;   THE INTERRUPT IS EXPECTED, DEVICE REGISTERS ARE SAVED AND THE
;   DRIVER IS CALLED AT ITS INTERRUPT RETURN ADDRESS. THE DRIVER FORKS,
;   CAUSING A RETURN TO THIS ROUTINE, WHICH RESTORES GENERAL REGISTERS
;   AND DISMISSES THE INTERRUPT.
;
; INPUTS:
;
;   00(SP) - POINTER TO ADDRESS OF THE IDB
;   04(SP) - SAVED R0
;   08(SP) - SAVED R1
;   12(SP) - SAVED R2
;   16(SP) - SAVED R3
;   20(SP) - SAVED R4
;   24(SP) - SAVED R5
;   28(SP) - PC AT THE TIME OF THE INTERRUPT
;   32(SP) - PSL AT THE TIME OF THE INTERRUPT
;
; OUTPUTS:
;
;   DEVICE REGISTERS ARE SAVED, IPL IS LOWERED TO FORK LEVEL, THE
;   INTERRUPT IS DISMISSED, ALL REGISTERS EXCEPT R0-R5 ARE PRESERVED.
;
;---
DL_INT::                                     ;INTERRUPT SERVICE ROUTINE
      MOVL   @(SP)+,R3                     ;REMOVE ADDRESS OF IDB FROM STACK
      ASSUME IDB$L_CSR EQ 0
      ASSUME IDB$L_OWNER EQ 4
      MOVQ   (R3),R4                      ;GET ADDRESS OF CSR AND UCB
      TSTL   R5                          ;IS R5 A ZERO
      BEQL   DL_UNSOINT                   ;IF EQL NO OWNER
      DEVICELOCK -
           LOCKADDR=UCB$L_DLCK(R5),-      ;LOCK DEVICE ACCESS
           CONDITION=NOSETIPL,-          ;DON'T CHANGE IPL
           PRESERVE=NO                   ;DON'T PRESERVE R0
      BBCC   #UCB$V_INT,-                 ;IF CLR - INTERRUPT NOT EXPECTED
           UCB$W_STS(R5),.40$           ;...

```

Sample Driver for the RL11, RL01, and RL02

```

CMPB    #CDF_READHEAD,UCB$B_CEX(R5)    ;READ HEADER FUNCTION?
BNEQ    10$                            ;IF NEQ - NO
MOVW    RL_MP(R4),UCB$W_DL_DB(R5)      ;SAVE SECTOR HEADER INFORMATION
MOVW    RL_MP(R4),UCB$W_DL_DB+2(R5)    ;...
MOVW    RL_MP(R4),UCB$W_DL_DB+4(R5)    ;...

10$:    MOVAB    RL_CS(R4),R2            ;GET ADDRESS OF CONTROL STATUS REGISTER
        MOVAB    UCB$W_DL_CS(R5),R3    ;GET ADDRESS OF REGISTER SAVE AREA
        MOVW    (R2)+,(R3)+            ;SAVE CONTROL STATUS REGISTER
        MOVW    (R2)+,(R3)+            ;SAVE BUFFER ADDRESS REGISTER
        MOVW    (R2)+,(R3)+            ;SAVE DISK ADDRESS REGISTER
        MOVW    (R2)+,(R3)+            ;SAVE MULTIPURPOSE REGISTER

20$:    MOVQ    UCB$L_FR3(R5),R3        ;RESTORE DRIVER CONTEXT
        JSB     @UCB$L_FPC(R5)        ;CALL DRIVER AT INTERRUPT RETURN ADDRESS

40$:    DEVICEUNLOCK -
        LOCKADDR=UCB$L_DLCK(R5),-    ;UNLOCK DEVICE ACCESS
        PRESERVE=NO                  ;DON'T PRESERVE RO

DL_UNSO LNT:
        POPR    #~M<R0,R1,R2,R3,R4,R5> ;RESTORE R0-R5
        REI     ;RETURN FROM INTERRUPT
        .PAGE
        .SBTTL REGISTER DUMP ROUTINE
; ++
;
; DL_REGDUMP - REGISTER DUMP ROUTINE
;
; FUNCTIONAL DESCRIPTION:
;
; THIS ROUTINE IS CALLED TO SAVE THE DEVICE REGISTERS AND UBA RESOURCE
; REGISTERS IN A SPECIFIED BUFFER. IT IS CALLED FROM THE DEVICE ERROR
; LOGGING ROUTINE AND FROM THE DIAGNOSTIC BUFFER FILL ROUTINE.
;
; INPUTS:
;
; R0      - ADDRESS OF REGISTER SAVE BUFFER
; R4      - ADDRESS OF DEVICE CONTROL STATUS REGISTER (CSR)
; R5      - ADDRESS OF UNIT CONTROL BLOCK (UCB)
;
; OUTPUTS:
;
; THE DEVICE AND UBA REGISTERS ARE SAVED IN THE SPECIFIED BUFFER.
; R0 CONTAINS THE ADDRESS OF THE NEXT EMPTY LONGWORD IN THE BUFFER.
; ALL REGISTERS EXCEPT R1 AND R2 ARE PRESERVED.
;
; --

DL_REGDUMP:
        MOVL    #<RL_NUM_REGS+5>,(R0)+ ;INSERT NUMBER OF REGISTERS
        MOVAL    UCB$W_DL_CS(R5),R1    ;GET ADDRESS OF SAVED DEVICE REGISTERS
        MOVZBL   #RL_NUM_REGS,R2      ;GET NUMBER OF DEVICE REGISTERS TO MOVE
10$:    MOVZWL   (R1)+,(R0)+            ;DUMP REGISTER IN BUFFER
        SOGTR   R2,10$                ;IF GTR - STILL MORE TO MOVE
        MOVZWL   (R1)+,(R0)+            ;DUMP DATAPATH NUMBER
        MOVL    (R1)+,(R0)+            ;DUMP DATAPATH REGISTER
        MOVL    (R1)+,(R0)+            ;DUMP FINAL MAP REGISTER
        MOVL    (R1)+,(R0)+            ;DUMP PREVIOUS MAP REGISTER
        MOVZBL   (R1)+,(R0)+            ;DUMP DATAPATH PURGE ERROR REGISTER
        RSB     ;RETURN

```

Sample Driver for the RL11, RL01, and RL02

```

.PAGE
.SBTTL MOVE TO USER BUFFER ROUTINE
; ++
; DL_MOVE_TO_BUFFER - MOVE TO USER BUFFER
; FUNCTIONAL DESCRIPTION:
; THIS ROUTINE MOVES DATA BETWEEN THE PHYSICALLY CONTIGUOUS BUFFER AND
; THE USER'S BUFFER.
; INPUTS:
; R5 - UCB ADDRESS
; OUTPUTS:
; DATA MOVE BETWEEN THE PHYSICALLY CONTIGUOUS BUFFER AND THE USER'S BUFFER.
; REGISTER'S R0, R1, AND R2 ARE DESTROYED
; --
DL_MOVE_TO_BUFFER:                                ; BUFFER MOVE ROUTINE
    BBS      #UCB$V_DL_MAPPING, -                ; ADAPTER MAPPING?
            UCB$W_DL_FLAGS(R5), 10$           ; IF BS YES NOTHING TO MOVE
    CMPB     #CDF_READDATA,UCB$B_CEX(R5) ; READ DATA OPERATION?
    BNEQ    10$                                ; IF NEQ NOT A READ
    BBS     #0,UCB$B_DL_DCHEK(R5), -          ; DATA CHECK IN PROGRESS?
            10$                                ; IF BS YES NOTHING TO MOVE
    TSTL    UCB$A_DL_MOVRTN(R5)                ; ANYTHING TO MOVE?
    BEQL    20$                                ; IF EQL NO
    MOVL    UCB$L_DL_BUFADR(R5),R0             ; GET USER BUFFER POINTER
    MOVL    UCB$A_DL_BUF_VA(R5),R1            ; GET PHYSICALLY CONTIGUOUS BUFFER ADDRESS
    MOVZWL  UCB$W_DL_PBCR(R5),R2              ; GET NUMBER OF BYTES TO TRANSFER
    JSB     @UCB$A_DL_MOVRTN(R5)              ; CALL MOVE ROUTINE
    MOVL    R0,UCB$L_DL_BUFADR(R5)            ; SAVE INTERNAL BUFFER POINTER
    MOVAB   G^IOC$MOVTOUSER2, -                ; SET NEXT MOVE ROUTINE TO BE USED
            UCB$A_DL_MOVRTN(R5)                ;
10$:      RSB                                ; RETURN
20$:      MOVAB   G^IOC$MOVTOUSER, -            ; SET NEXT MOVE ROUTINE TO BE USED
            UCB$A_DL_MOVRTN(R5)                ;
    RSB                                ; RETURN

```

Sample Driver for the RL11, RL01, and RL02

```

.PAGE
.SBTTL MOVE FROM USER BUFFER ROUTINE
;
;
; ++
;
; DL_MOVE_FROM_BUFFER - MOVE FROM USER BUFFER
;
; FUNCTIONAL DESCRIPTION:
;
; THIS ROUTINE MOVES DATA BETWEEN THE PHYSICALLY CONTIGUOUS BUFFER AND
; THE USER'S BUFFER.
;
; INPUTS:
;
;     R5 - UCB ADDRESS
;
; OUTPUTS:
;
;     DATA MOVE BETWEEN THE PHYSICALLY CONTIGUOUS BUFFER AND THE USER'S BUFFER.
;     REGISTER'S R0,R1, AND R2 ARE DESTROYED
;
; --
DL_MOVE_FROM_BUFFER:
; BUFFER MOVE ROUTINE
    BBS     #UCB$V_DL_MAPPING,- ; ADAPTER MAPPING?
           UCB$W_DL_FLAGS(R5),10$ ; IF BS YES NOTHING TO MOVE
    CMPB   #CDF_WRITEDATA,UCB$B_CEX(R5);WRITE DATA OPERATION?
    BNEQ   10$ ; IF NEQ NOT A WRITE
    BBS     #0,UCB$B_DL_DCHEK(R5),- ; DATA CHECK IN PROGRESS?
           10$ ; IF BS YES NOTHING TO MOVE
    MOVL   UCB$L_DL_BUFADR(R5),R0 ; GET USER BUFFER POINTER
    MOVL   UCB$A_DL_BUF_VA(R5),R1 ; GET PHYSICALLY CONTIGUOUS BUFFER ADDRESS
    MOVZWL UCB$W_DL_PBCR(R5),R2 ; GET NUMBER OF BYTES TO TRANSFER
    JSB    @UCB$A_DL_MOVRTN(R5) ; CALL MOVE ROUTINE
    MOVL   R0,UCB$L_DL_BUFADR(R5) ; SAVE INTERNAL BUFFER POINTER
    MOVAB  G^IOC$MOVFRUSER2,- ; SET NEXT MOVE ROUTINE TO BE USED
           UCB$A_DL_MOVRTN(R5) ;
10$:      RSB ; RETURN
DL_END:  ; ADDRESS OF LAST LOCATION IN DRIVER
        .END

```


F

Sample Driver for the DR11-W and DRV11-WA

The following driver, XADRIVER, controls the DR11-W, a 16-bit parallel DMA interface on UNIBUS systems. The driver also controls the DRV11-WA, a 16-bit parallel DMA interface on the Q22 bus. Operational details of these devices, as well as the capabilities controlled by the driver, can be found in the *VMS-I/O User's Reference Manual: Part II*.

You can find an online copy of the driver code (XADRIVER.MAR) in SYS\$EXAMPLES.

```
.TITLE XADRIVER - VAX/VMS DR11-W AND DRV11-WA DRIVER
.IDENT 'X-15'
```

```
*****
;*
;* COPYRIGHT (c) 1978, 1980, 1982, 1984, 1985, 1986 BY
;* DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.
;* ALL RIGHTS RESERVED.
;*
;* THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
;* ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
;* INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER
;* COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
;* OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
;* TRANSFERRED.
;*
;* THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
;* AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
;* CORPORATION.
;*
;* DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
;* SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
;*
;*
*****
;+
;
; FACILITY:
;
; VAX/VMS Executive, I/O Drivers
;
; ABSTRACT:
;
; This module contains the driver for the DR11-W (Unibus) and
; DRV11-WA (Q-bus). Since the driver was originally written for
; the DR11-W, many inline comments refer to the "DR11-W" and "Unibus"
; but apply equally well to the DRV11-WA and the Q-bus.
;
; For DR11-W users:
; This driver works for all hardware revision levels of
; the DR11-W.
;
; For DRV11-WA users:
; This driver works for all hardware revision levels of
; the DRV11-WA, up through and including CS Rev C.
```

Sample Driver for the DR11-W and DRV11-WA

```
;
;
;          BECAUSE ETCH REVISION E OF THE DRV11-WA
;          PROVIDES SEVERAL CUSTOMER MODIFIABLE SETTINGS,
;          IT IS VERY IMPORTANT THAT THE USERS OF THIS
;          BOARD PROPERLY CONFIGURE IT TO BE BACKWARDS
;          COMPATIBLE WITH EARLIER REVISIONS OF THE
;          DRV11-WA.  SPECIFICALLY, ON ETCH REVISION E
;          BOARDS, JUMPERS W2, W3, AND W6 MUST BE INSTALLED.
;
;          =====
;
; ENVIRONMENT:
;
;       Kernel Mode, Non-paged
;
;--

        .SBTTL  External and local symbol definitions

; External symbols

        $ACBDEF      ; AST control block
        $ADPDEF      ; Adapter control block
        $CRBDEF      ; Channel request block
        $DCDEF       ; Device types
        $DDBDEF      ; Device data block
        $DEVDEF      ; Device characteristics
        $DPTDEF      ; Driver prologue table
        $DYNDEF      ; Dynamic data structure types
        $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
        $SSDEF       ; System status codes
        $UCBDEF      ; Unit control block
        $VECDEF      ; Interrupt vector block
        $XADEF       ; Define device specific characteristics

; Local symbols

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

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

; Other constants

XA_DEF_TIMEOUT = 10      ; 10 second default device timeout
XA_DEF_BUFSIZ  = 65535   ; Default buffer size
XA_RESET_DELAY = <<2+9>/10> ; Delay N microseconds after RESET
; (rounded up to 10 microsec intervals)

; DR11-W definitions that follow the standard UCB fields
; *** N O T E *** ORDER OF THESE UCB FIELDS IS ASSUMED
```


Sample Driver for the DR11-W and DRV11-WA

```

$DEFINI UCB
.=UCB$L_DPC+4
$DEF UCB$L_XA_ATTIN ; Attention AST listhead
      .BLKL 1
$DEF UCB$W_XA_CSRTMP ; Temporary storage of CSR image
      .BLKW 1
$DEF UCB$W_XA_BARTMP ; Temporary storage of BAR image
      .BLKW 1
$DEF UCB$W_XA_CSR ; Saved CSR on interrupt
      .BLKW 1
$DEF UCB$W_XA_EIR ; Saved EIR on interrupt
      .BLKW 1
$DEF UCB$W_XA_IDR ; Saved IDR on interrupt
      .BLKW 1
$DEF UCB$W_XA_BAR ; Saved BAR register on interrupt
      .BLKW 1
$DEF UCB$W_XA_WCR ; Saved WCR register on interrupt
      .BLKW 1
$DEF UCB$W_XA_ERROR ; Saved device status flag
      .BLKW 1
$DEF UCB$L_XA_DPR ; Data Path Register contents
      .BLKL 1
$DEF UCB$L_XA_FMPR ; Final Map Register contents
      .BLKL 1
$DEF UCB$L_XA_PMPR ; Previous Map Register contents
      .BLKL 1
$DEF UCB$W_XA_DPRN ; Saved Datapath Register Number
      .BLKW 1 ; And Datapath Parity error flag
$DEF UCB$W_XA_BAETMP ; Temporary storage of BAE (DRV11-WA
      .BLKW 1 ; only)
$DEF UCB$W_XA_BAE ; Saved BAE register (DRV11-WA only)
      .BLKW 1

; Bit positions for device-dependent status field in UCB
      $VIELD UCB,0,<- ; UCB device specific bit definitions
            <ATTNAST,,M>,- ; ATTN AST requested
            <UNEXPT,,M>,- ; Unexpected interrupt received
            <IGNORE_UNEXPT,,M>,- ; Ignore initial interrupt on DRV11-WA
            >
UCB$K_SIZE=.
      $DEFEND UCB

; Device register offsets from CSR address
      $DEFINI XA ; Start of DR11-W definitions
$DEF XA_WCR ; Word count
      .BLKW 1
$DEF XA_BAR ; Buffer address
$DEF XA_BAE ; Buffer address extension (DRV11-WA)
      .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>-
            <FNCT3,8>-
            <STATUSA,2048>- ; Define CSR STATUS bit values
            <STATUSB,1024>-
            <STATUSC,512>-
            >

```

Sample Driver for the DR11-W and DRV11-WA

```

$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
>

$VIELD XA_BAE,0,<-          ; Extended bus address register
        <MSB_ADDR,6,M>,-  ; Qbus physical address <22:16>
        <,9,>,-          ;
        <CS_REV_C,,M>,-  ; true if DRV11-WA CS Rev C
>

$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 xfer
        <PAR,,M>,-      ; Parity error during DATI/P
        <ACLO,,M>,-    ; Power fail on this processor
        <MULTI,,M>,-    ; Multi-cycle request error
        <ATTN,,M>,-    ; ATTN - same as in CSR
        <NEX,,M>,-     ; NEX - same as in CSR
        <ERROR,,M>,-   ; ERROR - same as in CSR
>

        .BLKW 1

$DEF XA_IDR                ; Input Data Buffer register
$DEF XA_ODR                ; Output Data Buffer register
        .BLKW 1

$DEFEND XA                ; End of DR11-W definitions

.SBTTL Device Driver Tables
; Driver prologue table

```

Sample Driver for the DR11-W and DRV11-WA

```

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$B_FLCK,B,SPL$C_IOLOCK8 ; Device fork IPL
DPT_STORE  UCB,UCB$B_DIPL,B,22               ; Device interrupt IPL
DPT_STORE  UCB,UCB$L_DEVCHAR,L,<-            ; Device characteristics
           DEV$M_AVL!-                       ; Available
           DEV$M_RTM!-                       ; Real Time device
           DEV$M_ELG!-                       ; Error Logging enabled
           DEV$M_IDV!-                       ; input device
           DEV$M_ODV>                       ; output device
DPT_STORE  UCB,UCB$B_DEVCLASS,B,DC$_REALTIME ; Device class
DPT_STORE  UCB,UCB$B_DEVTYPE,B,DT$_DR11W ; Device Type
DPT_STORE  UCB,UCB$W_DEVBUFSIZ,W,-          ; Default buffer size
           XA_DEF_BUFSIZ
DPT_STORE  REINIT                             ; Start of reload
           ; initialization table
DPT_STORE  DDB,DDB$L_DDT,D,XA$DDT           ; Address of DDT
DPT_STORE  CRB,CRB$L_INTD+4,D,-              ; Address of interrupt
           XA_INTERRUPT                       ; service routine
DPT_STORE  CRB,CRB$L_INTD+VEC$L_INITIAL,-    ; Address of controller
           D,XA_CONTROL_INIT                 ; initialization routine
DPT_STORE  END                                 ; End of initialization
           ; tables

; Driver dispatch table
DDTAB      -                               ; DDT-creation macro
           DEVNAM=XA,-                       ; Name of device
           START=XA_START,-                  ; Start I/O routine
           FUNCTB=XA_FUNC_TABLE,-            ; FDT address
           CANCEL=XA_CANCEL,-                ; Cancel I/O routine
           REGDMP=XA_REGDUMP,-               ; Register dump routine
           DIAGBF=<<15*4>+<<3+5+1>*4>>,-     ; Diagnostic buffer size
           ERLGBF=<<15*4>+<1*4>+<EMB$L_DV_REGS AV>> ; Error log buffer size

;
; Function dispatch table
;
XA_FUNC_TABLE:                               ; FDT for driver
FUNCTAB  ,-                                  ; Valid I/O functions
<READPBLK,READLBLK,READVBLK,WRITEPBLK,WRITELBLK,WRITEVBLK,-
SETMODE,SETCHAR,SENSEMODE,SENSECHAR>
FUNCTAB  ,                                  ; No buffered functions
FUNCTAB  XA_READ_WRITE,-                     ; Device-specific FDT
<READPBLK,READLBLK,READVBLK,WRITEPBLK,WRITELBLK,WRITEVBLK>
FUNCTAB  +EXE$READ,<READPBLK,READLBLK,READVBLK>
FUNCTAB  +EXE$WRITE,<WRITEPBLK,WRITELBLK,WRITEVBLK>
FUNCTAB  XA_SETMODE,<SETMODE,SETCHAR>
FUNCTAB  +EXE$SENSEMODE,<SENSEMODE,SENSECHAR>

.SBTTL  XA_CONTROL_INIT, Controller initialization

```

Sample Driver for the DR11-W and DRV11-WA

```

; ++
; 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 time-out
;
; 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$L_UCBLST(R5),R0      ; Address of UCB
    MOVL   RO,IDB$L_OWNER(R5)      ; Make permanent controller owner
    BISW   #UCB$M_ONLINE,UCB$W_STS(RO)
    ADPDISP SELECT=ADAP_MAPPING,-   ; Set device status "on-line"
           ADDRLIST=<<YES,1$>>,-   ; Check for adapter mapping
           CRBADDR=R8,-
           SCRATCH=R1
    BUG_CHECK UNSUPRTCPU,FATAL      ; DRV11-WA not supported on non-mapping adapter
1$: ADPDISP SELECT=QBUS,-           ; Check for QBUS machine
           ADDRLIST=<<NO,9$>>,-
           ADPADDR=R1
    MOVB   #DT$XA_DRV11WA,-        ; If this is a Q-bus, then this is
           UCB$B_DEVTYPE(RO)      ; a DRV11-WA rather than a DR11-W.
;
; +
;
; DRV11-WAs at CS revision B and earlier incorrectly generated an interrupt
; whenever the Interrupt Enable control bit (IE) underwent a low to high
; transition. This phenomenon does not occur in boards at CS revision C.
;
; To account for this unsolicited interrupt, IGNORE_UNEXPT is set at
; initialization for all DRV11-WAs at CS revisions prior to C. When this
; bit is set, the next unexpected interrupt (as determined by the INT bit
; in UCB status word, which is set whenever an I/O request is outstanding)
; is discarded. The IGNORE_UNEXPT flag is necessary because driver
; initialization occurs at a different IPL from the interrupt handling
; routine.
;
; -
    BICW   #UCB$M_IGNORE_UNEXPT,-   ; start out assuming this is a
           UCB$W_DEVSTS(RO)        ; CS Rev C DRV11-WA
    TSTW   XA_BAR(R4)                ; BAR and BAE share the same physical
    MOVW   XA_BAE(R4),R1             ; address -- they must be read in order.
    BBS    #XA_BAE$V_CS_REV_C,R1,9$ ; branch around if CS Rev C

```

Sample Driver for the DR11-W and DRV11-WA

```
; BAE<15> is always set if the DRV11-WA is at CS Rev C or later.
BISW   #UCB$M_IGNORE_UNEXPT,- ; set flag so all interrupts are
      UCB$W_DEVSTS(RO)       ; discarded until further notice

; If powerfail has occurred and device was active, force device timeout.
; The user can set his own timeout interval for each request. Timeout
; is forced so a very long timeout period will be short circuited.
9$:    BBS     #UCB$V_POWER,UCB$W_STS(RO),10$
      ; Branch if powerfail
      BISB    #VEC$M_PATHLOCK,CRB$L_INTD+VEC$B_DATAPATH(R8)
      ; Permanently allocate direct datapath
10$:   BSBW    XA_DEV_HWRESET
      RSB     ; Done

      .SBTTL  XA_READ_WRITE, FDT for device data transfers

;++;
; XA_READ_WRITE, FDT for READLBLK,READVBLK,READPBLK,WRITELBLK,WRITEVBLK,
; WRITEPBLK
;
; Functional description:
;
; 1) Rejects QUEUE I/O's with odd transfer count
; 2) Rejects QUEUE I/O's for BLOCK MODE request to UBA Direct Data
;    PATH on odd byte boundary
; 3) Stores request timeout count specified in P3 into IRP
; 4) Stores FNCT bits specified in P4 into IRP
; 5) Stores word to write into ODR from P5 into IRP
; 6) Checks block mode transfers for memory modify access
;
; Inputs:
;
; R3 = Address of IRP
; R4 = Address of PCB
; R5 = Address of UCB
; R6 = Address of CCB
; R8 = Address of FDT routine
; AP = Address of P1
;     P1 = Buffer Address
;     P2 = Buffer size in bytes
;     P3 = Request timeout period (conditional on IO$M_TIMED)
;     P4 = Value for CSR FNCT bits (conditional on IO$M_SETFNCT)
;     P5 = Value for ODR (conditional on IO$M_SETFNCT)
;     P6 = Address of Diagnostic Buffer
;
; Outputs:
;
; R0 = Error status if odd transfer count
; IRP$L_MEDIA = Timeout count for this request
; IRP$L_SEGVBN = FNCT bits for DR11-W CSR and ODR image
;
;--
XA_READ_WRITE:
; The IO$M_INHERLOG ("inhibit error logging") function modifier was not
; intended to be used by this driver. However, since the definition for
; the IO$M_RESET modifier used to be the same as that for IO$M_INHERLOG,
; the error logging routine incorrectly used the IO$M_RESET bit to
; determine whether it should log errors. To solve this problem, the
; definition for IO$M_RESET was changed. For the sake of old programs, we
; manually move the RESET bit to its new location.
```

Sample Driver for the DR11-W and DRV11-WA

```

        BCC      #IO$V_INHERLOG,IRP$W_FUNC(R3),1$
                ; Branch if old reset bit not set
        BISW     #IO$M_RESET,IRP$W_FUNC(R3)
                ; Set new reset bit
1$:      BLBC    P2(AP),10$
                ; Branch if transfer count even
2$:      MOVZWL  #SS$BADPARAM,RO
                ; Set error status code
5$:      JMP     G^EXE$ABORTIO
                ; Abort request
10$:     MOVZWL  IRP$W_FUNC(R3),R1
                ; Fetch I/O Function code
        MOVL    P3(AP),IRP$L_MEDIA(R3)
                ; Set request specific timeout count
        BBS     #IO$V_TIMED,R1,15$
                ; Branch if timeout specified
        MOVL    #XA_DEF_TIMEOUT,IRP$L_MEDIA(R3)
                ; Else set default timeout value
15$:     BBC     #IO$V_DIAGNOSTIC,R1,20$
                ; Branch if not maintenance request
        EXTZV   #IO$V_FCODE,#IO$S_FCODE,R1,R1
                ; AND out all function modifiers
        CMPB    #IO$_READPBLK,R1
                ; If maintenance function, must be
                ; physical I/O read or write
        BEQL    20$
        CMPB    #IO$_WRITEPBLK,R1
        BEQL    20$
        MOVZWL  #SS$_NOPRIV,RO
                ; No privilege for operation
        BRB     5$
                ; Abort request
20$:     EXTZV   #0,#3,P4(AP),RO
                ; Get value for FNCT bits
        ASHL    #XA_CSR$V_FNCT,RO,IRP$L_SEGVBN(R3)
                ; Shift into position for CSR
        MOVW    P5(AP),IRP$L_SEGVBN+2(R3)
                ; Store ODR value for later
; If this is a block mode transfer, check buffer for modify access
; whether or not the function is read or write. The DR11-W does
; not decide whether to read or write, the user's device does.
; For word mode requests, return to read check or write check.
;
; If this is a BLOCK MODE request and the UBA Direct Data Path is
; in use, check the data buffer address for word alignment. If buffer
; is not word aligned, reject the request.
        BBS     #IO$V_WORD,IRP$W_FUNC(R3),30$
                ; Branch if word mode transfer
        BBS     #XA$V_DATAPATH,UCB$L_DEVDEPEND(R5),25$
                ; Branch if Buffered Data Path in use
        BLBS    P1(AP),2$
                ; DDP, branch on bad alignment
25$:     JMP     G^EXE$MODIFY
                ; Check buffer for modify access
30$:     RSB
                ; Return

.SBTTL  XA_SETMODE, Set Mode, Set characteristics FDT

```

Sample Driver for the DR11-W and DRV11-WA

```

; ++
; XA_SETMODE, FDT routine to process SET MODE and SET CHARACTERISTICS
;
; Functional description:
;
;     If IO$M_ATTNAST modifier is set, queue attention AST for the device.
;     If IO$M_DATAPATH modifier is set, queue packet.
;     Else, finish I/O.
;
; Inputs:
;
;     R3 = I/O packet address
;     R4 = PCB address
;     R5 = UCB address
;     R6 = CCB address
;     R7 = Function code
;     AP = QIO Parameter list address
;
; Outputs:
;
;     If IO$M_ATTNAST is specified, queue AST on UCB attention AST list.
;     If IO$M_DATAPATH is specified, queue packet to driver.
;     Else, use exec routine to update device characteristics.
;
; --
XA_SETMODE:
    MOVZWL  IRP$W_FUNC(R3),RO      ; Get entire function code
    BBC     #IO$V_ATTNAST,RO,20$   ; Branch if not an ATTN AST

; Attention AST request
    PUSHR  #~M<R4,R7>
    MOVAB  UCB$L_XA_ATTEN(R5),R7   ; Address of ATTN AST control block list
    JSB   G^COM$SETATTNAST        ; Set up attention AST
    POPR   #~M<R4,R7>
    BLBC  RO,50$                  ; Branch if error
    BISW  #UCB$M_ATTNAST,UCB$W_DEVSTS(R5)
                                           ; Flag ATTN AST expected.
    BBC   #UCB$V_UNEXPT,UCB$W_DEVSTS(R5),10$
                                           ; Deliver AST if unsolicited interrupt

    BSBW  DEL_ATTNAST
10$:  MOVZBL #SS$NORMAL,RO         ; Set status
    JMP   G^EXE$FINISHIOC         ; That's all for now (clears R1)

; If modifier IO$M_DATAPATH is set,
; queue packet. The data path is changed at driver level to preserve
; order with other requests.
20$:  BBS   S^#IO$V_DATAPATH,RO,30$ ; If BDP modifier set, queue packet
    JMP   G^EXE$SETCHAR          ; Set device characteristics

; This is a request to change data path useage, 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,RO        ; No priv for operation
50$:  CLRL  R1
    JMP   G^EXE$ABORTIO         ; Abort IO on error

```

Sample Driver for the DR11-W and DRV11-WA

```
.SBTTL XA_START, Start I/O routines
; ++
; XA_START - Start a data transfer, set characteristics, enable ATTN AST.
;
; Functional Description:
;
; This routine has two major functions:
;
; 1) Start an I/O transfer. This transfer can be in either word
;    or block mode. The FNCTN bits in the DR11-W CSR are set. If
;    the transfer count is zero, the STATUS bits in the DR11-W CSR
;    are read and the request completed.
;
; 2) Set Characteristics. If the function is change data path, the
;    new data path flag is set in the UCB.
;
; Inputs:
;
; R3 = Address of the I/O request packet
; R5 = Address of the UCB
;
; Outputs:
;
; R0 = final status and number of bytes transferred
; R1 = value of CSR STATUS bits and value of input data buffer register
; Device errors are logged
; Diagnostic buffer is filled
;
; --
.ENABL LSB

XA_START:
; Retrieve the address of the device CSR
    ASSUME IDB$L_CSR EQ 0
    MOVL   UCB$L_CRB(R5),R4      ; Address of CRB
    MOVL   @CRB$L_INTD+VEC$L_IDB(R4),R4
                                        ; Address of CSR

; Fetch the I/O function code
    MOVZWL IRP$W_FUNC(R3),R1      ; Get entire function code
    MOVW   R1,UCB$W_FUNC(R5)      ; Save FUNC in UCB for Error Logging
    EXTZV  #IO$V_FCODE,#IO$$_FCODE,R1,R2 ; Extract function field

; Dispatch on function code. If this is SET CHARACTERISTICS, we will
; select a data path for future use.
; If this is a transfer function, it will either be processed in word
; or block mode.
    CMPB   #IO$_SETCHAR,R2        ; Set characteristics?
    BNEQ   3$
```


Sample Driver for the DR11-W and DRV11-WA

```

; ++
; SET CHARACTERISTICS - Process Set Characteristics QIO function
;
; INPUTS:
;
;     XA_DATAPATH bit in Device Characteristics specifies which data path
;     to use.  If bit is a one, use buffered data path.  If zero, use
;     direct datapath.
;
; OUTPUTS:
;
;     CRB is flagged as to which datapath to use.
;     DEVDEPEND bits in device characteristics is updated
;     XA_DATAPATH = 1 -> buffered data path in use
;     XA_DATAPATH = 0 -> direct data path in use
; --

    MOVL   UCB$L_CRB(R5),R0           ; Get CRB address
    MOVQ   IRP$L_MEDIA(R3),UCB$B_DEVCLASS(R5) ; Set device characteristics
    BISB   #VEC$M_PATHLOCK,CRB$L_INTD+VEC$B_DATAPATH(RO)
                                           ; Assume direct datapath
    BBC    #XA$V_DATAPATH,UCB$L_DEVDEPEND(R5),2$ ; Were we right?
    BICB   #VEC$M_PATHLOCK,CRB$L_INTD+VEC$B_DATAPATH(RO) ; Set buffered datapath
2$:
    CLRL   R1                       ; Return Success
    MOVZWL #SS$_NORMAL,R0
    REQCOM

; If subfunction modifier for device reset is set, do one here
3$:   BBC    S^#IO$V_RESET,R1,4$      ; Branch if not device reset
       BSBW   XA_DEV_RESET           ; Reset DR11-W

; This must be a data transfer function - i.e. READ OR WRITE
; Check to see if this is a zero length transfer.
; If so, only set CSR FNCT bits and return STATUS from CSR
4$:   TSTW   UCB$W_BCNT(R5)          ; Is transfer count zero?
       BNEQ   10$                   ; No, continue with data transfer
       BBC    S^#IO$V_SETFNCT,R1,6$ ; Set CSR FNCT specified?
       DEVICELOCK -
           LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
           SAVIPL=- (SP),-          ; Save current IPL
           PRESERVE=NO              ; Don't preserve RO
       MOVW   IRP$L_SEGVBN+2(R3),XA_ODR(R4)
                                           ; Store word in ODR
       MOVZWL XA_CSR(R4),RO
       BICW   #<XA_CSR$M_FNCT!XA_CSR$M_ERROR>,RO
       BISW   IRP$L_SEGVBN(R3),RO
       BISW   #XA_CSR$M_ATTN,RO      ; Force ATTN on to prevent lost interrupt
       MOVW   RO,XA_CSR(R4)
       BBC    #XA$V_LINK,UCB$L_DEVDEPEND(R5),5$ ; Link mode?
       BICW3  #XA$K_FNCT2,RO,XA_CSR(R4) ; Make FNCT bit 2 a pulse
5$:
       DEVICEUNLOCK -
           LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
           NEWIPL=(SP)+,-          ; Enable interrupts
           PRESERVE=NO
6$:
       BSBW   XA_REGISTER            ; Fetch DR11-W registers
       BLBS   RO,7$                 ; If error, then log it
       JSB   G^ERL$DEVICERR         ; Log a device error
7$:   JSB   G^IOC$DIAGBUFILL        ; Fill diagnostic buffer if specified
       MOVL   UCB$W_XA_CSR(R5),R1   ; Return CSR and EIR in R1
       MOVZWL UCB$W_XA_ERROR(R5),RO ; Return status in RO

```

Sample Driver for the DR11-W and DRV11-WA

```

        BISB    #XA_CSR$M_IE,XA_CSR(R4) ; Enable device interrupts
        REQCOM                      ; Request done

; Build CSR image in R0 for later use in starting transfers
10$:
        MOVZWL  UCB$W_BCNT(R5),R0      ; Fetch byte count
        DIVL3   #2,R0,UCB$L_XA_DPR(R5) ; Make byte count into word count
        ;
        ; Set up UCB$W_CSRTMP used for loading CSR later
        ;
        MOVZWL  XA_CSR(R4),R0
        BICW    #^C<XA_CSR$M_FNCT>,R0
        BISW    #XA_CSR$M_IE!XA_CSR$M_ATTN,R0 ; Set Interrupt Enable and ATTN
        BBC     S^#IO$V_SETFNCT,R1,20$ ; Set FNCT bits in CSR?
        BICW    #<XA_CSR$M_FNCT>,R0    ; Yes, Clear previous FNCT bits
        BISB    IRP$L_SEGVBN(R3),R0    ; OR in new value
20$:
        BBC     S^#IO$V_DIAGNOSTIC,R1,23$ ; Check for maintenance function
        BISW    #XA_CSR$M_MAINT,R0    ; Set maintenance bit in CSR image

; Is this a word mode or block mode request?
23$:
        MOVW    RO,UCB$W_XA_CSRTMP(R5) ; Save CSR image in UCB
        BBC     S^#IO$V_WORD,R1,BLOCK_MODE ; Check if word or block mode
        BRW    WORD_MODE                ; Branch to handle word mode

; ++
; BLOCK MODE -- Process a Block Mode (DMA) transfer request
;
; FUNCTIONAL DESCRIPTION:
;
; This routine takes the buffer address, buffer size, function code,
; and function modifier fields from the IRP. It calculates the UNIBUS
; address, allocates the UBA map registers, loads the DR11-W device
; registers and starts the request.
; --
; Set up UBA
; Start transfer
BLOCK_MODE:
; If IO$M_CYCLE subfunction is specified, set CYCLE bit in CSR image
        BBC     #IO$V_CYCLE,R1,25$    ; Set CYCLE bit in CSR?
        BISW    #XA_CSR$M_CYCLE,UCB$W_XA_CSRTMP(R5) ; If yes, OR into CSR image

; Allocate UBA data path and map registers
25$:
        REQDPR                      ; Request UBA data path
        REQMPR                      ; Request UBA map registers
        LOADUBA                     ; Load UBA map registers

; Calculate the UNIBUS transfer address for the DR11-W from the UBA
; map register address and byte offset.

```

Sample Driver for the DR11-W and DRV11-WA

```

MOVZWL  UCB$W_BOFF(R5),R1      ; Byte offset in first page of xfer
MOVL    UCB$L_CRB(R5),R2      ; Address of CRB
INSV    CRB$L_INTD+VEC$W_MAPREG(R2),#9,#9,R1
                                ; Insert page number
EXTZV   #16,#2,R1,R2         ; Extract bits 17:16 of bus address
CMPB    #DT$_DR11W,-         ; If this is a DR11-W,
                                UCB$B_DEVTYPE(R5)
BEQL    100$                  ; then branch.
MOVW    R2,UCB$W_XA_BAETMP(R5) ; Save value of BAE prior to transfer
CLRL    R2                    ; Clear XBA bits
100$:   ASHL  #XA_CSR$V_XBA,R2,R2 ; Shift extended memory bits for CSR
        BISW  #XA_CSR$M_GO,R2    ; Set "GO" bit into CSR image
        BISW  R2,UCB$W_XA_CSRTMP(R5) ; Set into CSR image we are building
        BICW3 #<XA_CSR$M_GO!XA_CSR$M_CYCLE>,UCB$W_XA_CSRTMP(R5),R0
                                ; CSR image less "GO" and "CYCLE"
        BICW3 #XA$K_FNCT2,UCB$W_XA_CSRTMP(R5),R2 ; CSR image less FNCT bit 2
        MOVW  R1,UCB$W_XA_BARTMP(R5) ; Save BAR for error logging

; At this juncture:
;   RO = CSR image less "GO" and "CYCLE"
;   R1 = low 16 bits of transfer bus address
;   R2 = CSR image less FNCT bit 2
;   UCB$L_XA_DPR(R5) = transfer count in words
;   UCB$W_XA_CSRTMP(R5) = CSR image to start transfer with

; Set DR11-W registers and start transfer
; Note that read-modify-write cycles are NOT performed to the DR11-W CSR.
; The CSR is always written directly into. This prevents inadvertently setting
; the EIR select flag (writing bit 15) if error happens to become true.

DEVICELOCK -
        LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
        SAVIPL=-(SP),-           ; Save current IPL
        PRESERVE=NO              ; Don't preserve R0
SETIPL  #31,-                    ; Raise to IPL POWER
ENVIRON=UNIPROCESSOR
MNEGW   UCB$L_XA_DPR(R5),XA_WCR(R4)
                                ; Load negative of transfer count
MOVW    R1,XA_BAR(R4)            ; Load low 16 bits of bus address
CMPB    #DT$_DR11W,-           ; If this is a DR11-W,
                                UCB$B_DEVTYPE(R5)
BEQL    200$                    ; then branch.
MOVW    UCB$W_XA_BAETMP(R5),-   ; Load high bits of bus address
        XA_BAE(R4)
200$:   MOVW  RO,XA_CSR(R4)       ; Load CSR image less "GO" and "CYCLE"
        BBC   #XA$V_LINK,UCB$L_DEVDEPEND(R5),26$ ; Link mode?
        MOVW  R2,XA_CSR(R4)     ; Yes, load CSR image less "FNCT" bit 2
        BRB   126$              ; Only if link mode in dev characteristics
26$:    MOVW  UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ; Move all bits to CSR

; Wait for transfer complete interrupt, powerfail, or device timeout
126$:   WFIKPCX XA_TIME_OUT,IRP$L_MEDIA(R3) ; Wait for interrupt

; Device has interrupted, FORK
        IOFORK                    ; FORK to lower IPL

; Handle request completion, release UBA resources, check for errors

```

Sample Driver for the DR11-W and DRV11-WA

```

MOVZWL #SS$_NORMAL, -(SP) ; Assume success, store code on stack
CLRW   UCB$W_XA_DPRN(R5)  ; Clear DPR number and DPR error flag
PURDPR ; Purge UBA buffered data path
BLBS   RO, 27$           ; Branch if no datapath error
MOVZWL #SS$_PARITY, (SP) ; Flag parity error on device
INCB   UCB$W_XA_DPRN+1(R5) ; Flag PDR error for log
27$:   MOVL   R1, UCB$L_XA_DPR(R5) ; Save data path register in UCB
      EXTZV #VEC$V_DATAPATH, - ; Get Datapath register no.
      #VEC$S_DATAPATH, - ; For Error Log
      CRB$L_INTD+VEC$B_DATAPATH(R3), RO
      MOVB   RO, UCB$W_XA_DPRN(R5) ; Save for later in UCB
      EXTZV #9, #7, UCB$W_XA_BAR(R5), RO ; Low bits, final map register no.
      CMPB   #DT$_DR11W, - ; If this is a DR11-W,
      UCB$B_DEVTYPE(R5)
      BEQL   300$ ; then branch.
      MOVZWL UCB$W_XA_BAE(R5), R1 ; Fetch high bits of map register no.
      BRB   310$
300$:  EXTZV #4, #2, UCB$W_XA_CSR(R5), R1 ; Hi bits of map register no.
310$:  INSV   R1, #7, #2, RO ; Entire map register number
      CMPW   RO, #496 ; Is map register number in range?
      BGTR   28$ ; No, forget it - compound error
      MOVL   (R2)[RO], UCB$L_XA_FMPR(R5) ; Save map register contents
      CLRL   UCB$L_XA_PMPR(R5) ; Assume no previous map register
      DECL   RO ; Was there a previous map register?
      CMPV   #VEC$V_MAPREG, #VEC$S_MAPREG, -
      CRB$L_INTD+VEC$W_MAPREG(R3), RO
      BGTR   28$ ; No if gtr
      MOVL   (R2)[RO], UCB$L_XA_FMPR(R5) ; Save previous map register contents
28$:   RELMPR ; Release UBA resources
      RELDPR

; Check for errors and return status

TSTW   UCB$W_XA_WCR(R5) ; All words transferred?
BEQL   30$ ; Yes
MOVZWL #SS$_OPINCOMPL, (SP) ; No, flag operation not complete
30$:   BBC    #XA_CSR$V_ERROR, UCB$W_XA_CSR(R5), 35$ ; Branch on CSR error bit
      MOVZWL UCB$W_XA_ERROR(R5), (SP) ; Flag for controller/drive error status
      BSBW   XA_DEV_RESET ; Reset DR11-W
35$:   BLBS   (SP), 40$ ; Any errors after all this?

      CMPW   (SP), #SS$_OPINCOMPL ; Log the error, unless this is
      BNEQ   37$ ; a DRV11-WA running in link mode
      CMPB   #DT$_DR11W, - ; and the operation is incomplete,
      UCB$B_DEVTYPE(R5) ; in which case it is an expected
      BEQL   37$ ; error and not worth logging.
      BBS    #XA$V_LINK, - ; ...
      UCB$L_DEVDEPEND(R5), 40$ ; ...
37$:   JSB    G^ERL$DEVICERR ; Log the error.

40$:   BSBW   DEL_ATTNAST ; Deliver outstanding ATTN AST's
      JSB    G^IOC$DIAGBUFILL ; Fill diagnostic buffer
      MOVL   (SP)+, RO ; Get final device status
      MULW3 #2, UCB$W_XA_WCR(R5), R1 ; Calculate final transfer count
      ADDW   UCB$W_BCNT(R5), R1
      INSV   R1, #16, #16, RO ; Insert into high byte of IOSB
      MOVL   UCB$W_XA_CSR(R5), R1 ; Return CSR and EIR in IOSB
      BISB   #XA_CSR$M_IE, XA_CSR(R4) ; Enable interrupts
      REQCOM ; Finish request in exec

```

Sample Driver for the DR11-W and DRV11-WA

```

.DSABL LSB
;
; ++
; WORD MODE -- Process word mode (interrupt per word) transfer
;
; FUNCTIONAL DESCRIPTION:
;
; Data is transferred one word at a time with an interrupt for each word.
; The request is handled separately for a write (from memory to DR11-W
; and a read (from DR11-W to memory).
; For a write, data is fetched from memory, loaded into the ODR of the
; DR11-W and the system waits for an interrupt. For a read, the system
; waits for a DR11-W interrupt and the IDR is transferred into memory.
; If the unsolicited interrupt flag is set, the first word is transferred
; directly into memory without waiting for an interrupt.
; --
;
; .ENABL LSB
WORD_MODE:

; Dispatch to separate loops on READ or WRITE

        CMPB    #IO$_READPBLK,R2          ; Check for read function
        BNEQ   10$                        ; Br if not, must be write function
        BRW    30$                        ; Else, read

; ++
; WORD MODE WRITE -- Write (output) in word mode
;
; FUNCTIONAL DESCRIPTION:
;
; Transfer the requested number of words from user memory to
; the DR11-W ODR one word at a time, wait for interrupt for each
; word.
; --
10$:
        BSBW   MOVFRUSER                   ; Get two bytes from user buffer
        DEVLCK -
        LOCKADR=UCB$_DLCK(R5),-           ; Lock device access
        SAVIPL=- (SP),-                   ; Save current IPL
        PRESERVE=NO                       ; Don't preserve R0
        SETIPL #31,-                      ; Flag interrupt expected
        ENVIRON=UNIPROCESSOR              ; Raise IPL to power
        MOVW   R1,XA_ODR(R4)               ; Move data to DR11-W
        MOVW   UCB$_XA_CSRTMP(R5),XA_CSR(R4) ; Set DR11-W CSR
        BBC   #XA$_V_LINK,UCB$_DEVDEPEND(R5),15$ ; Link mode?
        BICW3 #XA$_K_FNCT2,UCB$_XA_CSRTMP(R5),XA_CSR(R4) ; Clear interrupt FNCT bit 2
                                                ; Only if link mode specified

15$:

; Wait for interrupt, powerfail, or device timeout
        WFIKPCX XA_TIME_OUTW,IRP$_L_MEDIA(R3)

; Check for errors, decrement transfer count, and loop until complete

```

Sample Driver for the DR11-W and DRV11-WA

```

IOFORK                                ; Fork to lower IPL
CMPB  #DT$_DR11W,-                    ; Branch if this is a DR11-W
      UCB$_DEVTYPE(R5)
BEQL  17$
BBC   #XA_CSR$V_ERROR,-               ; DRV11-WA - check ERROR bit in CSR.
      UCB$_XA_CSR(R5),20$             ; Branch on success.
BRW   40$                              ; Branch on error.
17$:  BITW  #XA_EIR$_NEX!-
      XA_EIR$_MULTI!-
      XA_EIR$_ACLO!-
      XA_EIR$_PAR!-
      XA_EIR$_DLT,UCB$_XA_EIR(R5) ; Any errors?
BEQL  20$                              ; No, continue
BRW   40$                              ; Yes, abort transfer.
20$:  DECW  UCB$_L_XA_DPR(R5)          ; All words transferred?
      BNEQ  10$                       ; No, loop until finished.

; Transfer is done, clear interrupt expected flag and FORK
; All words read or written in WORD MODE. Finish I/O.
RETURN_STATUS:
      JSB   G^IOC$DIAGBUFILL           ; Fill diagnostic buffer if present
      BSBW  DEL_ATTNAST                 ; Deliver outstanding ATTN AST's
      MOVZWL #SS$_NORMAL,R0            ; Complete success status
22$:  MULW3 #2,UCB$_L_XA_DPR(R5),R1     ; Calculate actual bytes xferred
      SUBW3 R1,UCB$_BCNT(R5),R1        ; From requested number of bytes
      INSV  R1,#16,#16,R0              ; And place in high word of R0
      MOVL  UCB$_XA_CSR(R5),R1         ; Return CSR and EIR status
      BISB  #XA_CSR$_IE,XA_CSR(R4)     ; Enable device interrupts
      REQCOM                               ; Finish request in exec

; ++
; WORD MODE READ -- Read (input) in word mode
;
; FUNCTIONAL DESCRIPTION:
;
; Transfer the requested number of words from the DR11-W IDR into
; user memory one word at a time, wait for interrupt for each word.
; If the unexpected (unsolicited) interrupt bit is set, transfer the
; first (last received) word to memory without waiting for an
; interrupt.
; --
30$:  DEVICELOCK -
      LOCKADDR=UCB$_L_DLCK(R5),- ; Lock device access
      SAVIPL=-(SP),-             ; Save current IPL
      PRESERVE=NO                 ; Don't preserve R0

; If an unexpected (unsolicited) interrupt has occurred, assume it
; is for this READ request and return value to user buffer without
; waiting for an interrupt.
      BBCC  #UCB$_V_UNEXPT,-
      UCB$_W_DEVSTS(R5),32$      ; Branch if no unexpected interrupt

      DEVICEUNLOCK -
      LOCKADDR=UCB$_L_DLCK(R5),- ; Unlock device access
      NEWIPL=(SP)+,-             ; Enable interrupts
      PRESERVE=NO

      BRB   37$                   ; continue

32$:  SETIPL #IPL$_POWER,-
      ENVIRON=UNIPROCESSOR

35$:

```

Sample Driver for the DR11-W and DRV11-WA

```

; Wait for interrupt, powerfail, or device time-out
        WFIKPCH XA_TIME_OUTW,IRP$L_MEDIA(R3)

; Check for errors, decrement transfer count and loop until done
        IOFORK                                ; Fork to lower IPL
37$:    CMPB    #DT$_DR11W,-                    ; Branch if this is a DR11-W
        UCB$B_DEVTYPE(R5)
        BEQL   1037$
        BBC    #XA_CSR$V_ERROR,-                ; DRV11-WA - check ERROR bit in CSR.
        UCB$W_XA_CSR(R5),1038$ ; Branch on success.
        BRW   40$                                ; Branch on error.
1037$:  BITW   #XA_EIR$M_NEX!-
        XA_EIR$M_MULTII!-
        XA_EIR$M_ACLO!-
        XA_EIR$M_PAR!-
        XA_EIR$M_DLT,UCB$W_XA_EIR(R5) ; Any errors?
        BNEQ  40$                                ; Yes, abort transfer.
1038$:  BSBW   MOVTOUSER                        ; Store two bytes into user buffer

; Send interrupt back to sender. Acknowledge we got last word.
        DEVICELOCK -
        LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
        SAVIPL=- (SP),- ; Save current IPL
        PRESERVE=NO ; Don't preserve R0
        MOVW   UCB$W_XA_CSRTMP(R5),XA_CSR(R4)
        BBC    #XA$V_LINK,UCB$L_DEVDEPEND(R5),38$ ; Link mode?
        BICW3  #XA$K_FNCT2,UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ; Yes, clear FNCT 2
38$:    DECW   UCB$L_XA_DPR(R5) ; Decrement transfer count
        BNEQ  35$ ; Loop until all words transferred
        DEVICEUNLOCK -
        LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
        NEWIPL=(SP)+,- ; Enable interrupts
        PRESERVE=NO
        BRW   RETURN_STATUS ; Finish request in common code

; Error detected in word mode transfer
40$:    BSBW   DEL_ATTNAST ; Deliver ATTN AST's
        BSBW   XA_DEV_RESET ; Error, reset DR11-W
        JSB   G^IOC$DIAGBUFILL ; Fill diagnostic buffer if present
        JSB   G^ERL$DEVICERR ; Log device error
        MOVZWL UCB$W_XA_ERROR(R5),R0 ; Set controller/drive status in R0
        BRW   22$

```

Sample Driver for the DR11-W and DRV11-WA

```

        .DSABL  LSB
;
; MOVFRUSER - Routine to fetch two bytes from user buffer.
;
; INPUTS:
;
;     R5 = UCB address
;
; OUTPUTS:
;
;     R1 = Two bytes of data from user's buffer
;     Buffer descriptor in UCB is updated.
;
        .ENABL  LSB
MOVFRUSER:
    MOVAL    -(SP),R1          ; Address of temporary stack loc
    MOVZBL  #2,R2             ; Fetch two bytes
    JSB     G^IOC$MOVFRUSER   ; Call exec routine to do the deed
    MOVL    (SP)+,R1          ; Retrieve the bytes
    BRB     20$               ; Update UCB buffer pointers
;
; MOVTOUSER - Routine to store two bytes into user's buffer.
;
; INPUTS:
;
;     R5 = UCB address
;     UCB$W_XA_IDR(R5) = Location where two bytes are saved
;
; OUTPUTS:
;
;     Two bytes are stored in user buffer and buffer descriptor in
;     UCB is updated.
;
MOVTOUSER:
    MOVAB   UCB$W_XA_IDR(R5),R1 ; Address of internal buffer
    MOVZBL  #2,R2
    JSB     G^IOC$MOVTOUSER     ; Call exec
20$:
    ADDW    #2,UCB$W_BOFF(R5)   ; Add two to buffer descriptor
    BICW    #^C<^X01FF>,UCB$W_BOFF(R5) ; Modulo the page size
    BNEQ    30$                 ; If NEQ, no page boundary crossed
    ADDL    #4,UCB$L_SVAPTE(R5) ; Point to next page
30$:
    RSB
;
        .DSABL  LSB

        .PAGE
        .SBTTL  DR11-W DEVICE TIMEOUT
; ++
; DR11-W device TIME-OUT
; If a DMA transfer was in progress, release UBA resources.
; For DMA or WORD mode, deliver ATTN ASTs, log a device timeout error,
; and do a hard reset on the controller.
;
; Clear DR11-W CSR
; Return error status
;
; Power failure will appear as a device timeout
; --
        .ENABL  LSB
XA_TIME_OUT: ; Timeout for DMA transfer

```


Sample Driver for the DR11-W and DRV11-WA

```

        IOFORK                ; Fork to complete request
        PURDPR                ; Purge buffered data path in UBA
        RELMPR                ; Release UBA map registers
        RELDPR                ; Release UBA data path
        BRB      10$          ; continue

XA_TIME_OUTW:                ; Timeout for WORD mode transfer

        IOFORK                ; Fork to complete operations
10$:   MOVL      UCB$L_CRB(R5),R4      ; Fetch address of CSR
        MOVL      @CRB$L_INTD+VEC$L_IDB(R4),R4
        BSBW     XA_REGISTER          ; Read DR11-W registers
        JSB      G^IOC$DIAGBUFILL     ; Fill diagnostic buffer
        JSB      G^ERL$DEVICTMO      ; Log device time out
        BSBW     DEL_ATTNAST          ; And deliver the ASTs
        BSBW     XA_DEV_RESET         ; Reset controller
        MOVZWL   #SS$_TIMEOUT,RO      ; Assume error status
        BBC      #UCB$V_CANCEL,-      ; Branch if not cancel
        MOVL     UCB$W_STS(R5),20$
        MOVZWL   #SS$_CANCEL,RO      ; Set status
20$:   CLRL     R1
        BICW     #UCB$M_ATTNAST!UCB$M_UNEXPT,UCB$W_DEVSTS(R5)
        ; Clear unwanted flags.
        BICW     #<UCB$M_TIM!UCB$M_INT!UCB$M_TIMEOUT!UCB$M_CANCEL!UCB$M_POWER>,-
        MOVL     UCB$W_STS(R5)        ; Clear unit status flags
        REQCOM   ; Complete I/O in exec
        .DSABL   LSB
        .PAGE

        .SBTTL   XA_INTERRUPT, Interrupt service routine for DR11-W
; ++
; XA_INTERRUPT, Handles interrupts generated by DR11-W
;
; Functional description:
;
; This routine is entered whenever an interrupt is generated
; by the DR11-W. It checks that an interrupt was expected.
; If not, it sets the unexpected (unsolicited) interrupt flag.
; All device registers are read and stored into the UCB.
; If an interrupt was expected, it calls the driver back at its Wait
; For Interrupt point.
; Deliver ATTN ASTs if unexpected interrupt.
;
; Inputs:
;
; 00(SP) = Pointer to address of the device IDB
; 04(SP) = saved R0
; 08(SP) = saved R1
; 12(SP) = saved R2
; 16(SP) = saved R3
; 20(SP) = saved R4
; 24(SP) = saved R5
; 28(SP) = saved PSL
; 32(SP) = saved PC
;
; Outputs:
;
; The driver is called at its Wait For Interrupt point if an
; interrupt was expected.
; The current values of the DR11-W CSRs are stored in the UCB.
;
; --
XA_INTERRUPT:                ; Interrupt service for DR11-W
        MOVL     @(SP)+,R4            ; Address of IDB and pop SP

```

Sample Driver for the DR11-W and DRV11-WA

```

MOVQ    (R4),R4                ; CSR and UCB address from IDB
DEVICELOCK -
        LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
        CONDITION=NOSETIPL,-     ; Don't change IPL
        PRESERVE=NO              ; Don't preserve RO
; Read the DR11-W device registers (WCR, BAR, CSR, EIR, IDR) and store
; into UCB.
        BSBW    XA_REGISTER      ; Read device registers

; Check to see if device transfer request active or not
; If so, call driver back at Wait for Interrupt point and
; Clear unexpected interrupt flag.
20$:    BCC     #UCB$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$L_FR3(R5),R3  ; Restore driver's R3
        JSB     @UCB$L_FPC(R5)    ; Call driver back
        BRB     30$

; Deliver ATTN ASTs if no interrupt expected and set unexpected
; interrupt flag.
25$:    BBSC    #UCB$V_IGNORE_UNEXPT,- ; Ignore spurious interrupt -
        UCB$W_DEVSTS(R5),24$ ; (DRV11-WA only.)
        BISW    #UCB$M_UNEXPT,UCB$W_DEVSTS(R5) ; Set unexpected interrupt flag
        BSBW    DEL_ATTNAST      ; Deliver ATTN ASTs
        BISB    #XA_CSR$M_IE,XA_CSR(R4) ; Enable device interrupts
        BRB     30$

; Restore registers and return from interrupt
24$:    NOP                ; allow a breakpoint here (spurious interrupt)
30$:    DEVICEUNLOCK -
        LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
        PRESERVE=NO              ; Don't preserve RO
        POPR    #M<RO,R1,R2,R3,R4,R5> ; Restore registers
        REI     ; Return from interrupt

        .PAGE
        .SBTTL XA_REGISTER - Handle DR11-W CSR transfers
; ++
; XA_REGISTER - Routine to handle DR11-W register transfers
;
; INPUTS:
;
; R4 - DR11-W CSR address
; R5 - UCB address of unit
;
; OUTPUTS:
;
; CSR, EIR, WCR, BAR, BAE, IDR, and status are read and stored into UCB.
; The DR11-W is placed in its initial state with interrupts enabled.
; RO - .true. if no hard error
;      .false. if hard error (cannot clear ATTN)
;
; If the CSR ERROR bit is set and the associated condition can be cleared, then
; the error is transient and recoverable. The status returned is SS$DRVERR.
; If the CSR ERROR bit is set and cannot be cleared by clearing the CSR, then
; this is a hard error and cannot be recovered. The returned status is

```

Sample Driver for the DR11-W and DRV11-WA

```

; SS$_CTRLERR.
;
;          RO,R1 - destroyed, all other registers preserved.
;--
XA_REGISTER:
    MOVZWL #SS$_NORMAL,RO          ; Assume success
    MOVZWL XA_CSR(R4),R1           ; Read CSR
    MOVW   R1,UCB$_XA_CSR(R5)      ; Save CSR in UCB
    BBC   #XA_CSR$_V_ERROR,R1,55$  ; Branch if no error
    MOVZWL #SS$_DRVERR,RO         ; Assume "drive" error
55$:    BICW #^C<XA_CSR$_M_FNCT>,R1 ; Clear all uninteresting bits for later
    CMPB  #DT$_XA_DRV11WA,-       ; If this is a DRV11-WA,
        UCB$_DEVTYPE(R5)         ;
    BEQL  57$                      ; then branch.
    BISB  #<XA_CSR$_M_ERROR/256>,XA_CSR+1(R4) ; Set EIR flag
    MOVW  XA_EIR(R4),UCB$_XA_EIR(R5) ; Save EIR in UCB
    BRB   59$
57$:    BISW #XA_CSR$_M_IE,R1       ; On the DRV11-WA, if the IE bit makes
        ; a 0->1 transition while READY=1, a
        ; spurious interrupt is generated.
        ; Therefore, we leave IE high at all
        ; times.
59$:    MOVW R1,XA_CSR(R4)          ; Clear EIR flag and errors
    MOVW  XA_CSR(R4),R1           ; Read CSR back
    BBC   #XA_CSR$_V_ATTN,R1,60$   ; If attention still set, hard error
    MOVZWL #SS$_CTRLERR,RO        ; Flag hard controller error
60$:    MOVW XA_IDR(R4),UCB$_XA_IDR(R5) ; Save IDR in UCB
    MOVW  XA_BAR(R4),UCB$_XA_BAR(R5)
    CMPB  #DT$_DR11W,-           ; If this is a DR11-W,
        UCB$_DEVTYPE(R5)         ;
    BEQL  70$                      ; then branch.
    MOVW  XA_BAE(R4),UCB$_XA_BAE(R5) ; Save BAE in UCB
70$:    MOVW XA_WCR(R4),UCB$_XA_WCR(R5)
    MOVW  RO,UCB$_XA_ERROR(R5)    ; Save status in UCB
    RSB

    .SBTTL XA_CANCEL, Cancel I/O routine
; ++
; XA_CANCEL, Cancels an I/O operation in progress
;
; Functional description:
;
;     Flushes Attention AST queue for the user.
;     If transfer in progress, do a device reset to DR11-W and finish the
;     request.
;     Clear interrupt expected flag.
;
; Inputs:
;
;     R2 = negated value of channel index
;     R3 = address of current IRP
;     R4 = address of the PCB requesting the cancel
;     R5 = address of the device's UCB
;
; Outputs:
;
;--
XA_CANCEL:
        ; Cancel I/O
    BBCC #UCB$_V_ATTNAST,-
        UCB$_DEVSTS(R5),20$      ; ATTN AST enabled?

```

Sample Driver for the DR11-W and DRV11-WA

```

; Finish all ATTN ASTs for this process.
    PUSHR    #^M<R2,R6,R7>
    MOVL     R2,R6                ; Set up channel number
    MOVAB    UCB$L_XA_ATT( R5),R7 ; Address of listhead
    JSB      G^COM$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$:
    DEVICELOCK -
        LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
        SAVIPL=- (SP),-          ; Save current IPL
        PRESERVE=NO              ; Don't preserve R0
    BBC      #UCB$V_INT,-        ; br if I/O not in progress
        UCB$W_STS(R5),30$
    JSB      G^IOC$CANCELIO     ; Check if transfer going
    BBC      #UCB$V_CANCEL,-
        UCB$W_STS(R5),30$      ; Branch if not for this guy
;
; Force timeout
;
    CLRL     UCB$L_DUETIM(R5)    ; clear timer
    BISW     #UCB$M_TIM,UCB$W_STS(R5) ; set timed bit
    BICW     #UCB$M_TIMEOUT,-
        UCB$W_STS(R5)          ; Clear timed out
30$:
    DEVICEUNLOCK -
        LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
        NEWIPL=(SP)+,-          ; Enable interrupts
        PRESERVE=NO
    RSB      ; Return

    .PAGE
    .SBTTL   DEL_ATTNA( ST, Deliver ATTN ASTs
; ++
; DEL_ATTNA( ST, Deliver all outstanding ATTN ASTs
;
; Functional description:
;
; This routine is used by the DR11-W driver to deliver all of the
; outstanding attention ASTs. It is copied from COM$DELATTNA( ST in
; the exec. In addition, it places the saved value of the DR11-W CSR
; and Input Data Buffer Register in the AST parameter.
;
; Inputs:
;
; R5 = UCB of DR11-W unit
;
; Outputs:
;
; R0,R1,R2 Destroyed
; R3,R4,R5 Preserved
; --
DEL_ATTNA( ST:
    DEVICELOCK -
        LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
        SAVIPL=- (SP),-          ; Save current IPL
        PRESERVE=NO              ; Don't preserve R0
    BBCC     #UCB$V_ATTNA( ST,UCB$W_DEVSTS(R5),30$
        ; Any ATTN ASTs expected?
    PUSHR    #^M<R3,R4,R5>      ; Save R3,R4,R5

```

Sample Driver for the DR11-W and DRV11-WA

```

10$:  MOVL    8(SP),R1          ; Get address of UCB
      MOVAB  UCB$L_XA_ATT( R1),R2 ; Address of ATTN AST listhead
      MOVL   (R2),R5          ; Address of next entry on list
      BEQL   20$              ; No next entry, end of loop
      BICW   #UCB$M_UNEXPT,UCB$W_DEVSTS(R1) ; Clear unexpected interrupt flag
      MOVL   (R5),(R2)        ; Close list
      MOVW   UCB$W_XA_IDR(R1),ACB$L_KAST+6(R5)
                                   ; Store IDR in AST parameter
      MOVW   UCB$W_XA_CSR(R1),ACB$L_KAST+4(R5)
                                   ; Store CSR in AST parameter
      PUSHAB B^10$            ; Set return address for FORK
      FORK                                ; FORK for this AST

; AST fork procedure
      MOVQ   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$L_KAST(R5)
      MOVZBL #PRI$I_OCOM,R2      ; Set up priority increment
      JMP    G^SCH$QAST          ; Queue the AST

20$:  POPR   #^M<R3,R4,R5>      ; Restore registers
30$:  DEVICEUNLOCK -
      LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
      NEWIPL=(SP)+,-           ; Enable interrupts
      PRESERVE=NO
      RSB                                ; Return

      .PAGE
      .SBTTL  XA_REGDUMP - DR11-W register dump routine
; ++
; XA_REGDUMP - DR11-W Register dump routine.
;
; This routine is called to save the controller registers in a specified
; buffer. It is called from the device error logging routine and from the
; diagnostic buffer fill routine.
;
; Inputs:
;
; R0 - Address of register save buffer
; R4 - Address of Control and Status Register
; R5 - Address of UCB
;
; Outputs:
;
; The controller registers are saved in the specified buffer.
;
; CSRTMP - The last command written to the DR11-W CSR by
;          by the driver.
; BARTMP - The last value written into the DR11-W BAR by
;          the driver during a block mode transfer.
; CSR - The CSR image at the last interrupt
; EIR - The EIR image at the last interrupt
; IDR - The IDR image at the last interrupt
; BAR - The BAR image at the last interrupt
; WCR - Word count register
; ERROR - The system status at request completion
; PDRN - UBA Datapath Register number
; DPR - The contents of the UBA Data Path register
; FMPR - The contents of the last UBA Map register
; PMRP - The contents of the previous UBA Map register
; DPRF - Flag for purge datapath error

```

Sample Driver for the DR11-W and DRV11-WA

```

;
;           0 = no purger datapath error
;           1 = parity error when datapath was purged
;
; BAETMP - The last value written to the BAE by the
;           driver during a block mode transfer (DRV11-WA only)
;
; BAE - The BAE image at the last interrupt (DRV11-WA only)
;
;
; Note that the values stored are from the last completed transfer
; operation. If a zero transfer count is specified, then the
; values are from the last operation with a non-zero transfer count.
;--
XA_REGDUMP:
    MOVZBL #15,(R0)+ ; 15 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$W_XA_DPRN+1(R5),(R0)+ ; Save Datapath Parity Error Flag
    MOVZWL UCB$W_XA_BAETMP(R5),(R0)+ ; Save BAE stored prior to xfer
    MOVZWL UCB$W_XA_BAE(R5),(R0)+ ; Save BAE store following xfer
    RSB

    .PAGE
    .SBTTL XA_DEV_RESET - Device reset DR11-W
;
;
; ++
; XA_DEV_RESET - DR11-W Device reset routine
;
; This routine raises IPL to device IPL, performs a device reset to
; the required controller, and reenables device interrupts.
;
; Must be called at or below device IPL to prevent a conflict in
; acquiring the device_spinlock.
;
; 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
    DEVICELOCK -
        LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
        SAVIPL=-(SP),- ; Save current IPL
        PRESERVE=NO ; Don't preserve R0

    BSBB XA_DEV_HWRESET

    DEVICEUNLOCK -
        LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
        NEWIPL=(SP)+,- ; Enable interrupts
        PRESERVE=NO

    POPR #^M<R0,R1,R2> ; Restore registers
    RSB

```

Sample Driver for the DR11-W and DRV11-WA

```
XA_DEV_HWRESET:
    CMPB    #DT$DR11W,-          ; If this is a DR11-W,
           UCB$B_DEVTYPE(R5)    ;
    BEQL    20$                  ; then branch.
    MOVW    #XA_CSR$M_IE,XA_CSR(R4) ; Clear all writable bits but IE.
    BITB    #XA_CSR$M_RDY,XA_CSR(R4); If not READY then no xfer in progress,
    BNEQ    40$                  ; So no need to reset device
    MNEGW   #1,XA_WCR(R4)        ; Tell it only 1 byte left to xfer
    MOVB    #XA_CSR$M_CYCLE/256,- ; and complete the transfer.
           XA_CSR+1(R4)
    BRB     30$
20$:      MOVB    #<XA_CSR$M_MAINT/256>,XA_CSR+1(R4)
    CLRB    XA_CSR+1(R4)
; *** Must delay here depending on reset interval
30$:      TIMEDWAIT TIME=#XA_RESET_DELAY ; No. of 10-micro-sec intervals to wait
    MOVB    #XA_CSR$M_IE,XA_CSR(R4) ; Reenable device interrupts
40$:      RSB
XA_END:   ; End of driver label
    .END
```

G VMS Version 5.0 and Kernel-Mode Code

Several features of VMS Version 5.0 have some impact on the execution of existing non-DIGITAL-supplied kernel-mode code, most notably device drivers. This chapter describes those changes DIGITAL requires or recommends in an existing non-DIGITAL-supplied device driver. It also provides a brief explanation of key VMS concepts that are integral to an understanding of the operation of privileged code under VMS Version 5.0.

G.1 Uniprocessor and Multiprocessor Device Drivers

One of the most significant components of VMS Version 5.0 is its support of a symmetric multiprocessing environment for certain VAX systems, including the VAX 8300/8350, VAX 8800/8830/8840, and VAX 6200 series. The multiprocessing environment provided by earlier versions of VMS was asymmetric in nature. Because only the primary processor could execute kernel-mode code, kernel-mode code, including device drivers, effectively ran in a uniprocessing environment and did not need to undertake any special actions due to the multiprocessing nature of the system.

In the symmetric multiprocessing environment supported by VMS Version 5.0, however, all processors in the system can execute kernel-mode code. Consequently, privileged code must take steps to ensure that its execution and use of memory are synchronized with kernel-mode code that may be executing concurrently on another processor. Such code must maintain two dimensions of synchronization: raising to the appropriate IPL for a certain transaction, while securing the proper spin lock for the object of that transaction.

For privileged code executing within a VMS uniprocessing environment VMS Version 5.0 transparently forgoes the second of these requirements. That is, on a VAX uniprocessor, or in a VMS multiprocessor system wherein multiprocessing is *not* enabled, privileged code may securely execute by adhering to the IPL synchronization method alone.

To support both uniprocessor and multiprocessor environments in the most efficient and secure way possible, VMS Version 5.0 incorporates special logic in the System Generation Utility (SYSGEN), the device driver loading mechanism, and several synchronization macros. This code enables VMS to discern the environment in which it is executing and, most importantly, to take steps to prohibit a privileged code thread from executing without proper synchronization in a multiprocessing environment.

As discussed in Section G.3, non-DIGITAL-supplied device drivers *must* be altered to execute correctly in a VMS symmetric multiprocessing environment. The modifications discussed in Section G.3 are not required for a device driver that will be loaded and executed *only* on a VMS uniprocessor system. However, the same macros, routines, and field names used in a multiprocessing environment are accepted by VMS in a uniprocessing environment. Furthermore, the spin lock synchronization macros and routines are specially designed to execute a streamlined code that obtains IPL synchronization alone in such an environment. DIGITAL recommends

VMS Version 5.0 and Kernel-Mode Code

G.1 Uniprocessor and Multiprocessor Device Drivers

that any driver that may execute in a multiprocessing environment be updated accordingly.

The remainder of this section identifies the activities of the MULTIPROCESSING system parameter, VMS driver loading mechanisms, and the VMS synchronization macros in creating a multiprocessing or uniprocessing environment and enforcing the appropriate synchronization.

G.1.1 MULTIPROCESSING System Parameter

Every VMS system is initially booted as a single processor, regardless of its hardware configuration. The setting of the MULTIPROCESSING system parameter for the first processor in the system to boot (called the *primary processor* in a multiprocessing environment) determines which synchronization image the secondary bootstrap program (SYSBOOT) loads into memory as part of the operating system. Table G-1 describes the contents of the three possible synchronization images.

Table G-1 VMS Synchronization Images

| Image | Results |
|---------------|---|
| Uniprocessing | Synchronization is accomplished by elevating IPL. Spin lock acquisition routines only achieve IPL synchronization. |
| Full-checking | Synchronization is accomplished by both elevating IPL and obtaining an appropriate spin lock. Spin lock acquisition routines perform both of these tasks. Spin lock acquisition routines also perform spin lock rank checking and verify the spin lock synchronization IPL, issuing appropriate bugchecks if they discover violations of synchronization rules. Spin lock acquisition routines maintain various debugging aids and performance analysis aids (such as the longwords in the spin lock data structure containing the PCs of the most recent acquisitions and releases of the spin lock and the set of counters in the per-CPU database structure (CPU)). (See Section G.3.7 for additional description of full-checking synchronization.) |
| Streamlined | Synchronization is accomplished by both elevating IPL and obtaining an appropriate spin lock. Spin lock acquisition routines do <i>not</i> perform checking and do <i>not</i> record the PCs of the spin lock acquisitions and releases. |

Table G-2 lists the possible settings of the MULTIPROCESSING system parameter.

VMS Version 5.0 and Kernel-Mode Code

G.1 Uniprocessor and Multiprocessor Device Drivers

Table G–2 Settings of MULTIPROCESSING System Parameter

| Value | Result |
|-------|--|
| 0 | Loads uniprocessing synchronization image for any hardware configuration |
| 1 | Loads full-checking synchronization image and sets multiprocessing-enabled bit (SMP\$_ENABLED in SMP\$GL_FLAGS) if the hardware configuration is capable of multiprocessing and two or more processors are available; otherwise, loads uniprocessing synchronization image. This is the default value. |
| 2 | Loads full-checking synchronization image and sets multiprocessing-enabled bit regardless of the hardware configuration. |
| 3 | Loads streamlined synchronization image and sets multiprocessing-enabled bit if the hardware configuration is capable of multiprocessing and two or more processors are available; otherwise, loads uniprocessing synchronization image. |

G.1.2 Device Driver Loading

In a VMS multiprocessing environment, the presence of a device driver that does not adhere to multiprocessing synchronization conventions can be fatal to proper system functions. VMS Version 5.0 takes steps to either prohibit the enabling of multiprocessing in a VAX system that has such a driver present or prevent the loading of such a driver if multiprocessing has already been enabled.

To accomplish this, the VMS driver-loading routine assumes that any driver that can run in a VMS multiprocessing environment uses the spin lock synchronization macros and loads the appropriate I/O database fields. (See Section G.3 for information on how to produce a driver that can execute in a VMS multiprocessing environment.) Use of the spin lock synchronization macros causes VMS to set the SMP-modified bit in the DPT (DPT\$_SMPMOD in DPT\$L_FLAGS).

If multiprocessing has *not* been enabled on the system, the driver loading mechanism checks the SMP-modified bit in the DPT and takes either of the following actions:

- If the SMP-modified bit is set, the driver loading mechanism loads the driver and calls its controller and unit initialization routines, as discussed in Chapter 15.
- If the SMP-modified bit is *not* set, the driver loading mechanism sets the unmodified-driver bit (SMP\$_UNMOD_DRIVER) in SMP\$GL_FLAGS, thus prohibiting the subsequent enabling of multiprocessing on the system. It then loads the driver and calls its controller and unit initialization routines, as described in Chapter 15. If such a driver has been successfully loaded into a VMS system, you cannot subsequently enable multiprocessing.

VMS Version 5.0 and Kernel-Mode Code

G.1 Uniprocessor and Multiprocessor Device Drivers

If multiprocessing is currently enabled on the system, the driver loading mechanism checks the SMP-modified bit in the DPT and takes either of the following actions:

- If the SMP-modified bit is set, the driver loading mechanism loads the driver and calls its controller and unit initialization routines, as discussed in Chapter 15.
- If the SMP-modified bit is *not* set, the driver loading mechanism does not load the driver, returning the error status `SS$_NONSMPLDRV` to its caller.

G.1.3 VMS Synchronization Macros

To support the spin lock synchronization required in VMS multiprocessor systems, VMS Version 5.0 adds the `DEVICELock/DEVICEUNLOCK`, `FORKLOCK/FORKUNLOCK`, and `LOCK/UNLOCK` macros to the existing `SETIPL` and `DSBINT/ENBINT` macros. As discussed in Section G.3, the `SETIPL` and `DSBINT/ENBINT` macros must not be used to synchronize systemwide activities in a VMS multiprocessing environment. However, the `DEVICELock/DEVICEUNLOCK`, `FORKLOCK/FORKUNLOCK`, and `LOCK/UNLOCK` macros are designed to operate appropriately in either a multiprocessing or uniprocessing environment. According to the value of the multiprocessing-enabled bit (`SMP$_ENABLED`) in `SMP$GL_FLAGS`, the run-time code produced by these macros behaves as follows:

- If multiprocessing has *not* been enabled in the system, these macros only raise or lower IPL to the IPL required to synchronize access to the specified system resource.
- If multiprocessing has been enabled in the system, these macros call the appropriate spin lock synchronization routine, which acquires or releases the spin lock corresponding to the system resource, raising or lowering IPL as required.

The setting of the `MULTIPROCESSING` system parameter controls the disposition of the multiprocessing-enabled bit, as discussed in Section G.1.1. Appendix B describes the VMS synchronization macros in full.

G.2 Changes Required of All Existing Drivers Under VMS Version 5.0

Most changes in VMS Version 5.0 are transparent to existing non-DIGITAL-supplied drivers and can be accommodated in the driver image by simply reassembling and relinking the driver. However, there are several required—and some recommended—changes that the writers and maintainers of these drivers should make before attempting these tasks. This section describes these modifications.

In addition, if a non-DIGITAL-supplied driver is to be loaded and run in a VMS symmetric multiprocessing system, it is critical that it be adapted according to the guidelines discussed in Section G.3. Failure to adapt such drivers to use multiprocessing synchronization mechanisms may result in either a failure to load the driver or the inability to enable multiprocessing on the system.

VMS Version 5.0 and Kernel-Mode Code

G.2 Changes Required of All Existing Drivers Under VMS Version 5.0

G.2.1 Specifying the Address of the Driver's Interrupt Service Routine in the DPT

In order to provide an optional method of servicing MicroVAX 3600-series or MicroVAX II Q22-bus device interrupts at the IPLs at which they are requested, VMS Version 5.0 defines several new symbolic offsets in the interrupt dispatch vector (VEC) portion of the channel request block (CRB).

One of these symbolic offsets is significant to all device drivers. Prior to VMS Version 5.0, device drivers initialized the location in the vector containing the address of the driver's interrupt service routine by referring explicitly to its location (CRB\$_INTD+4). With VMS Version 5.0, DIGITAL recommends that all device drivers refer to this location using the symbolic offset CRB\$_INTD+VEC\$_ISR, as follows:

Old: DPT_STORE CRB,CRB\$_INTD+4,D,LP\$INT_SERV_RTN

New: DPT_STORE CRB,CRB\$_INTD+VEC\$_ISR,D,LP\$INT_SERV_RTN

To use the new symbols, you must include the \$CRBDEF and \$VECDEF structure definition macros in the driver. All structure definition macros can be found in SYS\$LIBRARY:LIB.MLB.

G.2.2 Checking, Debiting, and Crediting a Process's Byte Count Quota

VMS Version 5.0 replaces the routines EXE\$BUFFRQUOTA and EXE\$BUFQUOPRC with a set of eight new routines that manipulate a job's byte count quota and byte limit, optionally allocating a nonpaged pool buffer of the requested size. To ensure proper synchronization, programs should use these routines and avoid any direct manipulation of the nonpaged pool quota fields JIB\$_BYTCNT and JIB\$_BYTLM).

Among the new routines are the following:

| Routine | Function |
|--------------------------|---|
| EXE\$CREDIT_BYTCNT | Returns credit to a job's byte count quota |
| EXE\$CREDIT_BYTCNT_BYTLM | Returns credit to a job's byte count quota and byte count limit |
| EXE\$DEBIT_BYTCNT | Determines whether a job's buffered byte count quota usage permits the process to be granted additional buffered I/O and, if so, adjusts the job's byte count quota |
| EXE\$DEBIT_BYTCNT_NW | Same function as EXE\$DEBIT_BYTCNT, but never places a process in a resource wait state pending the return of sufficient quota |

VMS Version 5.0 and Kernel-Mode Code

G.2 Changes Required of All Existing Drivers Under VMS Version 5.0

| Routine | Function |
|-----------------------------|--|
| EXE\$DEBIT_BYTCNT_BYTLM | Determines whether a job's buffered byte count quota usage permits the process to be granted additional buffered I/O and, if so, adjusts the job's byte count quota and byte count limit |
| EXE\$DEBIT_BYTCNT_BYTLM_NW | Same function as EXE\$DEBIT_BYTCNT_BYTLM, but never places a process in a resource wait state pending sufficient quota |
| EXE\$DEBIT_BYTCNT_ALO | Same function as EXE\$DEBIT_BYTCNT, but, if quota checks succeed, allocates the requested amount of pool |
| EXE\$DEBIT_BYTCNT_BYTLM_ALO | Same function as EXE\$DEBIT_BYTCNT_BYTLM, but, if quota checks succeed, allocates the requested amount of pool |

Many drivers written prior to VMS Version 5.0 contain code sequences similar to the following:

```
JSB      G^EXE$BUFFRQUOTA      ;Would buffer allocation
                                   ;exceed byte count quota?
BLBC     RO,ERROR              ;Branch if yes
JSB      G^EXE$ALLOCBUF        ;If not, allocate buffer
BLBC     RO,ERROR              ;Branch if error
MOVL     PCB$L_JIB(R4),R5      ;Obtain job information block
SUBL2    R1,JIB$L_BYTCNT(R5)   ;Decrement job's byte count quota
```

The new routines allow you to simplify such code sequences. For instance, a single routine, EXE\$DEBIT_BYTCNT_ALO, checks and debits quotas and allocates pool. When there is not enough quota available to service the request, the routine restores the deducted amount and returns the error SS\$_EXQUOTA IN R0.

In VMS Version 5.0, the preceding code example can be rewritten as follows:

```
JSB      G^EXE$DEBIT_BYTCNT_ALO ;Check for quota violation,
                                   ;allocate buffer, decrement
BLBC     RO,ERROR              ;JIB byte count quota
                                   ;Branch if error
```

VMS Version 5.0 and Kernel-Mode Code

G.2 Changes Required of All Existing Drivers Under VMS Version 5.0

G.2.3 Referring to the Current PCB

The symbol SCH\$GL_CURPCB is obsolete and should be replaced as follows:

- If the process's P1 space is available, use the P1 space location CTL\$GL_PCB.
- If the process's P1 space is *not* available, use the FIND_CPU_DATA macro, as follows:

```
FIND_CPU_DATA RO
MOVL CPU$_L_CURPCB(R0),R1
```

The FIND_CPU_DATA macro obtains the virtual address of the per-CPU database for the processor on which it executes. Code that issues the FIND_CPU_DATA macro must adhere to the following rules:

- It must be executing in kernel mode above IPL 2 when it invokes the FIND_CPU_DATA macro.
- It must take care to prevent rescheduling after issuing the macro as long as the information returned by FIND_CPU_DATA is in use. It typically does this by remaining at an IPL greater than 2.

G.2.4 Allocating System Page-Table Entries

The system routine IOC\$ALLOSPT has been replaced by the LDR\$ALLOC_PT.

IOC\$ALLOSPT was briefly described in Versions 4.5 and 4.6 of the VAX/VMS *Release Notes* as an appropriate method for non-DIGITAL-supplied VAXBI device drivers to map a portion of a device's node space to system virtual address space. See Appendix C for a full description of LDR\$ALLOC_PT.

G.2.5 Referring to a System Process Mailbox

An existing driver that refers to either the job controller's mailbox or OPCOM's mailbox must be altered to use the new symbolic names that point to these mailboxes. Usually, it is the driver's interrupt service routine or timeout handling routine that loads the address of the mailbox UCB into R3 and calls the system routine EXE\$SNDEVMSG, as follows.

```
MOVAB G^SYS$AR_JOBCTLMB,R3 ;Set address of job controller
; mailbox
JSB G^EXE$SNDEVMSG ;Sent message to job controller
```

The new symbolic names actually refer to global pointers to the mailbox UCB structures. They include the following:

| VMS Version 5.0 | Old | Name |
|-----------------|-----------------|--------------------------|
| SY\$AR_JOBCTLMB | SY\$AL_JOBCTLMB | Job controller's mailbox |
| SY\$AR_OPRMBX | SY\$AL_OPRMBX | OPCOM's mailbox |

VMS Version 5.0 and Kernel-Mode Code

G.2 Changes Required of All Existing Drivers Under VMS Version 5.0

G.2.6 Reassembling and Relinking the Driver

Because of changes in the definitions of data structures, the behavior of system macros, the location of global symbols, and the contents of system images, it is necessary to reassemble and relink non-DIGITAL-supplied drivers regardless of whether their contents have been modified.

To do so, reassemble your driver against SYS\$LIBRARY:LIB.MLB. For example:

```
$ MACRO MYDRIVER.MAR+SYS$LIBRARY:LIB.MLB/LIBRARY
```

Relink your driver against the VMS global symbol table. If the driver consists of several source files, you must specify the file that contains the driver prologue table as the first file in the list. The linker options file must contain the statement BASE=0. For example:

```
$ CREATE MYDRIVER.OPT
BASE=0
CTRL/Z
$ LINK/NOTRACE MYDRIVER1[,MYDRIVER2,...],-
MYDRIVER.OPT/OPTIONS,-
SYS$SYSTEM:SYS.STB/SELECTIVE_SEARCH
```

The linker will report that the image has no transfer address. You may ignore this message.

Once you have linked or relinked a driver, you should copy its image to the SYS\$LOADABLE_IMAGES or SYS\$SYSTEM directory. The SYSGEN LOAD and CONNECT commands first search for a driver in the SYS\$LOADABLE_IMAGES directory. If they do not find the driver, they then search the SYS\$SYSTEM directory.

G.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

VMS Version 5.0 contains several new routines that enforce synchronization in a symmetric multiprocessing (SMP) environment. Drivers do not generally call these routines explicitly, but rather invoke VMS-supplied macros that synchronize as appropriate to the processing environment. There are only a few instances in which existing non-DIGITAL-supplied drivers must change to conform to the new synchronization mechanisms. This section outlines those instances; Chapter 3 describes synchronization rules in greater detail.

G.3.1 Specifying the Fork Lock Index

To adapt a driver to execute properly in a VMS multiprocessor environment, you must replace all instances of UCB\$B_FIPL (or FKB\$B_FIPL) with UCB\$B_FLCK (or FKB\$B_FLCK). In addition, you must replace the invocation of the DPT_STORE macro that defined the driver's fork IPL with one that defines the driver's fork lock index, as follows:

Old: DPT_STORE UCB,UCB\$B_FIPL,B,8

New: DPT_STORE UCB,UCB\$B_FLCK,B,SPL\$C_IOLOCK8

VMS Version 5.0 and Kernel-Mode Code

G.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

To use the new symbol, include the \$UCBDEF structure definition macro in the driver. Fork lock (and other spin lock) indexes, such as SPL\$_IOLOCK8, are defined by the \$SPLCODDEF definition macro as invoked by DPTAB. Replace fork IPLs with the corresponding fork lock index according to the following list:

| IPL | Fork Lock Index |
|-----|-----------------|
| 8 | SPL\$_IOLOCK8 |
| 9 | SPL\$_IOLOCK9 |
| 10 | SPL\$_IOLOCK10 |
| 11 | SPL\$_IOLOCK11 |

All structure definition macros can be found in SYS\$LIBRARY:LIB.MLB.

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

Note that, if a driver fork process is not placed into execution according to one of these means, it must obtain the fork lock itself. (See the discussion in Section G.3.6.2.)

G.3.2 Synchronizing Access to the Device Database with the Interrupt Service Routine

The device database consists of device and adapter registers, plus driver-specific UCB fields that record the status of a device. As these locations are primarily accessed by the driver's interrupt service routine, the driver fork process must take special care to synchronize with the interrupt service routine whenever it accesses them.

G.3.2.1 Synchronizing at Device IPL

Previous versions of VMS used the DSBINT macro to synchronize with the interrupt service routine at device IPL (UCB\$_DIPL), as follows:

```
DSBINT UCB$_DIPL(R5)           ;Raise IPL to device IPL
                                ;Save current IPL on stack
                                ;Access device data
.
.
.
ENBINT                          ;Restore saved IPL
```

VMS Version 5.0 and Kernel-Mode Code

G.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

Under VMS Version 5.0, this code should be modified so that it obtains the appropriate device lock, as follows:

```
DEVICELOCK - ;Secure device lock
  LOCKADDR=UCB$L_DLCK(R5),- ;(also raises IPL to device IPL)
  SAVIPL=-(SP) ;Save current IPL on stack
;Access device data
.
.
.
DEVICEUNLOCK - ;Release device lock
  LOCKADDR=UCB$L_DLCK(R5),-
  NEWIPL=(SP)+ ;Restore old IPL from stack
```

G.3.2.2 Raising IPL to IPL\$_POWER

If the device driver start-I/O routine (or fork process) raises IPL to IPL 31 (IPL\$_POWER) to check for the occurrence of a power failure and to access device registers, it must ensure that it has explicitly synchronized with the device's database at device IPL. Under VMS Version 5.0, this means that the routine must first obtain the appropriate device lock, using the DEVICELOCK macro.

Because versions of VMS prior to Version 5.0 allowed only one processor, even in a VAX multiprocessor system, to execute kernel-mode code, the following code in a driver's start-I/O routine provided adequate synchronization:

```
DSBINT ;Raise IPL to 31
;Save current IPL on stack
BBC #UCB$V_POWER,-
UCB$W_STS(R5),30$ ;If clear, no power failure
;Service power failure
.
.
;Branch
30$: ;Start device
.
.
WFIKPC
```


VMS Version 5.0 and Kernel-Mode Code

G.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

Similarly, the interrupt service routine should release the device lock when it no longer needs to access the device database. Generally, this is immediately after the routine regains control from the driver fork process and before it restores the saved registers and issues an REI instruction, as follows:

```
DEVICEUNLOCK -           ;Release device lock
  LOCKADDR=UCB$L_DLCK(R5),-
  PRESERVE=NO           ;Do not save R0
  POPR #^M<R0,R1,R2,R3,R4,R5> ;Restore registers
  REI                   ;Exit from interrupt
```

Refer to Chapters 3 and 9 for additional information on the synchronization rules imposed on a driver's interrupt service routine.

G.3.3 Controller and Unit Initialization Routines

As discussed in Section 11.1, a device driver's controller and unit initialization routines are called during driver loading and reloading and during system recovery from a power failure.

In a VMS symmetrical multiprocessing environment, any logic in a driver's controller initialization routine or unit initialization routine that takes special action to service a power failure must adhere to the following rules:

- It cannot acquire any spin locks. Controller and unit initialization routines are called at IPL 31 during power failure recovery to reinitialize I/O devices before the processors are allowed to proceed with execution at lower IPLs. Because processors may have been holding spin locks at the time of the power failure, they will not be able to release them until after they resume execution. As a result, spin locks are not available to controller and unit initialization routines.
- It cannot perform any operation that requires the intervention of other processors in the system.

G.3.3.1 Permanently Allocating Map Registers and Buffered Data Paths

Because the map registers and buffered data paths of a UNIBUS adapter are shared by the devices residing on the bus, they are synchronized at a single fork IPL and, in a VMS multiprocessing system, by a single fork lock.

Prior to VMS Version 5.0, a unit initialization routine that permanently allocated map registers or a buffered data path could do so at IPL\$_POWER (its calling IPL). Under VMS Version 5.0, however, the map register and data path allocation routines require that the appropriate fork lock be held at the time of their calling. As a result, a unit initialization routine that permanently allocates these resources must fork before calling the allocation routine. The VMS fork dispatcher ensures that, when execution of the routine resumes, it is executing at fork IPL holding the fork lock.

The consequences of forking in a unit initialization routine are discussed at length in Section 11.1.5. Refer to Sections 12.2.2.2 and 12.2.1.2 for additional information on permanently allocating map registers and buffered data paths, respectively.

VMS Version 5.0 and Kernel-Mode Code

G.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

G.3.4 Timeout Handling Routine

In a VMS multiprocessing environment, the software timeout interrupt service routine calls a driver's timeout handling routine at device IPL, holding both the appropriate fork lock and device lock.

Previous editions of this manual have suggested that a timeout handling routine can explicitly lower its IPL from device IPL to fork IPL using a SETIPL instruction. This action assumed that the thread of code that resulted in the call to the routine originated in a software interrupt granted at IPL 7 (IPL\$_TIMERFORK).

In a VMS multiprocessing system, such a forced lowering of IPL would break synchronization. In addition, similar assumptions about the origin of the calling code thread cannot be guaranteed. Instead, those timeout handling routines that must lower IPL should issue the IOFORK macro to fork.

See Section 10.2 for additional information on the timeout handling routine.

G.3.5 General Methods for Synchronizing Kernel-Mode Code

In addition to the changes in the driver routines explicitly discussed in Sections G.3.2 and G.3.3, there may be other alterations required in device drivers and other kernel-mode code before they can execute successfully in a VMS symmetric multiprocessing environment. This section provides some general discussion of these changes. You can find additional information on multiprocessing synchronization in Chapter 3.

G.3.5.1 Using the Spin Lock Synchronization Macros

You must adapt most kernel-mode code that raises or lowers IPL so that it obtains appropriate synchronization in a VMS multiprocessing environment. Determine these locations by searching for instances of the system macros SETIPL, ENBINT, and DSBINT or for an instruction such as MTPR *x*, PR\$_IPL (where *x* is an IPL value). Do not change those instances of the SETIPL and DSBINT macros intended to achieve synchronization only on the local processor. After careful inspection proves that the macro in question is intended to achieve local processor synchronization only, add the argument **environ=UNIPROCESSOR** to their invocations.

You should replace most instances of these macros with a LOCK, UNLOCK, FORKLOCK, FORKUNLOCK, DEVICELOCK, or DEVICEUNLOCK macro, as shown in Table G-3. You can substitute the appropriate usage of any of these macros wherever Table G-3 lists the LOCK and UNLOCK macros. The formats of the spin lock synchronization macros are fully described in Appendix B. Table 3-3 lists the system spin locks.

VMS Version 5.0 and Kernel-Mode Code

G.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

Table G-3 Converting IPL Synchronization to Spin Lock Synchronization

| Existing Macro | Function | New Macro | Function |
|--|---|---|--|
| SETIPL <i>ipl</i> (where <i>ipl</i> is greater than 2) | Raise IPL | LOCK <i>lockname</i> , <i>lockipl</i> | Raise IPL, acquire spin lock |
| SETIPL <i>ipl</i> (where <i>ipl</i> is greater than 2) | Lower IPL from an IPL greater than 2 | UNLOCK <i>lockname</i> , <i>lockipl</i> | Release spin lock, lower IPL |
| SETIPL <i>ipl</i> (where <i>ipl</i> is less than 3) | Lower IPL from an IPL less than 3 | SETIPL <i>ipl</i> | Lower IPL |
| DSBINT <i>ipl</i> | Save current IPL and raise to specified IPL | LOCK <i>lockname</i> , <i>lockipl</i> , <i>savipl</i> | Save current IPL, raise IPL, acquire spin lock |
| ENBINT | Lower IPL and restore saved IPL | UNLOCK <i>lockname</i> , <i>lockipl</i> | Release spin lock, lower IPL |

G.3.5.2 Interlocking Access to Data Cells and Queues

VMS Version 5.0 assigns spin lock protection to system resources as described in Table 3-3. The system time and timer queue are managed under the TIMER and HWCLK spin locks, as detailed in Section 3.1.3.

In a VMS multiprocessing environment, any thread of code that manipulates bit fields at different IPLs without spin lock protection must do so with interlocked instructions (for example, BBCCI and BBSSI).¹ Instances of the INSQUE and REMQUE instructions may need to be changed to use the INSQTI and REMQHI instructions, respectively, if they are issued to manipulate a queue at multiple IPLs. Certain cells, such as PCB\$W_ASTCNT, must be incremented and decremented using an ADAWI instruction. INCx and DECx instructions are not interlocked in a VMS multiprocessing system.

Spin locks explicitly protect various system queues and lists. For example, the AST queue in the process control block (PCB\$L_ASTQFL) is synchronized by the SCHED spin lock and the variable region of nonpaged pool is protected by the POOL spin lock.

A fork lock implicitly protects the following adapter resource wait queues (at the specified listheads) at fork IPL, as long as the drivers for all devices on the adapter that require the resources use the same fork lock.

| Listhead | | |
|---------------|------|---|
| VMS Version | Old | Name |
| 5.0 | Same | Pending-I/O queue |
| UCB\$L_IOQFL | Same | UNIBUS buffered data path wait queue |
| ADP\$L_DPQFL | Same | UNIBUS/Q22 bus map register wait queue |
| ADP\$L_MRQFL | Same | Q22 bus alternate map register wait queue |
| ADP\$L_MR2QFL | Same | |

Because a single spin lock cannot control access to list items that must be accessed by code threads executing at different IPLs, VMS Version 5.0

¹ It is illegal to intermix interlocked and noninterlocked instructions that refer to the same bit: for instance BBCC and BBCCI. Should any noninterlocked instruction refer to the same bit, the bit is not interlocked.

VMS Version 5.0 and Kernel-Mode Code

G.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

provides either a processor-specific queue or a self-relative queue for such items. The following queues (at the specified listheads) are, under VMS Version 5.0, processor-specific queues whose forward and backward links are contained in the per-CPU database (described in Section A.4 and Table A-4).

| Listhead | | |
|--------------------|--------------|-----------------------------------|
| VMS Version 5.0 | Old | Name |
| CPU\$Q_SWIQFL | SWI\$GL_FQFL | Software interrupt queue listhead |
| CPU\$L_PSFL | IOC\$GL_PSFL | I/O postprocessing queue |

The following queues are, under VMS Version 5.0, self-relative queues whose forward and backward links are contained in the data area of the system loadable image SYSTEM_PRIMITIVES.EXE.² All system macros and routines that access these queues have been converted to access them with the INSQTI and REMQHI interlocked instructions.

| Listhead | | |
|--------------------|---------------|----------------------------------|
| VMS Version 5.0 | Old | Name |
| IOC\$GQ_SRPIQ | IOC\$GL_SRPFL | Nonpaged pool SRP lookaside list |
| IOC\$GQ_LRPIQ | IOC\$GL_LRPFL | Nonpaged pool LRP lookaside list |
| IOC\$GQ_IRPIQ | IOC\$GL_IRPFL | Nonpaged pool IRP lookaside list |

G.3.6 Miscellaneous Conversion Tasks

This section describes those activities performed by some kernel-mode code threads that should be examined in the course of converting them to run in a VMS multiprocessing environment.

G.3.6.1 Reading the System Time

As discussed in Section 3.1.3, because EXE\$GQ_SYSTIME can only be changed or compared with multiple instructions, any code thread in a multiprocessing system that must obtain a consistent copy of the quadword must first acquire proper synchronization.

VMS Version 5.0 supplies the READ_SYSTIME macro to simplify this procedure. It has the following format, where **dst** is the quadword destination where the macro returns the system time:

```
READ_SYSTIME dst
```

Use of the READ_SYSTIME macro is subject to the following restrictions:

- IPL must be less than 23.

² System cell EXE\$AR_SYSTEM_PRIMITIVES contains the address of this image; the macro \$\$SYSTEM_PRIM_DATADEF in SYS\$SYSTEM:LIB.MLB defines offsets into its data area.

VMS Version 5.0 and Kernel-Mode Code

G.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

- The processor must be executing in kernel mode.
- When using the macro within pageable program sections executing at IPL 2 and below, you must ensure that the pages involved are locked in memory.

G.3.6.2 Calling the Driver Fork Process from a TQE

Whenever VMS places a driver fork process into execution, it ensures that it is synchronized with other processes at that fork level. In other words, if it is generated by the conclusion of I/O preprocessing (EXE\$INSIOQ), the completion of a previous I/O request on a device unit (IOC\$REQCOM), or the operation of the fork dispatcher, the driver fork process is placed into execution at the correct fork IPL, holding the corresponding fork lock.

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

G.3.6.3 Invalidating Translation Buffer Entries

Prior to VMS Version 5.0, privileged code that changed a valid page-table entry (PTE) could flush the stale PTE from the processor's translation buffer by using the INVALID macro or writing directly to the Translation Buffer Invalidate Single (TBIS) processor register. Similarly, it could invalidate the entire translation buffer by using the INVALID macro or writing to the Translation Buffer Invalidate All (TBIA) processor register.

In a VMS Version 5.0 symmetric multiprocessing environment, processors must not use previously buffered PTE contents while another processor is changing that PTE. Once the PTE has been changed, other processors must flush the stale translation buffer entry for the PTE. To accomplish this, VMS has replaced the INVALID macro with the INVALIDATE_TB macro.

The INVALIDATE_TB macro flushes a single PTE or all PTEs from the processor translation buffers in either a VAX uniprocessor or multiprocessor system. In updating privileged code for VMS Version 5.0, you must replace any instances of an INVALID macro or of an MTPR instruction to PR\$_TBIS or PR\$_TBIA with a suitable invocation of the INVALIDATE_TB macro.

Appendix B contains a description of the INVALIDATE_TB macro.

G.3.6.4 Unsupported Use of the IRP

The VMS multiprocessing code employs a portion of the IRP (the 24 bytes following IRP\$_KEYDESC), previously used only as a fork block by the VMS disk and tape class drivers, to effect the transfer of an I/O request from a processor with no access to the device to another processor that does have access.

VMS Version 5.0 and Kernel-Mode Code

G.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

A driver that uses this portion of the IRP to store data can lose this data when the VMS I/O initiation routine (IOC\$INITIATE) attempts to transfer the request to the driver's start-I/O routine. VMS I/O initiation occurs when an FDT routine calls EXE\$QIODRVPKT or when the driver issues the REQCOM macro to complete the current I/O request.

G.3.7 Troubleshooting a Device Driver in a Multiprocessing System

If the full-checking synchronization image has been loaded into memory, the spin lock acquisition and releasing routines perform certain activities that aid in the debugging and tuning of a VMS multiprocessing system. These activities include the following:

- Enforcement of the spin lock ranking and IPL requirements. The means by which the multiprocessing synchronization routines accomplish this are discussed in Section G.3.7.1.
- Recording, for each spin lock, the last eight PCs that acquired or released the spin lock. These PCs are located at offset SPL\$_OWN_PC_VEC in the spin lock data structure (SPL). You can use the SDA command SHOW SPINLOCKS/FULL to display the contents of the PC list.
- Tallying, for each spin lock, the number of successful acquisitions and the number of failed acquisitions in SPL\$_ACQ_COUNT and SPL\$_BUSY_WAITS, respectively.

Section G.1.1 explains the settings of the MULTIPROCESSING system parameter that produce the full-checking synchronization environment.

The full-checking synchronization environment contains a mechanism for producing bugcheck messages that describe the detection of serious synchronization problems in the system. In most instances, these problems are caused by a non-DIGITAL-supplied device driver that does not adhere to multiprocessing synchronization rules.

This section describes the bugchecks that are possible in a VMS full-checking synchronization environment and the SDA commands that aid in the investigation of a multiprocessing system failure. It concludes with a brief description of VMS Version 5.0 changes to the XDELTA debugger.

G.3.7.1 Multiprocessing Bugchecks

In order to obtain a spin lock or fork lock, a processor must be executing at an IPL no higher than the lock's synchronization IPL (SPL\$_IPL). Additionally, the processor cannot obtain a spin lock or fork lock if the lock's rank (SPL\$_RANK) is lower than that of any locks the processor currently holds. To release a spin lock, a processor must be executing at or above the IPL at which it originally acquired the lock. However, a processor can release spin locks in any order of rank. (See Table 3-3 for additional information on spin lock IPL and rank requirements.)

In a full-checking synchronization environment, violation of spin lock synchronization will produce the bugchecks described in Table G-4.

VMS Version 5.0 and Kernel-Mode Code

G.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

Table G-4 Bugchecks Produced Within Full-Checking Synchronization

| | |
|------------|---|
| SPLIPLHIGH | <p>A processor has attempted to acquire a spin lock at an IPL higher than the IPL associated with spin lock synchronization (SPL\$B_IPL). SMP\$ACQUIRE (called by the LOCK and FORKLOCK macros with condition=NOSETIPL not specified) signals this bugcheck.</p> <p>A processor has attempted to acquire a device lock—not already owned by the acquiring processor—at an IPL higher than the IPL associated with device lock synchronization (SPL\$B_IPL). SMP\$ACQUIREL (called by the DEVICELOCK macro with condition=NOSETIPL not set) signals this bugcheck.</p> |
| SPLIPLLOW | <p>A processor has attempted to unconditionally or conditionally release a spin lock or device lock at an IPL lower than the IPL at which it originally acquired it. SMP\$RELEASE and SMP\$RESTORE (called by the UNLOCK and FORKUNLOCK macros) and SMP\$RELEASEL or SMP\$RESTOREL (called by the DEVICEUNLOCK macro) signal this bugcheck.</p> |
| SPLACQERR | <p>A processor has attempted to acquire a spin lock while holding a higher ranked spin lock. SMP\$ACQUIRE, SMP\$ACQUIREL, and SMP\$ACQNOIPL (called by the LOCK, FORKLOCK, and DEVICELOCK macros) signal this bugcheck.</p> |
| SPLRELEERR | <p>An attempt has been made to completely release a spin lock not owned by the releasing processor. SMP\$RELEASE and SMP\$RELEASEL (called by the UNLOCK, FORKUNLOCK, and DEVICEUNLOCK macros) signal this bugcheck.</p> |
| SPLRSTERR | <p>An attempt has been made to conditionally release a spin lock not owned by the releasing processor. SMP\$RESTORE and SMP\$RESTOREL (called by the UNLOCK, FORKUNLOCK, and DEVICEUNLOCK macros when condition=RESTORE is specified) signal this bugcheck.</p> |

G.3.7.2 Analyzing a Multiprocessing System Failure

When invoked to analyze either a crash dump or a running system, the VMS System Dump Analyzer (SDA) establishes a default context for itself from which it interprets certain commands.

When the subject of analysis is a VMS uniprocessing system, SDA's context is solely *process context*. That is, SDA can interpret its process-specific commands in the context of either the process current on the uniprocessor or some other process in some other scheduling state. When initially invoked to analyze a crash dump, SDA's process context defaults to the process that was current at the time of the crash. When invoked to analyze a running system, process context is initially that of the current process: that is, the one executing SDA. Change SDA's process context by entering commands in any of the following forms:

```
SET PROCESS/INDEX=nn
SET PROCESS name
SHOW PROCESS/INDEX=nn
```

When invoked to analyze a crash dump from a VMS multiprocessing system with more than one active CPU, SDA maintains a second dimension of context—its *CPU context*—that allows it to display certain processor-specific information, such as the reason for the bugcheck exception, the currently executing process, the current IPL, the contents of processor-specific registers, the interrupt stack pointer (ISP), and the spin locks owned by the processor. When invoked to analyze a multiprocessor's crash dump, the SDA CPU context defaults to that of the processor that induced the system failure.

Note: When you use SDA to analyze a running system, CPU context is not accessible to SDA. As a result, the SET CPU and SHOW CPU commands are not permitted.

VMS Version 5.0 and Kernel-Mode Code

G.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

Change the SDA CPU context by using any of the following commands:

```
SET CPU cpu-id
SHOW CPU cpu-id
SHOW CRASH
```

Changing CPU context involves an implicit change in process context in one of the following ways:

- If there is a current process on the CPU made current, SDA process context is changed to that of the CPU's current process.
- If there is no current process on the CPU made current, SDA process context is undefined and no process-specific information is available until SDA process context is set to that of a specific process.

Changing process context can involve a switch of CPU context as well. For instance, if you enter a SET PROCESS command for a process that is current on another CPU, SDA will automatically change its CPU context to that of the CPU on which the process is current. The following commands can have this effect if the **name** or index number (**nn**) refer to a current process.

```
SET PROCESS name
SET PROCESS/INDEX=nn
SHOW PROCESS name
SHOW PROCESS/INDEX=nn
```

G.3.7.2.1 Investigating the Status of Spin Locks

SDA in VMS Version 5.0 includes the command SHOW SPINLOCKS. The SHOW SPINLOCKS command displays various levels of information about system spin locks, fork locks, and device locks that help investigations of system failures caused by synchronization violations.

For each spin lock, fork lock, or device lock in the system, SHOW SPINLOCKS provides the following information:

- Name of the spin lock (or device name for the device lock)
- Address of the spin lock (SPL) structure
- The owner CPU's CPU ID
- IPL at which allocation of the lock is synchronized on a local processor
- Number of nested acquisitions of the spin lock by the processor (depth of ownership)
- Rank of the spin lock
- Number of processors waiting to obtain the spin lock
- Spin lock index

SHOW SPINLOCKS/BRIEF produces a condensed display of this same information.

If the VAX system under analysis had been executing with full-checking synchronization enabled (that is, with the MULTIPROCESSING system parameter set to 1 or 2), SHOW SPINLOCKS/FULL adds to the spin lock display the last eight PCs at which the lock was acquired or released.

VMS Version 5.0 and Kernel-Mode Code

G.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

G.3.7.3 Using XDELTA on SMP Systems

Only one processor in a VMS multiprocessing environment can be in XDELTA at a time. If one processor attempts to enter XDELTA while another processor is using XDELTA, it waits until the other processor has exited XDELTA. If the processor using XDELTA sets a breakpoint, other SMP processors are aware of the breakpoint. Therefore, when the code with the XDELTA breakpoint is executed on another processor, that processor will stop at the specified breakpoint and wait to enter XDELTA.

XDELTA uses its own system control block (SCB) to direct all interrupt handling to an error handling routine in XDELTA. Therefore, an error encountered by XDELTA will not affect any of the other processors which share the standard system SCB.

G.4 Multiprocessing Implementation Details

In order to develop or maintain code that interacts closely with the operating system, a system programmer should be aware of certain aspects of the underlying operation of VMS that have been altered or introduced in VMS Version 5.0. The implementation of symmetric multiprocessing has its greatest effect on non-DIGITAL-supplied device drivers that must be adapted to run in a VMS multiprocessing environment.

This section discusses the following operating system concepts, focusing on the effects of the Version 5.0 implementation of multiprocessing:

- Processor states and state transitions
- Initialization sequence of a VMS multiprocessing system
- Process scheduling
- System timekeeping

For information about multiprocessing synchronization and data structures, see Chapter 3 and Appendix A, respectively.

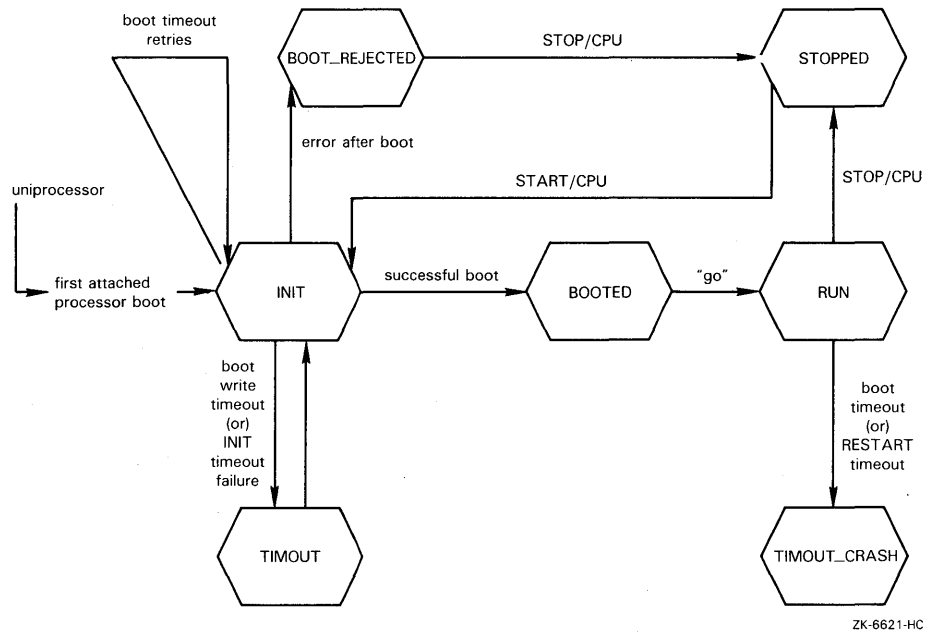
G.4.1 Processor States

A VMS multiprocessing system can be in one of several defined states, as illustrated in Figure G-1.

VMS Version 5.0 and Kernel-Mode Code

G.4 Multiprocessing Implementation Details

Figure G-1 Multiprocessor State Transitions



VMS boots initially as a uniprocessor and later creates a multiprocessing environment as part of system initialization (subject to the system parameters MULTIPROCESSING and SMP_CPUS). The DCL command START/CPU adds one or more available, inactive processors to the multiprocessing configuration. VMS stores a processor's state (and other processor-specific data) in its per-CPU database structure. The DCL command SHOW CPU retrieves this information for display.

Table G-5 describes the states defined for processors in a VMS multiprocessing system. State transitions are depicted in Figure G-1.

Table G-5 Multiprocessor States

| State | Description |
|--------------|--|
| Uniprocessor | The multiprocessing system implicitly begins processing as a uniprocessor prior to any attempt to bring a secondary processor into the system. |
| INIT | A secondary processor enters the INIT state during system initialization if the SMP_CPUS system parameter indicates that this processor is to be booted. Alternatively, a processor enters the INIT state when it is the object of a START/CPU command. Should the bootstrap operation fail, a timeout mechanism causes VMS to retry the operation a defined number of times. If it has not successfully bootstrapped after these attempts, the secondary processor enters the TIMOUT state. |
| TIMOUT | A secondary processor enters the TIMOUT state when it fails to boot after a defined number of retries. If it is subsequently the object of a START/CPU command, the secondary enters the INIT state for another series of retries of the bootstrap operation. |

VMS Version 5.0 and Kernel-Mode Code

G.4 Multiprocessing Implementation Details

Table G-5 (Cont.) Multiprocessor States

| State | Description |
|---------------|--|
| RUN | A secondary processor enters the RUN state when it successfully boots at the request of the primary processor and passes a series of processor validation checks. The primary processor itself is always in the RUN state. RUN is the normal state for all processors running in a multiprocessing environment. A processor in the RUN state is a formal member of the active set, actively participating in operating system activity in concert with other members of the active set. A secondary processor may exit the RUN state if it is the object of a STOP/CPU command. Additionally, a secondary can exit the RUN state if the primary processor requests a machine restart and the restart mechanism fails to restart the secondary. |
| BOOTED | A secondary processor enters the BOOTED state when, after completing all normal operations to enter the RUN state, it must wait for the primary processor to complete the initialization of the multiprocessing environment. When in the BOOTED state, the secondary processor is running, but has not yet been allowed by the primary to join the active set. |
| BOOT_REJECTED | A secondary processor enters the BOOT_REJECTED state upon successfully booting at the request of the primary, but failing the series of processor validation checks that ensure that all processors in a multiprocessing system are at equivalent hardware and firmware revision levels. To force its exit from the BOOT_REJECTED state, you must issue a STOP/CPU for its CPU ID prior to powering it down for repair. |
| STOPPED | A secondary processor enters the STOPPED state when it is the object of a STOP/CPU command. You can issue the STOP/CPU command only for a processor in the RUN or BOOT_REJECTED state. A subsequent START/CPU command causes the secondary processor to enter the INIT state. |
| TIMOUT_CRASH | When a secondary processor in the RUN state is restarting at the request of the primary (for instance, following a system power failure), a successful restart results in no transition from the RUN state. However, a failure to restart the secondary indicates that a critical processor resource has been lost. In this event, the secondary processor enters the TIMOUT_CRASH state and VMS induces a bugcheck. |

G.4.2 System Initialization

A VMS multiprocessing system initially boots as a uniprocessor. At this time, the primary processor is responsible for initializing system memory, loading VMS, and other system tasks. Because the primary CPU is the processor to which the console subsystem is physically or logically connected, it generates the output of all software messages to the console from any processor in the system. The primary CPU is also responsible for keeping system time.

When the primary CPU first enters program mode, the primary bootstrap program (VMB) has been loaded into memory, and its stack pointer register points to the end of a physical page that VMB later formats as a restart parameter block (RPB). The primary CPU uses the end of the RPB as a boot stack until VMS memory management is enabled, when the interrupt stack in its per-CPU database is used. (See Appendix A for additional information on multiprocessing data structures.)

VMS Version 5.0 and Kernel-Mode Code

G.4 Multiprocessing Implementation Details

VMB loads the secondary bootstrap program (SYSBOOT) into memory and transfers control to it. Among its tasks in the creation of a VMS multiprocessing environment, SYSBOOT reserves sufficient system resources for the creation of an individual per-CPU database structure, boot stack, and interrupt stack for each processor in the system's available set; that is, for all processors that can be brought into the system's active set. SYSBOOT also must load an appropriate system synchronization image into memory, based on the setting of the MULTIPROCESSING system parameter for the primary processor. Table G-1 describes the contents of the three possible synchronization images. Table G-2 lists the possible settings of the MULTIPROCESSING system parameter.

SYSBOOT completes its part in system initialization and transfers control to the INIT module of the VMS executive. In addition to its traditional duties, INIT completes the initialization of the primary processor's per-CPU database and calls the SMP\$SETUP_SMP entry point in the system-dependent initialization code (module SYSLOAxxx). This processor-dependent code has the responsibility of completing the multiprocessing initialization of a VAX system.

First, SMP\$SETUP_SMP performs the following general initialization tasks:

- Places the address of the interprocessor interrupt service routine into the system control block (SCB).
- Allocates a page of memory to serve as a *boot page*. It then inserts into this boot page code that is immediately executed by each processor in the system as it comes on line. It also stores data in the boot page required by each booting processor before it enables memory management.
- Establishes a vector containing the physical addresses of all known per-CPU databases. A newly bootstrapped processor accesses this vector as it begins execution in order to locate its per-CPU database area. The physical address of this vector is found in the RPB.
- Determines the processors that are to be brought into the system's active set, using the value of the SMP_CPUS system parameter.

The second phase of system-dependent initialization occurs in the system routine SMP\$SETUP_CPU, which executes once for each processor indicated in the SMP_CPUS system parameter. Also, it is SMP\$SETUP_CPU that executes whenever a START/CPU command attempts to place a secondary processor in the active set.

SMP\$SETUP_CPU performs the following functions:

- Allocates sufficient system resources (such as system page-table entries and page frame numbers) to map the per-CPU database and stacks of the processor.
- Maps the SPTs to the page-frame numbers (PFNs) of the allocated physical memory.
- Initializes as much of the per-CPU database as possible. Later, the newly executing processor will complete this initialization.
- Executes the processor-specific procedure that actually bootstraps the starting processor into the multiprocessing environment.

VMS Version 5.0 and Kernel-Mode Code

G.4 Multiprocessing Implementation Details

As each processor in the VMS multiprocessing system begins execution, it performs the following steps:

- Locates its per-CPU database
- Sets up any immediate context that it requires before enabling memory management
- Enables memory management
- Loads the interrupt stack pointer
- Completes the initialization of its per-CPU database
- Lowers IPL and begins processing as a member of the active set of the VMS multiprocessing system

G.4.3 Scheduling in a VMS Multiprocessing Environment

With VMS Version 5.0, the scheduler no longer schedules a null process. Rather, all idle processors run in an idle loop at IPL 3 on the interrupt stack. Within this loop, the scheduler monitors SCH\$GL_COMQS, the global cell that records the availability of computable inswapped processes. Each set bit in the longword corresponds to a process priority level at which there exist currently computable processes. When a process becomes computable, the bit in SCH\$GL_COMQS corresponding to its priority is updated under the SCHED spin lock.

When an idle processor determines that a computable process exists, it contends for the SCHED spin lock, along with all other idle processors that have noted the change to SCH\$GL_COMQS. Because only one processor at a time can enter the database protected by the SCHED spin lock, work is dispatched to each processor as it is able to secure the SCHED spin lock, until an inswapped computable process is scheduled on each active processor.

While a processor runs in the idle loop, its CPU\$L_CURPCB field points to the null PCB, and its CPU\$B_CUR_PRI field contains 255, representing the highest priority process. This arrangement prevents code in RSE (report system event) from issuing a RESCHED request when a process becomes computable and the processor is actually idle. Thus, a processor issues a RESCHED request only when it must preempt the current process. With VMS Version 5.0, idle processors avoid the overhead involved in rescheduling the null process.

For performance monitoring purposes, the time each processor spends executing on the interrupt stack at IPL 3, with the CPU\$L_CURPCB field pointing to the null PCB, is counted as both null time and interrupt stack time. As recorded in the per-CPU database structure, null time is correct. To determine that portion of interrupt stack time that does not include null time, you must first subtract null time from the interrupt stack time.

As a result of this implementation, VMS Version 5.0 ensures only that the n -highest real-time priority computable processes on an n -processor multiprocessing system will be current simultaneously. However, if all processes are of normal priority, VMS only guarantees that the highest priority normal process capable of running in the system will be scheduled.

VMS Version 5.0 and Kernel-Mode Code

G.4 Multiprocessing Implementation Details

G.4.4 Timekeeping in a VMS Multiprocessing Environment

Each processor in a VMS multiprocessor system has its own interval timer that, every ten milliseconds, causes a hardware interrupt at the interval clock's IPL (either IPL 22 or 24, depending upon processor type). The interval clock interrupt service routine runs on each processor to service these interrupts, but portions of this routine are reserved to run on only one processor in the system, the *primary CPU*. By default, the boot CPU is also the primary CPU, as it is the processor to which the console subsystem is physically attached.

The primary CPU has the sole responsibility of updating system data structures that maintain system time. Similarly, the primary CPU must examine the first entry in the systemwide timer queue to determine whether the first timer queue element (TQE) has expired, and request a software timer interrupt at IPL\$_TIMERFORK (IPL 7) if it has. The IPL\$_TIMERFORK interrupt service routine thereupon removes entries from the head of the timer queue until it has dispatched all due TQEs.

Given this strategy, VMS Version 5.0 supplies two spin locks for the timer queue:

HWCLK Interval clock database spin lock. Locks the interval clock database.
TIMER Software timer database spin lock. Locks the rest of the system timer queue.

The following example describes how VMS uses these two spin locks:

When servicing an interval clock interrupt, the primary CPU performs the following sequence of tasks:

- 1 Obtains the HWCLK spin lock, thus locking the interval clock database
- 2 Advances the system time stored in EXE\$GQ_SYSTIME by ten milliseconds
- 3 Compares the due time of the first TQE (EXE\$GQ_1ST_TIME) with EXE\$GQ_SYSTIME to determine whether the first TQE has expired
- 4 If it has, requests a software interrupt at IPL\$_TIMERFORK
- 5 Releases the HWCLK spin lock, thus unlocking the interval clock database

When servicing a software timer interrupt at IPL\$_TIMERFORK, the primary processor performs the following tasks:

- 1 Obtains the TIMER spin lock, thus locking the software timer database
- 2 Obtains the HWCLK spin lock, thus locking the interval clock database
- 3 Compares the due time of the first TQE (EXE\$GQ_1ST_TIME) with EXE\$GQ_SYSTIME to determine if the first TQE has expired
- 4 If the first TQE has expired, removes it and resets EXE\$GQ_1ST_TIME to reflect the due time of the TQE now at the head of the queue
- 5 Releases the HWCLK spin lock, thus unlocking the interval clock database
- 6 Releases the TIMER spin lock, thus unlocking the software timer database

VMS Version 5.0 and Kernel-Mode Code

G.4 Multiprocessing Implementation Details

When inserting a TQE into the timer queue, any processor performs the following tasks:

- 1** Obtains the TIMER spin lock, thus locking the software timer database.
- 2** Searches the timer queue to determine the proper place to insert the TQE in the queue. (If the TQE belongs at the head of the timer queue, it obtains the HWCLK spin lock, thus locking the interval clock database.)
- 3** Inserts the TQE in the timer queue. (If it has inserted the TQE at the head of the queue, it resets EXE\$GQ_1ST_TIME to reflect the due time of the TQE and then releases the HWCLK spin lock.)
- 4** Releases the TIMER spin lock, thus unlocking the software timer database.

Non-DIGITAL-supplied system code that must access the system time quadword (EXE\$GQ_SYSTIME) must do so after acquiring the HWCLK spin lock. See Section G.3.6.1 for some guidelines on how such code should perform these tasks.

Glossary

ACF: See *configuration control block*.

ACP: See *ancillary control process*.

adapter control block (ADP): A structure in the I/O database that describes an I/O adapter (or VAXBI device) and its resources.

active set: In a VMS symmetric multiprocessing system, those processors that have been bootstrapped into the system, have undergone initialization, and are capable of scheduling and executing processes. Together, the primary processor and all secondary processors make up a system's active set. Compare with *available set*.

ADP: See *adapter control block*.

allocate a device: To reserve a particular device unit for exclusive use. A user process can allocate a device only when that device is not allocated by any other process.

ancillary control process (ACP): A process that acts as an interface between user software and an I/O driver. An ACP provides functions supplemental to those performed by the driver, such as file and directory management.

Examples of ACPs are the magnetic tape ACP (MTAACP) and the network ACP (NETACP).

affinity: In a VMS symmetric multiprocessing system, a close association of a device or a process with a specific processor or set of processors in the system. See *device affinity* and *process affinity*.

assign a channel: To establish the necessary software linkage between a user process and a device unit before a user process can communicate with that device. A user process requests the system to assign a channel and the system returns a channel number.

AST: See *asynchronous system trap*.

ASTLVL: See *asynchronous system trap level*.

asynchronous system trap (AST): A software-simulated interrupt that passes control to a user-defined routine. ASTs enable a user process to be notified of the occurrence of a specific event asynchronously with respect to the execution of the user process.

If a user process has defined an AST routine for an event, the system interrupts the process and executes the AST routine when that event occurs. When the AST routine exits, the system resumes execution of the process at the point where it was interrupted.

Glossary

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 delivered while the processor is executing in a more privileged access mode.

available set: In a VMS symmetric multiprocessing system, those processors that have passed the system's power-on hardware diagnostics and may or may not be actively involved in the system. The available set includes the active set. Compare with *active set*.

ASMP: See *asymmetric multiprocessing*.

asymmetric multiprocessing (ASMP): A multiprocessing configuration in which the processors are not equal in their ability to execute operating system code. In general, a single processor is designated as the primary, or master, processor; other processors are the slaves. The slave processors are limited to performing certain tasks, whereas the master processor can perform all system tasks. Contrast with *symmetric multiprocessing*.

attached processor: See *secondary processor*.

backplane interconnect: An internal processor bus that allows I/O device controllers to communicate with main memory and the central processor. These I/O controllers may reside on the same bus as memory and the central processor (for instance, in a VAX 8200/8250/8300/8350 system), or they may be on a separate bus entirely (for instance, in a VAX 8600/8650/8670 system). In the latter case, an I/O adapter enables and controls the communications between the I/O bus and the processor and memory.

The backplane interconnect is called the synchronous backplane interconnect (SBI) in the VAX-11/780 and VAX 8600/8650/8670 systems, the CPU-to-memory interconnect (CMI) in the VAX-11/750 system, the VAXBI in the VAX 8200/8250/8300/8350 systems, and the memory interconnect (NMI or XMI) in VAX 8530/8550/8700/8800/8830/8840 and VAX 6200-series systems. The MicroVAX II and MicroVAX I processors use the Q22 bus as a backplane.

base register: A general register that contains the base address (the address of the first entry) of a list, table, array, or other data structure.

buffered data path: A UNIBUS adapter data path that transfers several bytes of data in a single backplane-interconnect transfer.

buffered I/O: An I/O operation, such as terminal or mailbox I/O, in which an intermediate buffer from the system's buffer pool is used instead of a buffer in process space. See also *direct I/O*.

bugcheck: The operating system's diagnostic that detects and reports internal inconsistencies. If the system can continue running, it declares a *nonfatal bugcheck* and reports it in an error log entry. A serious error results in a *fatal bugcheck*. As a result of a fatal bugcheck, the system shuts itself down in an orderly fashion.

busy wait: See *spin wait*.

CALL instructions: The processor instructions CALLG (Call Procedure with General Argument List) and CALLS (Call Procedure with Stack Argument List).

capability: In a VMS symmetric multiprocessing environment, an attribute of a single processor or set of processors. The capabilities required by a given process determine the set of processors on which it can be scheduled. For instance, the VMS routine that maintains the system time can execute only on the processor that has the timekeeper capability.

CCB: See *channel control block*.

channel: A logical path connecting a user process to a physical device unit. A user process requests the operating system to assign a channel to a device so the process can communicate with that device. See also *controller data channel*.

channel control block (CCB): A structure in the I/O database maintained by the Assign-I/O-Channel system service to describe the device unit to which a channel is assigned.

channel request block (CRB): A structure in the I/O database that describes the activity on a particular controller. The CRB for a controller contains pointers to the queue of drivers waiting to access a device through the controller.

configuration control block (ACF): A structure in the I/O database used by the autoconfiguration facility of the System Generation Utility to describe the device it is adding to the system. The information stored in the ACF might be useful to a device driver's unit delivery routine.

configuration register: A control and status register for an I/O adapter (for example, a UNIBUS adapter). It resides in the adapter's I/O space.

connect-to-interrupt: A function by which a process connects to a device interrupt vector. To perform a connect-to-interrupt, the process must map to the physical pages in the I/O space that contain the vector.

console: The manual control unit integrated into the central processor. The console includes a serial-line interface connected to a hardcopy terminal. This enables the operator to start and stop the system, monitor system operation, and run diagnostic programs.

console terminal: The terminal connected to the central processor's console.

context: The environment of an activity. See also *process context*, *hardware context*, and *software context*.

controller data channel: A logical path to which the driver of a device that shares a controller must gain access before it can use the controller to activate a device.

control and status register (CSR): A control and status register for a device or controller. It resides in the processor's I/O space.

CRB: See *channel request block*.

CSR: See *control and status register*.

database: A collection of related data structures; all the occurrences of data described by a database management system.

data structure: Any table, list, array, queue, or tree whose format and access conventions are well defined for reference by one or more images.

Glossary

DDB: See *device data block*.

DDT: See *driver dispatch table*.

device affinity: In a VMS symmetric multiprocessing system, a close association of a device with a specific processor or set of processors in the system. There are three dimensions to device affinity in a VMS system. First, physical connectivity describes those devices that are directly accessible only to the primary processor or to all processors. Secondly, affinity is a software mechanism that defines those processors that can initiate an I/O operation on the device. Finally, interruptibility describes the set of processors that can receive interrupts from a device.

device data block (DDB): A structure in the I/O database that identifies the generic device/controller name and driver name for a set of devices that share the same controller.

device driver: The set of instructions and tables that handles physical I/O operations to a device.

device interrupt: An interrupt received on interrupt priority levels 20 through 23. Device interrupts can be requested only by devices, controllers, and memories.

device lock: In a VMS symmetric multiprocessing system, a dynamic spin lock the ownership of which synchronizes device-specific code that executes at device IPL. A device lock is associated with each adapter or controller in the system. See *spin lock*.

device register: A location in controller logic used to request device functions (such as I/O transfers) and/or report status.

device unit: One device and its controlling logic (for example, a disk drive or terminal). Some controllers can have several device units connected to a single controller (for example, mass-storage controllers).

diagnostic program: A program that tests hardware, firmware, peripherals logic, or memory, and that reports any faults it detects.

direct data path: A UNIBUS adapter data path that transfers several bytes of data in a single backplane-interconnect transfer.

direct I/O: An I/O operation in which 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's buffer. Contrast with *system buffered I/O*.

direct-memory-access (DMA) transfer: The type of I/O transfer by which a device controller accesses memory directly and, as a result, can transfer a large amount of data without requesting a processor interrupt after each of the smaller amounts. Contrast with *programmed-I/O (PIO) transfer*.

DPT: See *driver prologue table*.

drive: The electromechanical unit of a mass storage device on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

driver dispatch table (DDT): A table in a driver that lists the addresses of the entry points of standard driver routines and the sizes of diagnostic and error message buffers for the device.

driver prologue table (DPT): A table in a driver that describes the driver and the type of device it controls to the VMS procedure that loads drivers into the system.

dynamic load balancing: A method of work distribution in which the operating system ensures that the system work load is evenly distributed among the processors. Dynamic load balancing in a VMS symmetric multiprocessing system is a direct effect of the implementation of the scheduler. In a VMS multiprocessing system, processors independently and continually look for processes to execute from a common pool of such processes.

ECC: Error-Correction Code.

error logger: A system process that empties the error logging buffers and writes the error messages into the error file. Errors logged by the system include memory errors, device errors and timeouts, and interrupts with invalid vector addresses.

exception: An event detected by the hardware or software (other than an interrupt or jump, branch, case, or call instruction) that changes the normal flow of instruction execution.

An exception is always caused by the execution of an instruction or set of instructions (whereas an interrupt is caused by an activity in the system that is independent of the current instruction).

There are three types of hardware exceptions: traps, faults, and aborts. Examples are attempts to execute a privileged or reserved instruction, trace traps, page faults, compatibility-mode faults, execution of breakpoint instructions, and arithmetic traps.

executive: The software that provides the basic control and monitoring functions of the operating system.

extended QIO processor: The facility that supplements the QIO driver's functions when the driver performs virtual I/O operations on file-structured devices (Files-11 On-Disk Structure Level 2). The XQP executes as a kernel-mode thread in the process of its caller.

FDT: See *function decision table*.

FDT routines: Driver routines called by the \$QIO system service to perform device-dependent preprocessing of an I/O request.

fork block: That portion of a data structure, such as the unit control block, which contains a driver's context while the driver is waiting for an event or a resource. A driver awaiting the processor resource has its UCB fork block linked into a processor-specific fork queue.

fork dispatcher: A VMS interrupt service routine that is activated by a software interrupt on the local processor at a fork IPL. Once activated, it obtains the fork lock associated with the fork IPL and dispatches driver fork processes from a fork queue until no processes remain in the queue for that IPL.

fork lock: In a VMS symmetric multiprocessing system, a static spin lock the ownership of which synchronizes the right of a driver's fork process to execute at its associated fork IPL. See *spin lock*.

Glossary

fork process: A process with a minimal context that executes instructions under a set of constraints: it executes at raised interrupt priority levels; it uses R0 through R5 only (other registers must be saved and restored); it executes in the system's virtual address space; it can refer to and modify static storage that is never modified by procedures that execute at a higher IPL. VMS uses software interrupts, spin locks, fork processes, and resource wait queues to synchronize executive operations.

fork queue: A processor-specific queue of fork blocks that are awaiting activation at a particular IPL by the VMS fork dispatcher.

function code: See *I/O function code*.

function decision table (FDT): A table in the driver that lists all valid function codes for the device and lists the addresses of preprocessing routines associated with each valid function of the device.

function modifier: See *I/O function modifier*.

generic device name: A device name that identifies the type of device but not a particular unit; a device name in which the specific controller and/or unit number is omitted (for example, MB).

hardware context: The values contained in the following registers while a process is executing:

- The PC
- The PSL
- The 14 general registers (R0 through R13)
- The four processor registers (P0BR, P0LR, P1BR and P1LR) that describe the process's virtual address space
- The SP for the access mode in which the processor is executing
- The contents to be loaded in the SP for every access mode other than the current access mode

When a process is executing, its hardware context is continually being updated by the processor. When a process is not executing, its hardware context is stored in its hardware PCB.

hardware process control block (hardware PCB): A data structure known to the processor that contains the hardware context when a process is not executing. A process's hardware PCB resides in its process header (PHD).

IDB: See *interrupt dispatch block*.

interrupt: An event other than an exception or a branch, jump, case, or call instruction that changes the normal flow of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs. See also *device interrupt*, *software interrupt*, and *urgent interrupt*.

interrupt dispatch block (IDB): A structure in the I/O database that describes the characteristics of a particular controller and points to devices attached to that controller.

interrupt priority level (IPL): The level at which a software or hardware interrupt is generated. There are 32 interrupt priority levels: IPL 0 is lowest, 31 is highest. The levels arbitrate contention for processor service. For example, a device cannot interrupt a processor if the processor is currently executing at an IPL greater than the IPL of the device's interrupt request.

interrupt service routine (ISR): A routine executed when a device interrupt occurs.

interrupt stack (IS): The processor-specific stack used when the processor is executing instructions in interrupt context. In the VMS operating system, all hardware interrupts (and all software interrupts above IPL 3) are serviced on a processor-specific interrupt stack and not one of the perprocess stacks.

interrupt stack pointer (ISP): The pointer to the top of the interrupt stack.

interrupt vector: See *vector*.

I/O database: A collection of data structures that describe I/O requests, controllers, device units, volumes, and device drivers in a VMS system. Examples are the driver dispatch table, driver prologue table, device data table, unit control block, channel request block, I/O request packet, and interrupt dispatch block.

I/O driver: See *driver*.

I/O function: An I/O operation interpreted by the operating system and typically resulting in one or more physical I/O operations.

I/O function code: A 6-bit value specified in a \$QIO system service call that describes the particular I/O operation to be performed (such as, read, write, rewind).

I/O function modifier: A 10-bit value specified in a \$QIO system service call that modifies an I/O function code (for example: read terminal input, no echo).

I/O lockdown: The state of a page such that it cannot be paged or swapped out of memory.

I/O request packet (IRP): A structure in the I/O database that describes an individual I/O request. The \$QIO system service creates an IRP for each I/O request. VMS and the driver of the target device use information in the IRP to process the request.

I/O rundown: An operating system function in which the system cleans up any I/O in progress when an image exits.

I/O space: The regions of physical address space that contain the configuration registers and device control and status register and data registers. These regions are physically noncontiguous.

I/O status block (IOSB): A data structure associated with the \$QIO system service. This service optionally returns a status code, number of bytes transferred, and device/function-dependent information in an I/O status block. The information is not returned from the system service call, but filled in by VMS when the I/O request completes.

IPL: See *interrupt priority level*.

IRP: See *I/O request packet*.

Glossary

ISP: See *interrupt stack pointer*.

ISR: See *interrupt service routine*.

limit: The size or number of items requiring system resources (such as mailboxes, locked pages, I/O requests, or open files) that a job is allowed to have at any one time during execution, as specified by the system manager in the user authorization file. See *quota*.

load balancing: A function of the operating system by which work is distributed equally among all processors in a system. For more information, see *static load balancing* and *dynamic load balancing*.

locking a page in memory: Making a page ineligible for either paging or swapping. A page stays locked in physical memory until VMS specifically unlocks it.

logical-I/O function: A set of I/O operations (for example, read-logical-block and write-logical-block) that allow restricted direct access to device-level I/O operations using logical block numbers.

loosely coupled system: A multiprocessing system configuration consisting of separate operating systems that communicate through some message transfer mechanism. Contrast with *tightly coupled system*.

mailbox: A software data structure that is treated as a record-oriented device for interprocess communication (for example, the error logger and OPCOM read from systemwide mailboxes). Communication using a mailbox is similar to other forms of device-independent I/O. Senders write to a mailbox; the receiver reads from that mailbox.

machine check: An exception that is reported when the processor or an external adapter detects an internal error. If the machine check is recoverable, the machine check handler logs the condition in an error log entry. If an unrecoverable machine check occurs while the processor is in supervisor or user mode, the machine check handler reports the exception to that mode. However, if an unrecoverable machine check occurs in kernel or executive mode, a fatal bugcheck results. See also *exception* and *bugcheck*.

map register: See *scatter-gather map*.

MASSBUS adapter (MBA): An interface device between the backplane interconnect and the MASSBUS.

memory interconnect: The name of the internal processor bus for the VAX-11/750 (CMI), VAX 8530/8550/8700/8800/8830/8840 (NMI), and VAX 6200 series (XMI).

multiprocessing system: A system containing two or more general purpose processors. These processors are connected through hardware so that they can work on the same application concurrently. See *asymmetric multiprocessing* and *symmetric multiprocessing*.

multiprogramming: A mode of operation in which hardware resources are shared among multiple, independent software processes.

nexus: A physical connection to the synchronous backplane interconnect (SBI). For example, when connected to the SBI, the central processor, memory subsystem, and I/O controllers are known as nexuses. See also *synchronous backplane interconnect*.

node: A VAXBI interface—such as a central processor, controller, or memory subsystem—that occupies one of 16 logical locations on a VAXBI bus. See also *VAXBI*.

offset: A displacement from the beginning of a data structure to the beginning of a field within that data structure. Offsets for items within a data structure usually have an associated symbol. The name of the symbol is used to refer to the field; its value is the offset.

page-frame number (PFN): The high-order 21 bits of the physical address of a page in physical memory.

page-table entry (PTE): The data structure that identifies the physical location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page-frame number needed to map the virtual page to a physical page. When it is not in memory, the page-table entry contains the information needed to locate the page on secondary storage (disk).

parallel processing: A method of computing that occurs when a section of an application is divided into multiple tasks, and those multiple tasks are executed simultaneously on multiple processors.

PCB: See *process control block*.

PFN: See *page-frame number*.

physical address: The address used by hardware to identify a location in physical memory or on directly-addressable secondary storage devices such as disks. A physical-memory address consists of a page-frame number and the number of a byte within the page. A physical-disk-block address consists of a cylinder or track and a sector number.

physical address space: The set of all possible physical addresses that can be used to refer to locations in memory (memory space) or device registers (I/O space).

physical-I/O functions: A set of I/O functions that allows access to all device-level I/O operations except maintenance-mode operations.

PID: See *process identification*.

primary processor: The processor in a VMS symmetric multiprocessing system that is either logically or physically attached to the console device. Only the primary processor performs the initialization activities that define the VMS environment and prepare memory for the entire system. In addition, the primary processor serves as the system timekeeper.

process: The basic entity, scheduled by the system software, that provides the context in which an image executes. A process consists of an address space, hardware context, and software context.

process affinity: In a VMS symmetric multiprocessing system, a close association of a process with a specific processor or set of processors in the system. Process affinity can be indicated as either a requirement that a process run only on the processor with a specific CPU ID or on a processor or set of processors that have a needed capability. See *capability*.

process context: The hardware and software contexts of a process.

Glossary

process control block (PCB): A data structure used to contain process context. The hardware PCB contains the hardware context. The software PCB contains the software context, which includes a pointer to the hardware PCB.

process identification (PID): A 32-bit value that uniquely identifies a process. Each process has a PID and a name.

process I/O channel: See *channel*.

process page tables: The page tables used to describe process virtual memory.

process priority: The priority assigned to a process for scheduling purposes. The operating system recognizes 32 levels of process priority, where 0 is lowest and 31 is highest. Levels 16 through 31 are used for real-time processes. The system does not modify the priority of a real-time process (although the system manager or the process itself might). Levels 0 through 15 are used for normal processes. The system can temporarily increase the priority of a normal process based on the activity of the process.

Contrast with *interrupt priority level*.

programmed-I/O (PIO) transfer: The type of I/O transfer, largely conducted by the driver program, that requires processor intervention after each byte or word is transferred. Drivers for relatively slow devices, such as printers, card readers, terminals, and some disk and tape drives use PIO data transfers. Contrast with *direct-memory-access (DMA) transfer*.

program section (psect): A portion of a program with a given protection and set of storage-management attributes. Program sections that have the same attributes are gathered together by the linker to form an image section.

PTE: See *page-table entry*.

Q22 bus: The hardware interconnect by which MicroVAX-3600 series, MicroVAX II, and MicroVAX I peripheral devices communicate with main memory and the processor.

QIO: Queue I/O Request system service. The VMS system service that services \$QIO and \$QIOW requests. The Queue I/O Request system service prepares an I/O request for processing by the driver and performs device-independent preprocessing of the request. This system service also calls driver FDT routines. See also *FDT routines*.

quota: The total amount of a system resource, such as CPU time, that a job is allowed to use in an accounting period, as specified by the system manager in the user-authorization file. See *limit*.

return status code: See *status code*.

SBI: See *synchronous backplane interconnect*.

scatter-gather map: A technique by which a set of physically discontinuous pages are made to seem contiguous to an I/O controller performing a direct-memory-access transfer. It is I/O adapter hardware that generally provides this means of mapping physical pages to I/O adapter address space.

secondary processor: The processor or processors in a VMS symmetric multiprocessing system that do not have the initialization and timekeeper responsibilities of the primary processor.

small process: A system process that has no control region in its virtual address space and has an abbreviated context. Examples are the swapper and the null process. A small process is scheduled in the same manner as user processes, but must remain resident until it completes execution; it cannot be swapped.

shared memory: A generic term referring to any memory that can be accessed by two or more concurrent processes. In a VMS symmetric multiprocessing system, a single copy of the VMS operating system resides in memory. Each processor in the system can access this memory, as can any process executing on any processor.

SMP: See *symmetric multiprocessing*.

software context: The context maintained by VMS to describe a process. See also *software process control block (PCB)*.

software process control block (software PCB): The data structure used to contain a process'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 the process is swapped out of memory; unique identification of the process; and address of the process header (which contains the hardware PCB). The software PCB resides in the system region of virtual address space. It is not swapped with a process.

start-I/O routine: The routine in a device driver that is responsible for obtaining needed resources and for activating the device unit. An example of a needed resource is the controller's data channel.

spin lock: In a VMS symmetric multiprocessing system, a semaphore associated with a set of system structures, fields, or registers whose integrity is critical to the performance of a specific operating system task. There are two types of spin lock. Static spin locks are assembled permanently into the system; the same static spin locks exist in the same memory locations in all VMS multiprocessing systems. A fork lock is a form of static spin lock. Dynamic spin locks are created as required by the I/O configuration of a system; as a result, the set of dynamic spin locks differs from processor to processor. A device lock is a form of dynamic spin lock. See *fork lock* and *device lock*.

spin wait: In a VMS symmetric multiprocessing system, an execution loop performed by a processor attempting to acquire a spin lock already owned by another processor in the system. This activity is also known as a busy wait.

static load balancing: A method of work distribution in which every process in an application is preassigned to a processor during process creation.

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*.

Glossary

symmetric multiprocessing (SMP): A multiprocessing system configuration in which all processors have equal access to operating system code residing in shared memory and can perform all, or almost all, system tasks.

synchronous backplane interconnect (SBI): The part of the VAX-11/780, VAX-11/785, and VAX 8600/8650/8670 hardware that interconnects the processor, memory controllers, MASSBUS adapters, and the UNIBUS adapter.

System Page Table (SPT): The data structure that maps the system virtual addresses, including the addresses used to refer to the process page tables. The SPT contains one PTE for each page of system virtual memory. The physical base address of the SPT is contained in a processor register called the System Base Register (SBR).

system virtual address (SVA): A virtual address identifying a location mapped to an address in system space.

tightly coupled system: A multiprocessing system configuration consisting of multiple processors sharing a single copy of the operating system. These processors are connected so that they can communicate and share data. Contrast with *loosely coupled system*.

timeout: The expiration of the time limit in which a device is to complete an I/O transfer. The driver's wait-for-interrupt request specifies the timeout limit.

timer: A system process that maintains the time of day and the date. It is also alert for device timeouts and performs time-dependent scheduling upon request. The timer's interrupt service routine creates the timer process.

UCB: See *unit control block*.

UNIBUS adapter: An interface device between the backplane interconnect and the UNIBUS. In a VAX-11/780, VAX-11/785, or VAX 8600/8650/8670 system this device is called the UBA. In a VAX-11/750 system, it is called the UBI. In a VAX 8200/8250/8300/8350 or VAX 8530/8550/8700/8800 system, it is called a DWBUA.

unit control block (UCB): A structure in the I/O database that describes the characteristics of a device unit and current activity on it. The unit control block also holds the fork block for its unit's device driver; the fork block is part of the UCB and is a critical part of a driver fork process. The UCB also provides a static storage area for the driver.

unit initialization routine: The routine that readies controllers and device units for operation. Controllers and device units require initialization after a power failure and during execution of the driver-loading procedure.

urgent interrupt: An interrupt received on interrupt priority levels 24 through 31. These can be generated only by the processor for the interval clock, serious errors, and power failures.

VAXBI: The part of the VAX 8200/8250/8300/8350 hardware that connects I/O adapters with memory controllers and the processor. In a VAX 8530/8550/8700/8800 system, the part of the hardware that connects I/O adapters with the bus that interfaces with the processor and memory.

vector: A one-dimensional array.

An interrupt or exception vector is a storage location known to the system that contains the starting address of a routine to be executed when a given interrupt or exception occurs. The system defines separate vectors for each interrupting adapter and for classes of exceptions. Each system vector is a longword.

For the purpose of handling exceptions, users can declare up to two software-exception vectors (primary and secondary) for each of the four processor-access modes. Each vector contains the address of a condition handler, and is a longword.

virtual-I/O functions: A set of I/O functions that must be interpreted by an ancillary control process.

wait-for-interrupt request: A request made by a driver's start-I/O routine after it activates a device. The request causes the driver's fork process to be suspended until the device requests an interrupt or the device times out.

XDELTA: A software tool for debugging the VMS operating system and device drivers.

XQP: See *extended QIO processor*.

Index

A

Aborting an I/O request

See I/O request

ACB\$_QUOTA • C-7, C-10

ACB (AST control block) • 4-18, A-38, A-64,
C-2, C-4

contents • C-6

Accessibility of memory

See Buffer

Access violation

See SS\$_ACCVIO

ACF (configuration control block) • A-2 to A-4

ACL (access rights list) • A-45

ACP (ancillary control process) • A-11, A-38,
A-39, A-52

See also XQP

class • A-27

default • A-27

ACP_MULTIPLE parameter • A-27

Action routine

See FDT routine

Action routine bit mask • 4-10

Active set • G-23

Adapter

See I/O adapter

Adapter control block

See ADP

Adapter dispatch table • 12-27, 12-31, A-6, A-7

address • A-6

examining • 16-8 to 16-9

ADP\$_AVECTOR • 14-8

ADP\$_BIMASTER • 14-8, 14-15

ADP\$_BI_IDR • 14-8, 14-12

ADP\$_CSR • 14-8, C-79

ADP\$_DPQFL • C-84, G-14

ADP\$_MBASCB • 14-8, A-7

ADP\$_MBASPTTE • 14-8, A-7

ADP\$_MR2QFL • G-14

ADP\$_MRQFL • G-14

ADP\$_VECTOR • 12-31

ADP\$_W_ADPTYPE • 14-8, B-3

ADP\$_W_BI_VECTOR • 14-8, 14-13

ADP\$_W_DPBITMAP • 12-17, C-93

ADP\$_W_TR • 14-8, 14-15

ADP\$_W_XBIA_TR • 14-15

ADP (adapter control block) • 1-6, 12-15,
A-4 to A-10

address • 4-5, 12-17, 12-19, 12-31, A-24,
A-35

alternate map register allocation information •
A-10

alternate map register wait queue • A-9

data path allocation information • 12-17, A-9

data path wait queue • 12-17, A-7

fields supporting ADPDISP macro • B-3

for generic VAXBI device • 14-8

for MBA • 13-4, 13-6 to 13-7

for VAXBI adapter • 14-8

map register allocation information • A-9

map register wait queue • A-8

size • A-5

ADPDISP macro • 5-5, B-2 to B-4

examples • B-4

Affinity

See Device affinity

Alignment of data transfer • 12-3

Allocation class • A-27

Alternate map registers • 12-2, 12-5, 12-22,
A-8, A-25, B-3

See also Map registers

allocating • 12-19, C-61 to C-62

allocating permanent • 11-2, 12-20, A-25

loading • 12-22, B-41, C-72 to C-73

number of active • A-10

number of disabled • A-10

releasing • 12-26, B-48, C-81 to C-82

requesting • B-53, C-89 to C-90

Alternate map register wait queue • A-9, C-90,
G-14

Alternate start I/O routine • 7-5, C-17

address • 6-4, A-29, D-2

context • D-2

entry point • D-2

exit method • D-2

input • D-2

register usage • D-2

synchronization requirements • D-2

ARB (access rights block) • 4-8, A-41

AST (asynchronous system trap) • C-6 to C-7

See also Attention AST

control • A-64

Index

AST (asynchronous system trap) (cont'd.)
 delivering • 3–4, C–2, C–11
 for aborted I/O request • C–11
 out of band • 11–7, A–64
 process-requested • 4–18, C–7, C–10, C–71
 queuing • 3–4, C–71
 special kernel-mode • 3–4, 4–17, 7–7,
 7–7 to 7–8, A–11
 user specified • A–38
AST control block
 See ACB
ASTLVL (AST level) processor register • 3–4
AST procedure (for connect to interrupt facility) •
 18–18
AST service routine (for connect to interrupt
 facility) • 18–8, 18–10, 18–12
AT\$_MBA • A–32
AT\$_UBA • A–32
Attached processor
 See Secondary processor
Attention AST
 See also AST
 blocking • A–61, A–62
 delivering • C–2
 disabling • C–6 to C–7
 enabling • C–6 to C–7
 flushing • C–4
Attention condition • 13–8 to 13–9
 See also MBA, MBA\$_AS, MASSBUS
Attention summary register
 See MBA\$_AS
Autoconfiguration
 See also System Generation Utility
 driver control of • 15–17 to 15–18

B

Backplane interconnect • 1–11, 1–15, 12–1
 See also VAXBI, CMI, SBI, Q22 bus
Backplane interconnect interface chip
 See BIIC
BADDALRQSZ bugcheck • C–3, C–19
BI
 See VAXBI bus
BIIC
 CSR space • 14–5
BIIC\$_BCICR • 14–13, 14–26 to 14–27
BIIC\$_BER • 14–6, 14–12, 14–13,
 14–24 to 14–25

BIIC\$_BICSR • 14–11, 14–23 to 14–24
BIIC\$_DTREG • 14–6, 14–23
BIIC\$_EAR • 14–26
BIIC\$_EICR • 14–9, 14–13, 14–25 to 14–26
BIIC\$_GPRO • 14–28
BIIC\$_GPR1 • 14–28
BIIC\$_GPR2 • 14–28
BIIC\$_GPR3 • 14–28
BIIC\$_IDR • 14–12, 14–26
BIIC\$_IPIDR • 14–26
BIIC\$_IPIMR • 14–26
BIIC\$_IPIISR • 14–26
BIIC\$_IPISTPF • 14–27
BIIC\$_SAR • 14–26
BIIC\$_UICR • 14–9, 14–13, 14–27 to 14–28
BIIC\$_WSR • 14–27
BIIC\$_V_ARBCNTRL • 14–11
BIIC\$_V_BROKE • 14–11
BIIC\$_V_SST • 14–11
BIIC\$_V_STS • 14–11
BIIC (backplane interconnect interface chip) • 14–5
 clearing error register • 14–12
 enabling error interrupts • 14–13, 14–25
 enabling options • 14–13
 initializing • 11–2
 self test • 14–11
 setting interrupt vectors • 14–13
\$BIICDEF macro • 14–5, 14–21
BIIC registers
 accessing • 14–5
 symbolic names • 14–21 to 14–28
BIOCNT (buffered I/O count) • 2–3
BIOLM (buffered I/O limit) quota
 adjusting • 4–17
 charging • 4–7, 4–10
 checking • 4–7
 for mailbox • A–52
BIRQ level • 12–34, 12–35
BI-to-UNIBUS adapter
 See DWBUA
BOOTED processor state • A–15, G–22
Booting with XDELTA • 16–1 to 16–5
Boot page • G–23
Boot stack • A–14
BOOT_REJECTED processor state • A–15, G–22
BPT (Breakpoint) instruction • 16–6
Breakpoint
 clearing • 16–18
 complex • 16–18
 displaying XDELTA breakpoint list • 16–18
 proceeding from • 16–5, 16–18

- Breakpoint (cont'd.)
 - setting in driver code • 16-6, 16-10, 16-17
- BREAKPOINTS parameter • 16-1, 16-5
- BR level • 12-34
 - relation to SCB vectors • A-9
- Buffer
 - allocating • 1-18, 2-3, 7-6 to 7-7, C-12 to C-13, C-14, C-15, C-22 to C-23, G-5
 - allocating a physically contiguous • 12-26, C-16
 - data area • 7-6
 - deallocating • 2-7, 4-17, 7-7, C-3, C-19
 - format • 7-7
 - header area • 7-6, 7-7
 - locking • 1-18, 6-7, A-41, A-42, C-31 to C-33, C-34 to C-36, C-40 to C-42, C-45 to C-47, C-53 to C-54, C-57 to C-58
 - locking multiple areas • C-34, C-45, C-57
 - moving data from system to user • C-78
 - moving data from user to system • C-77
 - size • 7-6, 12-26
 - storing address of • 7-6
 - testing accessibility of • 7-6, B-36 to B-37, C-31 to C-33, C-34 to C-36, C-40 to C-42, C-43 to C-44, C-45 to C-47, C-53 to C-54, C-55 to C-56, C-57 to C-58
 - unlocking • C-105
- Buffer address register • 12-22
- Buffered data path • 12-8, A-8
 - See also Data path
 - allocating permanent • 11-2, 12-18, A-24, G-12
 - flow of read operation using • 12-12 to 12-13
 - flow of write operation using • 12-12
 - functions • 12-11
 - odd transfer • A-8
 - purging • 12-13, 12-19, 12-24 to 12-25, C-79 to C-80
 - releasing • 10-2, 12-19, 12-25, B-50, C-84
 - requesting • 12-11, 12-17 to 12-18, B-55, C-93 to C-94
 - rules for using • 12-11, 12-14
 - speed • 12-14
- Buffered data path wait queue
 - See Data path wait queue
- Buffered function bit mask • 4-9, 6-7
- Buffered I/O • 1-18, 2-3, 4-9, 11-6, 14-16, A-39, A-40, A-58
 - chained • A-39
- Buffered I/O (cont'd.)
 - complex • A-39
 - FDT routines for • 7-6 to 7-8
 - functions • 6-4
 - postprocessing • 7-7 to 7-8, C-70
 - reasons for using • 1-18, 6-7
- Buffered read function bit
 - See IRP\$V_FUNC
- Bugcheck • 16-20
 - BADDALRQSZ • C-3, C-19
 - examining information regarding • 16-5
 - ILLQBUSCFG • A-20
 - INCONSTATE • C-85, C-94
 - SPLACQERR • 16-25, 16-26, C-107, G-18
 - SPLIPLHIGH • 16-25, C-107, C-108, G-18
 - SPLIPLLOW • 16-25, C-109, C-110, C-111, C-112, G-18
 - SPLRELEERR • 16-25, 16-26, C-109, C-110, G-18
 - SPLRSTERR • 16-25, 16-26, C-111, C-112, G-18
 - UBMAPEXCED • C-73, C-76
 - UNSUPRTCPU • B-9
- BUGREBOOT parameter • 16-2, 16-5, 16-20
- Bus grant • 12-34, 12-35
- Bus request
 - See BR level, BIRQ level
- Busy bit
 - See UCB\$V_BSY
- BYTCNT (byte count) quota • 3-12
 - checking • G-5
 - crediting • C-18, G-5
 - debiting • C-12, C-20 to C-21, C-22 to C-23, G-5
 - system maximum • C-20, C-22
 - verifying • C-20 to C-21, C-22 to C-23
- Byte count quota
 - See BYTCNT
- Byte count register
 - See MBA\$L_BCR
- Byte limit
 - See BYTLM
- Byte offset register • 12-13
- BYTLM (byte limit)
 - crediting • G-5
- BYTLM (byte limit) quota • 3-12
 - checking • G-5
 - crediting • C-18
 - debiting • C-12, C-20 to C-21, C-22 to C-23, G-5

Index

C

- Cache control block • A-62
- Caching • A-54
- CAN\$C_CANCEL • 11-7
- CAN\$C_DASSGN • 11-7
- Cancel I/O bit
 - See UCB\$V_CANCEL
- Cancel I/O routine • 1-4, 9-7, 11-6 to 11-8, A-29
 - address • 6-3, 11-1, D-3
 - context • 11-7, D-4
 - device dependent • 11-8
 - device independent • 11-8
 - entry point • D-3
 - exit method • D-4
 - flushing ASTs in • C-4
 - for connect to interrupt facility • 18-8, 18-10, 18-17 to 18-18
 - input • D-4
 - of CONINTERR.EXE • 18-12, 18-17
 - register usage • D-4
 - synchronization requirements • D-4
 - when unneeded • 11-7
- \$CANDEF macro • 11-7
- Card reader • A-54
 - device driver • 9-6 to 9-8
- Carriage control • A-53
- CASE macro • B-5
 - example • B-5
- CCB\$B_AMOD • C-100
- CCB\$L_UCB • 4-4
- CCB (channel control block) • 1-6, 4-4, A-11
 - address • C-100
- Channel • 1-6
 - See also Process I/O channel
- Channel control block
 - See CCB
- Channel index number • 4-4, 11-8, C-66, C-100, D-4
- Channel request block
 - See CRB
- Channel wait queue
 - See Device controller data channel wait queue
- CHMK (Change Mode to Kernel) instruction • 4-1
- \$CINDEF macro • 18-10
- Class driver
 - See Terminal class driver
- Class driver entry vector table • A-33
- Class driver vector table • 17-5, A-67
 - address • 17-8, B-7
 - relocating • B-6
- CLASS_CTRL_INIT macro • 17-11, A-67, B-6
- CLASS_DDT vector table entry • 17-18
- CLASS_DISCONNECT service routine • 17-18
- CLASS_DS_TRANS service routine • 17-11, 17-18
- CLASS_FORK service routine • 17-13, 17-18
- CLASS_GETNXT service routine • 17-19, A-67, B-7
 - address • 17-8
- CLASS_POWERFAIL service routine • 17-11, 17-20
- CLASS_PUTNXT service routine • 17-16, 17-19, A-67, B-7
 - address • 17-8
- CLASS_READERROR service routine • 17-16, 17-21
- CLASS_SETUP_UCB service routine • 17-11, 17-20
- CLASS_SET_LINE service routine • 17-11
- CLASS_UNIT_INIT macro • 17-8, 17-11, 17-18, B-7
- Clock
 - See Interval clock
- Cloned UCB routine • 11-11 to 11-12, A-56
 - address • 6-4, A-30, D-5
 - context • D-5
 - exit method • 11-12, D-6
 - input • 11-11, D-5
 - register usage • 11-11, D-5
 - synchronization requirements • D-5
- CMI (CPU-to-memory interconnect) • 1-11
- Coding conventions
 - See Device driver
- COM\$DELATTNAST • C-2
- COM\$DRVDEALMEM • 14-18, C-3
- COM\$FLUSHATTNS • C-4, C-6
- COM\$POST • 3-4, 7-5, C-5, D-2
- COM\$SETATTNAST • C-6 to C-7
- Command address register
 - See MBA\$_CAR
- Configuration register
 - See CSR, MBA\$_CSR
- CONFREGL array • 14-6
- CONINTERR.EXE • 18-7, 18-12 to 18-13
 - cancel I/O routine of • 18-12
 - connecting to • 18-8
- CONNECT command
 - See System Generation Utility

- Connect to interrupt driver
 - See CONINTERR.EXE
- Connect to interrupt facility
 - cancel I/O routine • 18–17 to 18–18
 - condition values returned • 18–11
 - CONNECT command • 18–8
 - example of A/D converter using • 18–18, 18–20 to 18–22
 - example of time sampling using • 18–18, 18–22 to 18–24
 - example of watchdog timer using • 18–18, 18–19 to 18–20
 - interrupt service routine • 18–16 to 18–17
 - mapping I/O address space • 18–7
 - privileges required • 18–11
 - programming language requirements • 18–13
 - start I/O routine • 18–15
 - SYSGEN requirements • 18–8
 - unit initialization routine • 18–14 to 18–15
 - user-specified routines • 18–8, 18–13 to 18–18
- Control and status register
 - See CSR
- Control block
 - See Data structure
- Controller
 - See Device controller
- Controller initialization routine • 1–3, 11–1 to 11–6, 15–4, 15–8
 - address • 4–4, 6–3, 11–1, 12–31, A–23, B–24, D–7
 - allocating controller data channel in • 8–4
 - context • 11–1, D–7
 - entry point • D–7
 - exit method • D–7
 - for generic VAXBI device • 14–10 to 14–15
 - forking • A–19
 - forking in • 3–21, 11–5 to 11–6
 - for terminal port driver • 17–11, B–6
 - functions • 11–1, D–8
 - input • 11–2, D–7
 - register usage • D–7
 - synchronization requirements • D–7, G–12
- Control mask
 - See Device activation bit mask
- Control register
 - See CSR, MBA\$L_CR
- Coroutine • C–35, C–46, C–58, C–105
- Corruption
 - detecting • 16–22 to 16–24
- CPU\$_PHY_CPUID • C–68
- CPU\$_PSBL • C–5, C–10, C–24, C–92
- CPU\$_PSFL • 3–5, C–70, G–15
- CPU\$_SWIQFL • C–26, C–30, G–15
- CPU\$_WORK_IFQ • A–16
- CPU (per-CPU database) • A–12 to A–17
 - creation • G–23
 - locating • B–29, G–7
- CPUDISP macro • 5–5, B–8 to B–9
- CPU ID • A–16, C–68
- CRB\$_MASK • 4–4, 14–7
- CRB\$_AUXSTRUC • 12–26
- CRB\$_DLCK • 3–20
- CRB\$_INTD • 4–4, A–20 to A–25
- CRB\$_INTD+VEC\$_INITIAL • 11–4
- CRB\$_INTD+VEC\$_UNITINIT • 11–4
- CRB\$_LINK • 13–12
- CRB\$_WQBL • 14–7
- CRB\$_WQFL • 4–4, 14–7, C–83, C–88
- CRB\$_V_UNINIT • 14–7
- CRB (channel request block) • 1–6, 4–4 to 4–5, A–17 to A–25
 - alternate map register allocation information • 12–20
 - creation • 15–4
 - data path allocation information • 12–17 to 12–18
 - for generic VAXBI device • 14–7
 - fork block • 3–21, 15–7, A–19
 - for MBA • 13–4, 13–6 to 13–7, 13–12, 13–14
 - initializing • 6–3, B–24
 - map register allocation information • 12–19 to 12–20
 - periodic wakeup of • A–20
 - primary • 13–12, A–52
 - reinitializing • 6–3, B–24
 - secondary • 13–12, A–20
 - synchronizing access to • 3–15
- CSR
 - fixed space • 15–12
 - floating space • 15–12
- CSR (control and status register) • 12–4, 12–22
 - See also Device registers
 - address • 4–5, 8–4, 12–23, A–35
 - bad address • A–35
 - displaying address • 15–9
 - loading • 8–5
 - locating device registers from • 12–23
 - of LP11 printer • 2–5
 - specifying address • 15–5
 - specifying offset for multiunit controller • 15–5
- CTL\$_GL_CCBASE • C–100

Index

CTL\$GL_PCB • G-7

D

Data path • 1-17, 12-8 to 12-14,
12-17 to 12-19, A-24

See also Buffered data path, Direct data path
autopurging • A-8, B-3

buffered • 12-2, A-8, B-3

direct • B-3

mixed use of direct and buffered • 12-19

purging • 10-2, 12-13, 12-19,

12-24 to 12-25, B-46, C-79 to C-80

speed • 12-10, 12-11, 12-14

Data path allocation bit map • A-9

Data path register • 12-8, 12-15

purge error • C-80

Data path wait queue • 12-25, A-7, C-85, C-94,
G-14

Data storage • 5-1

device specific • 4-4, 11-2, A-40, A-47,
B-20

Data structure • A-1

See also I/O database

defining bit field within • B-70 to B-71

defining field within • B-12, B-13, B-14

initializing • 6-1, B-22 to B-24

Data transfer

See also DMA transfer, PIO transfer
alignment • 12-3

byte aligned • 12-2, 12-22, B-3, C-76

byte count • A-58, A-62

byte offset • 12-13, 12-18, A-58, C-75

in reverse direction • 13-3, 13-13

longword-aligned 32-bit random-access •
12-11

mixing read and write functions in • 12-10

negative byte count • C-32, C-35, C-41, C-43,
C-45, C-54, C-55, C-57

overlapping with seek operation • 8-3

size • 12-23

speed • 12-10, 12-11, 12-14

starting address • 12-22 to 12-23, 12-26,
A-58

to randomly ordered addresses • 12-10

word aligned • 12-2, C-76

zero byte count • C-32, C-41, C-54

\$DCDEF macro • A-54, B-3, B-19

DDB\$_LINK • 11-4

DDB\$_UCB • 11-4

DDB\$_DRVNAME • 4-6

DDB\$_NAME • 4-6

DDB (device data block) • 1-5, 4-6, 11-4,
A-25 to A-27

address • A-52

creation • 15-4

initializing • 6-3, B-24

reinitializing • 6-3, B-24

DDT\$_ALTSTART • 7-5, D-2

DDT\$_CANCEL • D-3

DDT\$_CLONEDUCB • D-5

DDT\$_REGDUMP • D-14

DDT\$_START • D-15

DDT\$_UNITINIT • 11-4, D-21

DDT\$_UNSOLINT • D-23

DDT\$_W_ERRORBUF • 11-9

DDT (driver dispatch table) • 1-2, 11-1, 11-9,
A-27 to A-30, C-99

address • 6-3, A-27, A-58, B-24

creating • 6-3 to 6-4, 11-3, B-10 to B-11

of terminal class driver • 17-18

relocating addresses specified in • 11-4

DDTAB macro • 11-9, 15-1, B-10 to B-11,
C-99

example • B-11

Debugging

device driver • 16-1 to 16-27

\$DEFEND macro • A-48, B-13

example • B-14

\$DEFINI macro • A-48, B-14

example • B-14

\$DEF macro • A-48, B-12

example • B-14

DELTA

See Delta/XDelta Utility

Delta/XDelta Utility (DELTA/XDELTA) •
16-1 to 16-20

base register • 16-13

predefined • 16-13

X4 • 16-13

X5 • 16-13

XE • 16-13

XF • 16-13

changing contents of location using • 16-15,
16-16

closing location using • 16-16

commands

executing string • 16-19

indirect • 16-17

predefined in XE and XF • 16-13

summary • 16-10 to 16-12

Delta/XDelta Utility (DELTA/XDELTA) (cont'd.)

- depositing command string in system patch space for use by • 16-19
 - displaying contents of address range using • 16-16
 - displaying contents of location using • 16-16
 - expressions • 16-12
 - formats
 - address display • 16-15
 - instruction display • 16-16
 - guidelines • 16-20
 - prefixes
 - G • 16-13
 - H • 16-13
 - setting PC with • 16-18
 - stepping through code with • 16-19
 - symbols
 - period (.) • 16-13
 - Q • 16-13, 16-16, 16-17
 - using in multiprocessing environment • 16-7, G-20
 - values • 16-12
- DEV\$_AVL • 17-20
- DEV\$_ELG • 11-9, C-8
- DEV\$_NET • 17-12
- DEV\$_RED • 17-20
- \$DEVDEF macro • A-53, A-54
- Device
- See also Device unit
 - allocation class • A-27
 - associated mailbox • A-56
 - bus • A-54
 - byte-addressable • 12-22
 - card reader • A-54
 - cluster accessible • A-52
 - cluster available • A-54
 - DIGITAL-supplied • 15-12 to 15-13
 - directory structured • A-53
 - disk • A-54, C-50, C-92
 - dual ported • A-53, A-54
 - file structured • 2-3, 4-8, A-27, A-53
 - input • A-53
 - line printer • A-54
 - mailbox • A-53, A-54
 - mounted • A-53, A-56
 - mounted foreign • A-53
 - network • A-53
 - offsettable • 14-9
 - on VAXBI bus • 14-2
 - output • A-53
 - random access • A-53
 - real time • A-53, A-54

Device (cont'd.)

- record oriented • A-53
 - reference count • A-57
 - sequential block-oriented • A-53
 - shareable • A-53
 - spooled • A-53
 - synchronous communications • A-54
 - tape • A-54, C-92
 - terminal • A-53, A-54
 - timed out • A-56
 - word-aligned • 12-18
 - workstation • A-54
- Device activation bit mask • 8-4
- Device affinity • A-54, C-69
- Device allocation lock • A-52
- Device characteristics • 7-8, A-53 to A-54
- retrieving • C-48
 - setting • C-49 to C-50
 - specifying • 6-2, B-24
- Device class • A-54
- specifying • 6-2, B-24
- Device controller • 1-5, 1-6, A-17
- See also MBA, Controller initialization routine
 - initializing • 11-1
 - intelligent • 1-18
 - multiunit • 3-23, 4-4, 4-14, 8-3, 8-6, 9-8, A-35, A-52, A-55
 - number of units created for • 15-6, B-20
 - number of units supported by • A-33, A-35, A-36, B-20
 - reinitializing • B-20
 - single-unit • 3-24, 4-5, 10-2, 11-2, 15-2, A-35
 - status • A-20
 - synchronizing access to • 3-15
- Device controller channel wait queue • 3-24
- Device controller data channel • 4-4 to 4-5, 13-12, 13-14
- See also Secondary controller data channel
 - obtaining ownership of • 3-23, 4-4, 8-3 to 8-4, A-35, B-57, C-97 to C-98
 - owner • 4-5
 - releasing • 3-24, 8-6, 10-2, B-49, C-83
 - releasing before waiting for interrupt • C-102
 - relinquishing ownership • B-72
 - requesting • 8-3
 - retaining ownership • B-72
 - retaining while waiting for interrupt • C-102
 - unavailability • 8-3
- Device controller data channel wait queue • 8-3, A-19, C-83, C-88, C-98
- Device database • 3-5, 3-15, G-9

Index

- Device database (cont'd.)
 - synchronizing access to • 3-19 to 3-20, B-15 to B-16
- Device data block
 - See DDB
- Device driver • 1-1
 - assembling with SYS\$LIBRARY:LIB.MLB • 15-1, G-8
 - asynchronous nature • 1-1, 1-8 to 1-9, 5-1
 - branching on adapter characteristics • B-2 to B-4
 - branching on processor type • B-8 to B-9
 - calculating base address • 16-7
 - coding conventions • 5-1 to 5-3, 15-1, 16-21, 16-21
 - components • 1-2 to 1-4, 5-1
 - context • 1-7 to 1-9
 - converting uniprocessing to multiprocessing • G-8 to G-20
 - debugging • 16-1 to 16-20
 - displaying address of • 15-10
 - entry points • 1-2, 6-3 to 6-4, A-27, D-1 to E-1
 - example • E-1 to E-29, F-1 to F-25
 - flow • 1-8 to 1-9, 1-19 to 1-21
 - for generic VAXBI device • 14-1 to 14-28, C-103
 - for MASSBUS device • 13-1 to 13-15
 - for Q22 bus device • 12-1 to 12-36
 - for UNIBUS device • 12-1 to 12-36
 - functions • 1-2
 - hardware considerations • 1-9 to 1-16
 - implementing a conditional wait • B-63, B-64
 - linking with SYS\$SYSTEM:SYS.STB • 15-1, 16-7, G-8
 - loading • 6-1, 11-3 to 11-4, 13-6 to 13-7, 15-1 to 15-20, 16-5, A-32
 - machine independence • 1-10, 5-5, 12-16, B-2 to B-4, B-8 to B-9
 - maximum number of supported units • 6-2
 - multiprocessor • 15-10, G-1, G-3
 - name • 4-6, 15-3, 15-6, 15-7, 15-9, A-27, A-33, B-20
 - program sections • 6-3, 15-1, 16-7, B-11, B-19
 - reloading • 15-7 to 15-8
 - size • 5-1, A-32
 - storing data from • 5-1
 - suspending • 2-6, 8-6 to 8-7, 12-24, A-52
 - synchronization flow • 3-16 to 3-19
 - synchronization methods used by • 1-7, 3-1 to 3-24
- Device driver (cont'd.)
 - template for • 5-6 to 5-15
 - uniprocessor • 15-10, G-1, G-3
 - unloading • A-32, B-20
- Device interrupt • 1-6, 3-5, 4-14, 9-1 to 9-8, 12-27 to 12-35
 - See also Interrupt service routine
 - destination for VAXBI node • 14-8
 - direct-vector • 12-2, 12-28, 12-30, 12-32, A-7, A-8, A-23, B-3
 - disabling • 5-4, 10-4
 - enabling • 2-5, 11-2
 - expected • 8-7, 9-3 to 9-4, A-56, C-102
 - multilevel Q22 bus • 12-32, 12-34 to 12-36, A-20
 - non-direct-vector • 12-2, 12-29, 12-30, 12-32, A-7, A-23
 - on MASSBUS • 13-8
 - servicing • 2-6
 - unsolicited • 9-4 to 9-8, A-29
 - waiting for • 2-5 to 2-6, 4-14, 8-6 to 8-7, 12-24, B-73, C-101 to C-102
- Device interrupt vector • 12-27, 14-8, 14-9
 - connecting to • 18-7 to 18-24
 - for generic VAXBI device • 14-13
 - multiple • 12-32, 14-7
 - specifying address • 15-6
 - specifying multiple • 15-6
- Device IPL • 3-5, 9-1, A-55, B-15 to B-16
 - specifying • 6-2, B-24
- Device lock • 3-6, 3-12, 3-15, 8-5, A-47, A-56, C-102
 - See also Spin lock
 - acquisition IPL • C-108
 - address • 3-20, A-20, A-35, A-52
 - multiple acquisition of • B-17, C-112
 - obtaining • 3-9, B-15 to B-16, C-106, C-108
 - ownership • 3-15
 - rank • 3-15
 - releasing • 3-9, B-17 to B-18, C-110
 - restoring • B-17, C-112
- DEVICELOCK macro • 3-8, 3-9, B-15 to B-16, B-61, B-72, C-106, C-108, G-4, G-10, G-11
 - example • B-16, B-18, B-61
 - used by interrupt service routine • 9-3
- Device mode • 7-8
- Device name • 1-5, A-27
- Device registers • 1-6, 1-16 to 1-18, 12-23
 - accessing • 2-5, 4-5, 12-4, 12-23, 14-5, 16-20 to 16-21, 18-1, A-23, A-35, B-15 to B-16

- Device registers (cont'd.)
 - clearing error status • 11-1
 - modification by power failure • 8-5
 - modifying • 5-3
 - of LP11 printer • 2-5
 - rules for referencing • 5-3 to 5-4, 12-4
 - saving the value of • 11-10, D-14
 - synchronizing access to • 3-5, 3-15, 8-5
- Device timeout
 - See Timeout
- Device timeout bit
 - See UCB\$V_TIMEOUT
- Device type • A-54
 - specifying • 6-2, B-24
- Device unit • 1-5, A-47
 - See also UCB, Unit initialization routine
 - activating • 2-5, 8-4 to 8-5, 12-23
 - allocating • A-52, A-53, A-56
 - autoconfiguring • 15-19 to 15-20, B-20
 - busy indicator • A-56
 - CSR address • 15-9
 - deaccessing • A-11
 - deallocating • A-56
 - description • 4-4
 - error retry count • A-58
 - initializing • 11-1
 - marking available • A-53
 - marking on line • 11-2, A-56
 - name • 4-6
 - number • A-55
 - operations count • C-92
 - reference count • 11-6, D-3
 - reinitializing • B-20
 - status • 4-4, A-56 to A-57
 - vector address • 15-9
- DEVICEUNLOCK macro • 3-9, B-17 to B-18, B-61, C-110, C-112, G-4, G-11, G-12
 - example • B-16, B-18, B-61
 - issued by IOC\$WFIKPC and IOC\$WFIRLCH • C-102
- Diagnostic buffer • 4-18, A-39, A-41, A-57, A-62, C-69
 - copied to process space • C-71
 - filling • C-67
 - size • A-29
 - specifying • 4-8, 6-4
- Diagnostic register
 - See MBA\$L_DR
- DIOLM (direct I/O limit) quota
 - adjusting • 4-17
 - charging • 4-7, 4-10
- DIOLM (direct I/O limit) quota (cont'd.)
 - checking • 4-7
- Direct data path • 12-8, 12-10
 - See also Data path
 - functions • 12-10
 - odd transfer • A-8
 - purging • 12-19, 12-24 to 12-25
 - requesting • 12-18
 - speed • 12-10
- Direct I/O • 1-18, 7-4, 14-16, A-39, A-58
 - additional buffer regions for • A-41 to A-43
 - checking accessibility of process buffer for • C-43 to C-44, C-55 to C-56
 - FDT routines for • 7-5, 7-8
 - locking a process buffer for • C-31 to C-33, C-34 to C-36, C-40 to C-42, C-45 to C-47, C-53 to C-54, C-57 to C-58
 - postprocessing • C-70
 - reasons for using • 1-18, 6-7
 - unlocking process buffer • C-105
- Direct memory access transfer
 - See DMA transfer
- Directory sequence number • A-61, A-62
- Direct-vector interrupt • 12-2, 12-28, 12-30, 12-32, 16-9, A-7, A-8, A-23, B-3
- Disk driver • 7-8, 8-3, 8-6, 9-5, A-57, A-58
 - See also MBA, MASSBUS
 - ECC correction routine for • C-65
 - pack acknowledgment in • 11-2
 - recording disk geometry in • 11-2
 - removing a disk volume in • 9-8
 - using local disk UCB extension • A-48, A-61 to A-62
 - waiting for disk unit spinup in • 11-2
- DLDRIVER.MAR • E-1 to E-29
- DMA transfer • 1-17 to 1-18, 5-5
 - See also Map registers, Data path
 - byte-aligned • 12-11
 - calculating starting address • 12-26 to 12-27
 - detecting memory error during • 12-25
 - flow • 1-19 to 1-21, 12-8
 - for modify operation • C-31 to C-33, C-34 to C-36
 - for read operation • C-40 to C-42, C-45 to C-47
 - for write operation • C-53 to C-54, C-57 to C-58
 - longword-aligned 32-bit random-access • 12-12, 12-14

Index

DMA transfer (cont'd.)
on MicroVAX I • 12-24 to 12-25,
12-26 to 12-27
on Q22 bus • 12-15 to 12-16,
12-19 to 12-26
on UNIBUS • 12-15 to 12-26
on VAXBI bus • 14-15 to 14-19
postprocessing • 12-16, 12-24 to 12-26
start I/O routine • 8-1 to 8-7
using direct data path in • 12-10
using direct I/O in • 6-7
using I/O adapter resources in • 12-2 to 12-14

DMB32 asynchronous/synchronous multiplexer •
14-17

DPT\$V_NOUNLOAD • 15-7
DPT\$V_SMPMOD • 15-10, G-3
DPT\$V_SUBCNTRL • 13-14
DPT\$V_SVP • A-58, B-19, C-77, C-78
DPT\$W_DEFUNITS • 15-18
DPT\$W_DELIVER • 15-18, D-19
DPT\$W_UNLOAD • D-9

DPT (driver prologue table) • 1-2, 3-5, 11-1,
16-7, A-30 to A-34, A-53, A-54
creating • 6-1 to 6-3, B-19 to B-24
initialization table • 6-2, 15-4, A-32,
B-23 to B-24
linked into system DPT list • 15-3, 15-7, 15-8
reinitialization table • 6-3, 15-4, 15-8, B-24,
B-24

DPTAB macro • 6-1, 11-1, 14-9, 15-1, A-48,
B-19 to B-21
controlling autoconfiguration with •
15-17 to 15-18
example • B-21
used by MASSBUS drivers • 13-14

DPT_STORE macro • 3-5, 6-2 to 6-3, 11-9,
B-22 to B-24
example • B-21

DR11-W driver • F-1 to F-25

Driver
See Device driver

Driver dispatch table
See DDT

Driver prologue table
See DPT

Driver unloading routine • 6-3, 11-4, 14-18,
15-7 to 15-8, B-20, B-24
address • 6-2, A-33, D-9
context • D-9
exit method • D-9
functions • D-9
input • D-9

Driver unloading routine (cont'd.)
register usage • D-9
synchronization requirements • D-9

DRV11-WA driver • F-1 to F-25

DSBINT macro • 3-8, 3-9, 8-5, 8-6, B-25,
G-4, G-10
replacing with spin lock synchronization macro •
G-13

Dual path UCB extension • A-48

Dual ported device • A-53

DWBUA (BI-to-UNIBUS adapter) • 1-12, 14-9,
18-3
See also UNIBUS adapter

DWMBA
See Memory interconnect to VAXBI adapter

DYN\$C_BUFIO • C-12, C-22
DYN\$C_IRP • C-12

Dynamic spin lock • 3-12

DZ11 controller • A-19
DZ32 controller • A-19

E

ECC error correction • A-57, A-58, A-62, B-19,
C-65

ECC position register • A-62

ECRB (Ethernet controller data block) • B-2

EMB\$C_DA • 11-9
EMB\$C_DE • 11-9
EMB\$C_DT • 11-9
EMB\$L_DV_REGSAV • 11-9
EMB\$W_DV_STS • C-91

\$EMBDEF macro • 11-8

EMB spin lock • 3-13, C-8

Emulated instructions
in device driver • 5-3

ENBINT macro • 3-8, 3-9, B-26, G-4
replacing with spin lock synchronization macro •
G-13

Encryption key • A-41

Entry point
specifying in driver tables • B-11

\$EQLST macro • B-27 to B-28
example • B-27, B-71

ERL\$DEVICEATTN • 11-9, C-8 to C-9, D-14

ERL\$DEVICERR • 11-9, A-29, A-58, A-60,
C-8 to C-9, D-14

ERL\$DEVICTMO • 10-6, 11-9, A-29, A-58,
A-60, C-8 to C-9, D-14

ERL\$RELEASEMB • 10-3, C-92

- Error
 - See also Error logging
 - associated with I/O request • 11–9
 - not associated with I/O request • 11–9
 - servicing within driver • 1–3, 8–5, C–79 to C–80
 - Error log allocation buffer • 11–9, C–8
 - Error logging • A–58, C–8 to C–9
 - driver prerequisites • 11–8
 - enabling • A–53
 - error log sequence number • A–41
 - final error count • 10–3
 - inhibiting • C–8
 - in progress • A–56
 - performed by IOC\$REQCOM • C–92
 - Error logging enable bit
 - See UCB\$V_ERLOGIP
 - Error logging routine • 1–4, 11–8 to 11–10, A–29
 - See also Register dumping routine address • 11–1
 - Error log in progress bit
 - See UCB\$V_ERLOGIP
 - Error log UCB extension • A–48, A–58 to A–60
 - Error message buffer • 3–13, 10–3, A–60, A–62, C–79
 - allocating • 11–9, C–8
 - filling • C–9
 - initializing • 11–9
 - releasing • 10–3, C–92
 - size • C–8
 - specifying size • 6–4, 11–9, A–29
 - written into by IOC\$REQCOM • C–92
 - Error status
 - clearing • 11–1
 - Event flag • A–38
 - handling for aborted I/O request • C–11
 - posting • 4–17
 - setting • 2–7
 - Exception
 - generating • 5–4
- EXE\$ABORTIO • 7–4, 17–12, A–39, C–7, C–10 to C–11, C–33, C–42, C–44, C–46, C–49, C–50, C–54, C–56, C–58, D–11
- EXE\$ALLOCBUF • 7–6, 14–16, C–12 to C–13
- EXE\$ALLOCIRP • A–41, A–43, C–12 to C–13
- EXE\$ALONONPAGED • C–13, C–14, C–59
- EXE\$ALONPAGVAR • C–15
- EXE\$ALOPHYCNTG • 12–26, 14–18, C–16
- EXE\$ALTQUEPKT • 7–5, A–29, C–5, C–17, D–2, D–11
- EXE\$ASSIGN • 11–11, A–11, D–5
- EXE\$BUFFRQUOTA
 - replaced in VMS Version 5.0 • G–5
- EXE\$BUFQUOPRC
 - replaced in VMS Version 5.0 • G–5
- EXE\$CANCEL • 11–6 to 11–7, C–66
- EXE\$CREDIT_BYTCNT • 7–7, C–18, G–5
- EXE\$CREDIT_BYTCNT_BYTLM • C–18, G–5
- EXE\$DASSGN • A–11
- EXE\$DEANONPAGED • C–3, C–13, C–19
- EXE\$DEBIT_BYTCNT • C–20 to C–21, G–5
- EXE\$DEBIT_BYTCNT_ALO • 7–6, 14–16, C–22 to C–23, G–6
- EXE\$DEBIT_BYTCNT_BYTLM • 7–6, C–20 to C–21, G–5
- EXE\$DEBIT_BYTCNT_BYTLM_ALO • 7–6, 14–16, C–22 to C–23, G–6
- EXE\$DEBIT_BYTCNT_BYTLM_NW • C–20 to C–21, G–6
- EXE\$DEBIT_BYTCNT_NW • C–20 to C–21, G–5
- EXE\$FINISHIO • 7–4, 7–8, 17–12, A–40, C–24 to C–25, C–48, C–49, C–50, D–11
- EXE\$FINISHIOC • 7–4, A–40, C–24 to C–25, D–11
- EXE\$FORK • 11–5, A–19, B–30, C–26
- EXE\$FORKDSPATH • 3–5, 3–21, A–52
- EXE\$GB_CPUATYPE • B–9
- EXE\$GL_ABSTIM • A–20
- EXE\$GL_CONFREGL • 14–6
- EXE\$GL_INTSTK
 - replaced by CPU\$_INTSTK • A–12
- EXE\$GQ_1ST_TIME • 3–7, 3–8, 3–12, 3–13, C–29
- EXE\$GQ_SYSTIME • 3–7, 3–8, 3–13, B–47, C–67
 - reading • G–15
- EXE\$HWCLKINT • 3–7
- EXE\$INSERTIRP • 4–12, A–38, A–39, A–55, C–27, C–28, C–38
- EXE\$INSIOQ • 3–20, 4–12, 8–1, A–56, C–28, C–38
 - returning control to • 4–14
- EXE\$INSIOQC • C–28
- EXE\$INSTIMQ • C–29
- EXE\$IOFORK • 9–4, 10–1 to 10–2, 12–24, A–51, A–52, C–30
- EXE\$MODIFY • C–31 to C–33
- EXE\$MODIFYLOCK • C–32, C–34 to C–36
- EXE\$MODIFYLOCKR • A–42, C–32, C–34 to C–36, C–105
- EXE\$ONEPARM • 7–8, A–40, C–37
- EXE\$QIO • 4–1 to 4–12, A–11, A–29, A–36 to A–39, A–41

Index

EXE\$QIOACPPKT • A-52
EXE\$QIODRVPKT • 4-12, 7-4, 7-8, 7-9, 8-1,
C-32, C-37, C-38, C-41, C-50, C-54,
C-60, D-11
EXE\$QIORETURN • 17-12, C-39
EXE\$READ • 7-8, A-41, C-40 to C-42
EXE\$READCHK • 7-6, C-43 to C-44
EXE\$READCHKR • C-32, C-35, C-41,
C-43 to C-44, C-45
EXE\$READLOCK • C-41, C-45 to C-47
EXE\$READLOCKR • A-42, C-41, C-45 to C-47,
C-105
EXE\$SENSEMODE • 7-8, C-48
EXE\$SETCHAR • 7-8, C-49 to C-50
EXE\$SETMODE • 7-8, C-49 to C-50
EXE\$SNDEVMSG • 9-7 to 9-8, 10-6,
C-51 to C-52, G-7
EXE\$SWTIMINT • 3-7
EXE\$TIMEOUT • A-52, A-56, A-57
EXE\$WRITE • 7-8, A-41, C-53 to C-54
EXE\$WRITECHK • 7-6, C-55 to C-56
EXE\$WRITECHKR • C-54, C-55 to C-56, C-57
EXE\$WRITELOCK • C-54, C-57 to C-58
EXE\$WRITELOCKR • A-42, C-54, C-57 to C-58,
C-105
EXE\$WRTMAILBOX • C-51, C-59
EXE\$ZEROPARM • 7-9, A-40, C-60
Expected interrupt
See Device interrupt
External register base
See MBA\$L_ERB

F

FDT (function decision table) • 1-2, 4-9
address • 4-7, 6-3, A-29
as used by EXE\$QIO • 4-7
creating • 6-4 to 6-7, 11-3, B-34 to B-35
dispatching to FDT routines from • 4-10
relocating addresses specified in • 11-4
size • A-30
specifying buffered functions in • 4-9
specifying legal functions in • 4-9
FDT routine • 1-3, 1-18, 2-3 to 2-4
adjusting process quotas in • C-12
allocating IRPE in • A-41
allocating system buffer in • 7-6 to 7-7
calling sequence • 7-2
completing an I/O operation in • C-24 to C-25
context • 4-12, 7-1, D-10

FDT routine (cont'd.)
creating • 7-1 to 7-5
dispatched to/from EXE\$QIO • 4-10
ensuring an even byte count in • 12-23
entry point • D-10
exit method • 7-2 to 7-4, D-11
for buffered I/O • 7-6 to 7-8
for direct I/O • 7-5, 7-8, C-31 to C-33,
C-40 to C-42, C-53 to C-54
provided by VMS • 7-8 to 7-9
register usage • 5-2, 7-1, D-10
returning to the system service dispatcher •
C-39
setting attention ASTs in • C-6
specifying • D-10
synchronization requirements • D-10
unlocking process buffers in • C-105
File structured device • A-53
File system
synchronizing access to • 3-12
FILSYS spin lock • 3-12
FIND_CPU_DATA macro • B-29, G-7
example • B-29
Floating address • 15-12
Floating CSR space
assigning to device • 15-19
current base • 15-19
Floating-point instructions
in device driver • 5-3
Floating vector space
assigning to device • 15-19
current base • 15-19
Fork block • 1-5, 1-8, 3-21, 3-24,
4-13 to 4-14, 8-7, 10-1, B-72, C-26,
C-30, C-101 to C-102
dequeuing • 3-5
in CRB • 15-7, A-19
in extended UCB • 11-5
in UCB • A-51 to A-52
Fork context • 1-8, 3-20 to 3-21, 4-13
Fork database • 3-5
accessing • B-31 to B-32
synchronizing access to • 3-20 to 3-22
Fork dispatcher • 2-6, 3-3, 3-5, 3-7, 3-21,
B-31
functions • 4-15
Forking • 3-15, 3-21, B-30, B-40, C-26, C-30,
G-9
avoiding multiple • 11-5
from controller initialization routine •
11-5 to 11-6, D-7
from driver unloading routine • D-9

Forking (cont'd.)

- from interrupt service routine • 9–5
- from unit initialization routine • 11–5 to 11–6, D–21
- in terminal port driver • 17–13, 17–18

Fork IPL • 2–4, 3–2, 3–5, 3–14, 3–20, 4–15, A–51, B–31 to B–32

Fork lock • 2–4, 3–5, 3–7, 3–12, 3–14 to 3–15, 3–20, 11–6, 12–15, A–19, A–47

See also Spin lock

- acquisition IPL • C–107
- multiple acquisition of • B–33, C–111
- obtained by fork dispatcher • 3–5
- obtaining • 3–9, B–31 to B–32, C–107
- ownership • 16–26
- rank • 3–12 to 3–13
- releasing • 3–9, B–33, C–109
- restoring • B–33, C–111

Fork lock index • 3–12 to 3–13, A–51

- list • G–9
- placing in UCB\$B_FLCK • 6–2, B–24, G–8

FORKLOCK macro • 3–8, 3–9, B–31 to B–32, C–107, G–4

- example • B–32

FORK macro • 3–11, 3–21, 12–18, 12–20, B–30, C–26

See also IOFORK macro

Fork process • 1–8, 3–20 to 3–22, 8–1

- context • 4–12 to 4–13, 4–13 to 4–14, 4–14, 8–1 to 8–2
- creating • B–30, B–40, C–26, C–30
- creation by driver • 2–6, 4–14, 10–1 to 10–2
- creation by IOC\$INITIATE • 4–12 to 4–13, 8–1, 10–3, C–68 to C–69
- reactivating • 4–15 to 4–16
- rules • 3–22
- suspending • 4–14, 8–6 to 8–7, B–72, C–101 to C–102

Fork queue • 3–22, 4–14, 4–15, A–16, A–51, C–26, C–30, G–15

FORKUNLOCK macro • 3–9, B–33, C–109, C–111, G–4

- example • B–32

Full-checking synchronization image • 16–25, G–17

- loading • G–2

Full duplex device driver • 7–5, D–2

- I/O completion for • C–5

FUNCTAB macro • 6–6, B–34 to B–35

- example • B–35

Function decision table

- See FDT

G

General purpose registers

- rules for using in driver code • 5–2

Generic VAXBI device • 11–2, 14–1 to 14–28

- See also VAXBI node
- initialized by driver • 14–9 to 14–15
- initialized by VMS • 14–5 to 14–9
- interrupt destination • 14–8

H

Hardware clock

- See Interval clock

HWCLK spin lock • 3–7, 3–8, 3–13, C–29, G–14, G–15, G–25

I

I/O adapter • 1–6, 1–10 to 1–16, 1–17

- See also UBA, UNIBUS adapter, MBA, and Q22 bus
- configuration register • A–6
- data path register • B–46
- displaying nexus value • 15–8, 15–9
- number of address bits • A–8, B–3
- on VAXBI bus • 14–2
- type • 14–8, A–6, A–32, B–3, B–19

I/O adapter registers

- See Map registers, Data path register, Vector register, Byte count register, MBA

I/O address space • 18–1 to 18–7

- access to during bus power failure • 18–6
- error in mapping • 18–6
- mapping to process address space • 18–4, 18–5 to 18–7, 18–7
- of VAXBI bus • 14–2
- rules for referencing • 18–6

I/O channel

- See Process I/O channel

I/O completion

- See I/O postprocessing

I/O database • 1–4 to 1–6, A–1

- creation • 6–1, 6–2, 11–3, 13–6, 15–3 to 15–6, 15–11, A–32, B–24

Index

I/O database (cont'd.)

- examining with XDELTA • 16–10
- for MASSBUS configuration • 13–6 to 13–7, 13–12
- for two-controller configuration • 4–6
- initializing • 11–3, 15–11
- locating • 15–10
- referencing fields in • 5–1
- reinitializing • 11–4

I/O function

- analyzing • 8–2
- indicating a buffered • 4–9, 6–4
- indicating as legal to a device • 4–9, 6–4
- preprocessing • 4–10

I/O function code • 4–9, A–38

- converting to device-specific function code • 8–4
- defined by VMS • 6–4 to 6–6
- defining device-specific • 6–7

I/O function modifier • 4–9

I/O postprocessing • 3–4, 10–1 to 10–4, A–40

- device-dependent • 2–7, 4–17, 7–7, 10–2 to 10–4
- device-independent • 2–7, 4–17 to 4–18, 7–7, C–70 to C–71
- for aborted I/O request • C–10
- for buffered I/O • 7–7 to 7–8, 12–25
- for DMA transfer • 12–16, 12–24 to 12–26
- for full duplex device driver • C–5
- for I/O request involving no device activity • C–24 to C–25
- synchronization flow • 3–4

I/O postprocessing queue • 10–3, 11–6, A–16, A–57, C–5, C–92, G–15

I/O preprocessing

- See also SY\$QIO and FDT routine
- completing • 4–12, 6–4
- device-dependent • 2–3 to 2–4, 4–9 to 4–12, 7–1 to 7–9
- device-independent • 2–3, 4–1 to 4–9
- IPL requirements • 3–4

I/O request

- aborting • 7–4, 10–6, C–10 to C–11
- canceling • 11–6 to 11–8, A–29, A–56, C–66
- completing • C–91 to C–92
- example • 2–1 to 2–7
- outstanding on channel • A–11
- restarting after power failure • 8–5
- retrying • 10–5 to 10–6
- returning completion status of to process • 2–7, 4–18, 7–4, 10–2, 10–3
- status • A–39

I/O request (cont'd.)

- synchronizing simultaneous processing of multiple • 7–5
- validating device-dependent arguments • 2–3
- validating device-independent arguments • 2–2 to 2–3, 4–7
- with no parameters • 7–9, C–60
- with one parameter • 7–8, C–37

I/O request packet

See IRP

I/O space

- of MASSBUS • 13–4
- of Q22 bus • 12–4
- of UNIBUS • 12–4
- rules for referencing • 5–3, 5–4
- writing to • 5–4

I/O status block

See IOSB

IDB\$_ADP • 4–5

IDB\$_CSR • 4–5, 13–4, 13–12, 14–8

IDB\$_OWNER • 3–24, 4–4, 4–5, 8–4, 8–7, 9–3, 11–2, C–83, C–97

IDB\$_UCBLST • 14–20

IDB\$_NO_CSR • A–35

IDB\$_W_UNITS • 14–7, 15–6

IDB (interrupt dispatch block) • 1–6, 4–5 to 4–6, 12–23, A–34 to A–36

address • 4–4, 8–4, 12–31, 12–33

creation • 15–4, B–20

for generic VAXBI device • 14–7

for MBA • 13–4, 13–6 to 13–7, 13–12, 13–14

size • B–20

Idle time • G–24

IFNORD macro • B–36 to B–37

IFNOWRT macro • B–36 to B–37

IFRD macro • B–36 to B–37

example • B–37

IFWRT macro • B–36 to B–37

ILLQBUSCFG bugcheck • A–20

Image termination • 11–6, D–3

INCONSTATE bugcheck • C–85, C–94

IN\$BRK • 16–6

Initialization routine

See Unit initialization routine, Controller initialization routine

Initialization table • 6–2, A–33, B–23

INIT module • G–23

INIT processor state • A–15, G–21

Input device • A–53

- Interlocked instructions
 - using in multiprocessing environment • G-14 to G-15
- Interprocessor interrupt • 3-4, 3-13, A-15
- Interrupt • 3-2
 - See also Device interrupt
 - blocking • B-25, B-60
 - dismissing • 10-1
 - interprocessor • 3-4, 3-13, A-15
 - requesting an XDELTA • 16-7 to 16-8
 - requesting a software • 3-9, B-62
- Interrupt context • 1-8, 9-3
- Interrupt dispatch block
 - See IDB
- Interrupt dispatcher • 3-5, 12-24, 14-7, 14-9, A-7, A-8
 - for MASSBUS • 13-7, 13-7 to 13-10, 13-14 to 13-15, D-23
 - for Q22 bus • 12-27 to 12-35
 - for UNIBUS • 12-27 to 12-35, A-23
- Interrupt enable bit • 8-4
- Interrupt expected bit
 - See UCB\$V_INT
- Interrupt priority level
 - See IPL
- Interrupt service routine • 1-3, 3-3, 3-13, 9-1 to 9-8, 12-24, A-52
 - address • 6-3, 12-33, A-23, B-24, D-12, G-5
 - context • 9-3, D-12
 - entry point • 4-14, D-12
 - example • 9-6 to 9-8
 - exit method • D-13
 - for connect to interrupt facility • 18-10, 18-16 to 18-17
 - for LP11 printer • 2-6
 - for MASSBUS device • 13-10, 13-15, D-12
 - for solicited interrupt • 9-3 to 9-4
 - for terminal port driver • 17-16
 - for unsolicited interrupt • 9-4 to 9-8, D-23
 - functions • 4-14, 9-1, D-13
 - input • D-13
 - of CONINTERR.EXE • 18-13
 - of UNIBUS adapter • 12-30
 - preemption of device timeout handling • 10-5
 - register usage • 8-7, D-12
 - specifying more than one • D-12
 - synchronization requirements • 3-5, 3-19, 9-3, D-12, G-11 to G-12
- Interrupt stack • 8-1
 - address • A-15
- Interrupt transfer routine • 12-32
- Interrupt transfer vector
 - See VEC
- Interrupt vector • 15-9
 - See Device interrupt vector number • 15-6
- Interval clock • 3-6, 3-7, 3-13, G-25 to G-26
 - interrupt service routine • 3-7, 3-8
 - role in device timeouts • 1-3
- INVALIDATE spin lock • 3-13
- INVALIDATE_TB macro • B-38 to B-39, G-16
- INVALID macro
 - replaced by INVALIDATE_TB macro • G-16
- IO\$_INHERLOG • C-8
- IO\$_AVAILABLE function • 7-8
- IO\$_CONINTREAD function • 18-8, 18-9
- IO\$_CONINTWRITE function • 18-8, 18-9
- IO\$_PACKACK function • 7-8
- IO\$_SENSECHAR function
 - servicing • C-48
- IO\$_SENSEMODE function
 - servicing • C-48
- IO\$_SETCHAR function • 11-9
 - servicing • C-49 to C-50
- IO\$_SETMODE function • 17-13
 - servicing • C-49 to C-50
- IO\$_TTY_PORT function • 17-12
- IO\$_UNLOAD function • 7-8
- \$IO650DEF macro • 18-1
- \$IO730DEF macro • 18-1
- \$IO750DEF macro • 18-1
- \$IO780DEF macro • 18-1
- \$IO790DEF macro • 18-1
- \$IO8NNDDEF macro • 14-14, 18-1
- \$IO8PSDEF macro • 14-14
- \$IO8SSDEF macro • 14-14, 18-1
- \$IO9CCDEF macro • 14-14, 18-1
- IOC\$ALLOSPT
 - replaced by LDR\$ALLOC_PT • G-7
- IOC\$ALOALTMAP • A-9, C-61 to C-62, C-90
- IOC\$ALOALTMAPN • 12-20, C-61 to C-62
- IOC\$ALOALTMAPSP • C-61 to C-62
- IOC\$ALOUBAMAP • C-63 to C-64, C-87, C-96
- IOC\$ALOUBAMAPN • 12-20, C-63 to C-64
- IOC\$APPLYECC • A-62, C-65
- IOC\$CANCELIO • 11-8, A-56, C-66, D-3
- IOC\$DIAGBUFILL • A-29, A-41, C-67
- IOC\$GL_CRBTMOUT • A-20
- IOC\$GL_DEVLIST • 11-4, A-25
- IOC\$GL_DPTLIST • 15-3, 15-8

Index

- IOC\$GL_IRPFL
 - replaced in VMS Version 5.0 • G-15
- IOC\$GL_LRPFL
 - replaced in VMS Version 5.0 • G-15
- IOC\$GL_MUTEX • 11-11, D-5
- IOC\$GL_PSFL
 - replaced by CPU\$_PSFL • G-15
- IOC\$GL_SRPFL
 - replaced in VMS Version 5.0 • G-15
- IOC\$GQ_IRPIQ • G-15
- IOC\$GQ_LRPIQ • G-15
- IOC\$GQ_SRPIQ • G-15
- IOC\$GW_MAXBUF • C-20, C-22
- IOC\$INITIATE • 3-20, 4-12 to 4-13, 8-1, 10-3, A-29, A-40, A-55, A-56, A-58, C-28, C-38, C-67, C-68 to C-69, C-92, D-15
- IOC\$IOPPOST • 3-4, A-41, A-42, C-70 to C-71
 - unlocking process buffers • C-105
- IOC\$LOADALTMAP • 12-22, B-41, C-72 to C-73
- IOC\$LOADMBAMAP • 13-3, B-42, C-74
- IOC\$LOADDUBAMAP • 12-21 to 12-22, A-24, B-43, C-75 to C-76
- IOC\$LOADDUBAMAPA • 12-22, C-75 to C-76
- IOC\$MNTVER • A-29
- IOC\$MOVFRUSER • 12-26, 14-18, B-19, C-77
- IOC\$MOVFRUSER2 • C-77
- IOC\$MOVTOUSER • 12-27, 14-19, B-19, C-78
- IOC\$MOVTOUSER2 • C-78
- IOC\$PURGDATAP • 12-24 to 12-25, 12-27, A-24, B-46, C-79 to C-80
- IOC\$RELALTMAP • 12-26, A-9, A-52, B-48, C-81 to C-82
- IOC\$RELCHAN • 10-2, A-19, A-35, A-52, B-49, C-83, C-92
 - called by IOC\$WFIRLCH • C-102
- IOC\$RELDATAP • 12-25, A-7, A-9, A-52, B-50, C-84
- IOC\$RELMAPREG • 12-25 to 12-26, A-8, A-9, A-24, A-25, A-52, B-51, C-86 to C-87
- IOC\$RELSCHAN • A-19, A-20, A-35, B-52, C-88
- IOC\$REQALTMAP • 12-19, A-9, A-52, B-53, C-89 to C-90
- IOC\$REQCOM • 3-20, 8-1, 10-3 to 10-4, A-29, A-38, A-40, A-55, A-56, A-57, A-58, A-60, B-54, C-13, C-91 to C-92, D-15
 - error logging activities • 11-9
- IOC\$REQDATAP • 12-17, A-7, A-9, A-24, A-52, B-55, C-93 to C-94
- IOC\$REQDATAPNW • 12-18, C-93 to C-94
- IOC\$REQMAPREG • 12-19, A-8, A-9, A-24, A-25, A-52, B-56, C-95 to C-96
- IOC\$REQPCHANH • A-19, A-35, A-52, B-57, C-97 to C-98
- IOC\$REQPCHANL • 8-3 to 8-4, A-19, A-35, A-52, B-57, C-97 to C-98
- IOC\$REQSCHANH • A-19, A-20, A-35, B-58, C-97 to C-98
- IOC\$REQSCHANL • A-19, A-20, A-35, A-52, B-58, C-97 to C-98
- IOC\$RETURN • 11-7, B-11, C-99
- IOC\$SEARCHDEV • A-52
- IOC\$VERIFYCHAN • C-100
- IOC\$WFIKPCH • 4-13, 4-14, 8-7, A-52, A-56, A-57, C-101 to C-102
- IOC\$WFIRLCH • 4-13, 4-14, A-56, A-57, C-101 to C-102
- \$IODEF macro • 6-4
- IOFORK macro • 3-11, 3-21, 4-14, 9-4, 10-1, 12-24, B-40, C-30
- IOLOCK10 fork lock • 3-12
- IOLOCK11 fork lock • 3-13
- IOLOCK8 fork lock • 3-7, 3-12
- IOLOCK9 fork lock • 3-12
- IOSB (I/O status block) • 7-4, 10-2, 10-3, A-39, A-40, C-5, C-10, C-71, C-92
 - validating access to • 4-7
- \$IOUV1DEF macro • 18-1
- \$IOUV2DEF macro • 18-1
- IPL\$_ASTDEL • 3-2, 3-4, 3-16, 4-7, C-10, C-12, C-31, C-34, C-37, C-38, C-40, C-43, C-48, C-49, C-55, C-60, C-71, C-100, C-109, C-111, C-112, D-5, D-10
- IPL\$_EMB • C-8
- IPL\$_FILSYS • 3-12
- IPL\$_IOLOCK8 • 3-12
- IPL\$_IOPPOST • 2-7, 3-2, 3-4, 4-17, 10-3, 11-6, C-5, C-10, C-24, C-71, C-92
- IPL\$_JIB • 3-12
- IPL\$_MAILBOX • 3-2, 3-8, 3-13, 9-7, 10-6, C-51, C-59
- IPL\$_MMG • 3-12
- IPL\$_POOL • 3-2, C-14, C-15
- IPL\$_POWER • 3-6, 8-5 to 8-6, 11-4, 15-4, D-7, D-9
- IPL\$_QUEUEAST • 3-2, 3-7, 3-12, 18-15, 18-17, C-2, C-3
- IPL\$_RESCHED • 3-2, 3-5, 3-7, B-29, C-107, C-108
- IPL\$_SCHED • 3-12
- IPL\$_SYNCH • 3-2, 3-7, 3-8
- IPL\$_TIMER • 3-12, C-29
- IPL\$_TIMERFORK • 3-2, 3-7, 10-4

IPL (interrupt priority level) • 1–7, 3–1 to 3–11
 See also Device IPL, Fork IPL
 hardware • 3–1
 lowering • 3–8 to 3–11, 3–21, 8–7, C–26, C–30
 modifying • B–15 to B–16, B–17 to B–18, B–25, B–26, B–31 to B–32, B–33, B–44 to B–45, B–60, B–66
 raising • 3–8 to 3–11, 3–14, B–60
 relation to spin lock • 3–13
 saving • 3–9, B–15, B–31, B–44, B–59
 software • 3–1

IRP\$_CARCON • A–41, C–32, C–41, C–54
 IRP\$_PRI • C–27
 IRP\$_BCNT • 8–2, C–32, C–35, C–41, C–43, C–45, C–54, C–55, C–57, C–68, C–69, C–70
 writing • 7–6

IRP\$_DIAGBUF • C–67, C–68, C–69
 IRP\$_IOST2 • C–32, C–41, C–54
 IRP\$_KEYDESC • C–70
 IRP\$_MEDIA • 7–4, 10–3, 11–6, A–40, C–37, C–50, C–60
 IRP\$_PID • 11–8, C–66, D–4
 IRP\$_SVAPTE • 8–2, C–32, C–35, C–41, C–46, C–54, C–58, C–68, C–69
 for buffered I/O • 7–6, 7–7

IRP\$_BUFIO • C–70
 IRP\$_DIAGBUF • C–67, C–68, C–69, C–70
 IRP\$_EXTEND • C–70
 IRP\$_FUNC • 7–6, 7–7, 11–6, C–32, C–35, C–41, C–43, C–46
 IRP\$_KEY • C–70
 IRP\$_MBXIO • C–70
 IRP\$_PHYSIO • C–70
 IRP\$_W_BOFF • 7–6, 7–7, 8–2, C–32, C–35, C–41, C–46, C–54, C–58, C–68, C–69, C–70
 IRP\$_W_CHAN • 11–8, C–66, D–4
 IRP\$_W_FUNC • 8–4
 IRP\$_W_STS
 for read function • 7–6, 7–7
 for write function • 7–7

IRP (I/O request packet) • 1–6, A–36 to A–41
 allocating • 4–7
 copying to UCB • 8–2
 creation • 2–3, 4–7
 current • A–55
 deallocation • 2–7, C–71
 dequeuing from UCB • A–38
 device-independent portion of • 4–8

IRP (I/O request packet) (cont'd.)
 insertion in pending-I/O queue • 2–4, 4–12, 7–4, 8–1, C–27, C–28
 insertion in postprocessing queue • 2–7
 removal from pending-I/O queue • 2–7, 4–12, 10–3
 size • A–36
 storing data in • 5–1, G–16
 unlocking buffers specified in • C–105

IRPE (I/O request packet extension) • A–39, A–41 to A–43, C–70
 address • A–41
 allocating • A–41
 deallocation • A–42, C–71, C–105
 unlocking buffers specified in • C–71, C–105

J

JIB\$_BYTCNT • 3–12, 7–6, 7–7, C–12, C–18, C–20, C–22, G–5
 JIB\$_BYTLM • 3–12, C–12, C–18, C–20, C–22, G–5
 JIB\$_BYTCNT_WAITERS • C–18
 JIB (job information block) • 3–12
 JIB spin lock • 3–12, C–18, C–20, C–22
 Job attached bit
 See UCB\$_JOB
 Job controller • A–57
 sending a message to • 9–7 to 9–8, C–52, C–59
 Job information block
 See JIB
 Job quota • G–5
 byte count • 2–3, 3–12, C–12, C–18, C–20 to C–21, C–22 to C–23
 byte limit • 3–12, C–12, C–18, C–20 to C–21, C–22 to C–23

K

Kernel stack • 8–1

L

LDR\$_ALLOC_PT • 14–15, C–103, G–7
 LDR\$_DEALLOC_PT • C–104
 LDR\$_GL_FREE_PT • C–103, C–104

Index

LDR\$GL_SPTBASE • C-103, C-104
Legal function bit mask • 4-9
LOADALT macro • 12-10, 12-22, B-41, C-72
LOADER\$_PTE_NOT_EMPTY status • C-104
LOADMBA macro • 13-3, 13-12, 13-13, B-42, C-74
LOADUBA macro • 12-10, 12-11, 12-21, B-43, C-75
Local disk UCB extension • A-48, A-61 to A-62
 required for error logging • 11-8, C-9
 required for IOC\$APPLYECC routine • C-65
Local processor • 1-7
Local tape UCB extension • A-48, A-60 to A-61
 required for error logging • 11-8, C-9
Lock ID • A-52
LOCK macro • 3-8, 3-9, B-44 to B-45, C-107, G-4
Lock manager • A-52
Logical I/O function
 translation from virtual function to • 2-3
 translation to physical function • C-31, C-40, C-53
Longword access enable bit
 See VEC\$V_LLWAE
Longword-aligned random-access mode • 12-2, 12-11, 12-14, A-24
Lookaside list
 See Nonpaged pool
Loopback mode • A-69
LWAE (longword access enable) bit
 See VEC\$V_LLWAE

M

Machine check • 3-13, 16-21, 18-6
 condition handler • 18-6
Machine check protection block • 14-11
Macro
 format • B-1
Mailbox • A-53, A-54, A-55
 associated with device • A-56
 buffered I/O quota for • A-52
 I/O function • A-39
 in shared memory • A-57
 marked for deletion • A-57
 of job controller • 9-7, G-7
 of OPCOM process • 10-6, G-7
 permanent • A-57
 sending a message to • C-51 to C-52, C-59
Mailbox (cont'd.)
 synchronizing access to • 3-8, 3-13
Mailbox driver • 15-5
MAILBOX spin lock • 3-13, C-51, C-59
Maintenance function • 17-13
Map register base register
 See MBA\$_MAP
Map registers • 1-17, 12-2, 12-4 to 12-7, 12-15, 12-19 to 12-22, A-8, A-23 to A-24, A-24, B-3
 See also Alternate map registers
 allocating • C-63 to C-64
 allocating permanent • 11-2, 12-20 to 12-21, A-24, G-12
 byte offset bit • C-75
 calculating the number needed • 12-19
 format • 12-5 to 12-7, 12-21
 invalidating • 12-7, 12-13, 12-22
 loading • 12-21 to 12-22, B-43, C-75 to C-76
 number of active • A-9
 number of disabled • A-9
 of MBA • 13-2, B-42, C-74
 of Q22 bus • 12-5
 of UBA • 12-5
 operation • 12-5 to 12-7
 releasing • 10-2, 12-25 to 12-26, B-51, C-86 to C-87
 requesting • 12-19 to 12-21, B-56, C-95 to C-96
Map register valid bit • 12-21
Map register wait queue • 12-19, 12-25, A-8, C-87, C-96, G-14
MASSBUS
 configuration • 13-1, 13-4
 I/O address space • 18-1
 I/O database • 13-4, 13-6 to 13-7
 servicing multiunit controller on • 13-2, 13-6, 13-11, 13-12, 13-14
 servicing single-unit controller on • 13-6, 13-10, 13-11, 13-12, 13-14
MASSBUS adapter
 See MBA
MASSBUS driver
 DPT for • 13-14
 interrupt service routine • 13-15
 start I/O routine • 13-12
 unit initialization routine • 13-11
 unsolicited interrupt service routine • 13-14
MBA\$INT • 13-14 to 13-15, D-23
MBA\$_AS • 13-4, 13-5, 13-8 to 13-9, 13-9, 13-10

- MBA\$_BCR • 13-3, 13-4, 13-13, C-74
 - MBA\$_CAR • 13-4
 - MBA\$_CR • 13-4
 - MBA\$_CSR • 13-4, 13-13
 - MBA\$_DR • 13-4
 - MBA\$_ERB • 13-4, 13-5, 13-11
 - MBA\$_MAP • 13-4, C-74
 - MBA\$_SMR • 13-4
 - MBA\$_SR • 13-4, 13-10, 13-12
 - MBA\$_VAR • 13-3, 13-4, 13-13, C-74
 - MBA (MASSBUS adapter) • 1-10, 1-11
 - address space • 13-4 to 13-5
 - data path • 13-3
 - functions • 13-1, 13-8 to 13-9
 - nexus value of • 15-5
 - obtaining ownership • 13-2, 13-6 to 13-10, 13-12 to 13-13
 - registers • 13-1 to 13-6
 - device • 13-5, 13-11, 13-12
 - external • 13-2
 - internal • 13-2
 - map • 13-2 to 13-6, B-42, C-74
 - releasing secondary data channel • C-88
 - subunit number • 13-1
 - unit number • 13-1, 13-11, 15-6
 - \$MBADEF macro • 13-4 to 13-5
 - MCHECK spin lock • 3-13
 - \$MCHKDEF macro • 14-11
 - Media ID • A-58
 - MEGA spin lock • 3-13
 - Memory
 - See also Buffer, Nonpaged pool
 - detecting corruption in • 16-22 to 16-24
 - detecting parity errors in • 12-25, B-46
 - testing accessibility of • B-36 to B-37
 - Memory interconnect to VAXBI adapter • 14-2, 14-6, 14-8
 - ADP address • 14-8
 - Memory management resources
 - synchronizing access to • 3-12
 - MicroVAX 3600 series • 1-15
 - booting with XDELTA from • 16-2
 - requesting an XDELTA interrupt from • 16-8
 - MicroVAX I • 1-16
 - accomplishing a DMA transfer on • 12-24 to 12-25
 - adapter logic • 12-1
 - booting with XDELTA from • 16-2
 - comparison with other VAX systems • 1-18
 - DMA transfer • 12-26 to 12-27
 - requesting an XDELTA interrupt from • 16-8
 - MicroVAX II • 1-15
 - adapter logic • 12-1
 - booting with XDELTA from • 16-2
 - requesting an XDELTA interrupt from • 16-8
 - MMG\$_GL_SBICONF • 14-6
 - MMG\$_IOLOCK • C-32, C-35, C-41, C-46, C-54, C-58
 - MMG\$_UNLOCK • A-42, C-105
 - MMG spin lock • 3-12, C-16, C-103, C-104, C-105
 - Modem signals
 - input transitions of • 17-14
 - sending to device • 17-12
 - Mount verification • A-39, A-57
 - Mount verification routine • A-29, A-30
 - MSG\$_CRUNSOLIC • 9-7
 - MSG\$_DEVOFFLIN • 10-6
 - Multilevel device interrupt dispatching • 12-32, 12-34 to 12-36, A-20
 - Multiprocessing device driver
 - analyzing crash dumps • G-18 to G-19
 - incompatibility with uniprocessing driver • 15-10, G-3
 - using XDELTA • 16-7, G-20
 - writing • G-8 to G-20
 - Multiprocessing environment
 - contrasted with uniprocessing environment • 3-10, G-1
 - debugging a driver designed for • 16-25 to 16-27
 - MULTIPROCESSING parameter • 16-24, 16-25, G-2 to G-3, G-4, G-23
 - Multiprocessor state • A-15, G-20 to G-24
 - Mutex
 - for ACL • A-45
 - for I/O database • D-5
 - I/O database • 11-11
-
- N
-
- NBI
 - See Memory interconnect to VAXBI adapter
 - Network device • A-53
 - Nexus • 15-5, 15-8, 15-9
 - Nexus ID • A-6
 - Node • 15-5, 15-8, 15-9
 - See VAXBI node
 - Node ID • 14-8, A-6
 - Node private space • 14-5
 - Node space • 14-5

Index

Node space (cont'd.)

- accessing BIIC registers within • 14–5
 - address • 14–8
 - mapped by VMS • 14–6
- Non-direct-vector interrupt • 12–2, 12–29, 12–30, 12–32, 16–8, A–7, A–23
- Nonpaged pool
- allocating • C–12 to C–13, C–14, C–15, C–22 to C–23
 - allocating in initialization routine • 11–2
 - deallocating • C–3, C–19
 - lookaside list • C–13, C–14, G–15
 - synchronizing access to • 3–13
 - variable region • C–15, G–14
- NPR (Nonprocessor request)
- See DMA transfer
- Null process • G–24

O

- Object
- protection • A–45
- Online bit
- See UCBSV_ONLINE
- Online condition
- on MASSBUS • 13–9
- OPCOM process
- sending a message to • 10–6, C–52, C–59
- Operator device • A–53
- ORB (object rights block) • A–43 to A–45
- address • A–52
 - cloned • 11–12, D–6
- Output device • A–53

P

- Page fault
- taken within driver code • 3–4
- Page table
- physical address of • 14–18
- Page-table entry
- allocating • C–103
 - deallocating • C–104
 - format • 14–17
 - modifying • B–38, G–16
- Paging I/O function • A–39
- PAT\$A_NONPGD • 16–20
- Patch space • 16–20

PBI

- See Memory interconnect to VAXBI adapter
- PCB\$_ASTQFL • G–14
- PCB\$_JIB • 7–6
- PCB\$_PID • 11–8, C–66, D–4
- PCB\$_SSRWAIT • 4–7, C–12, C–20, C–22
- PCB\$_ASTCNT • C–4, C–6, C–10
- modifying with ADAWI instruction • G–14
- PCB\$_BIOCNT • 2–7
- PCB (process control block) • 3–4, 16–13
- referring to current • G–7
 - synchronizing access to • 3–12
- PDT (port descriptor table) • A–58
- Pending-I/O queue • 3–20, 4–12, 8–1, 11–6, A–38, A–55, C–27, C–28, C–37, C–38, C–71, C–92, G–14
- bypassing • 7–5, C–17
 - length • A–57, C–28
 - synchronizing with driver internal queue • 7–5
- Per-CPU database
- See CPU
- PERFMON spin lock • 3–13
- Performance
- stack time • A–16
- PFN database
- examining with XDELTA • 16–13 to 16–14
- PFN mapping • 18–5 to 18–7
- deleting a page designated for • 18–6
 - modifying a page designated for • 18–5
- PHD\$_BIOCNT • 2–7
- Physical address
- format • 18–4
- Physical I/O function • A–39, C–70
- PID (process identification number) • A–52
- PIO transfer • 1–17
- example • 2–1 to 2–7
 - using buffered I/O in • 6–7
 - using I/O adapter resources in • 12–2
- Pool checking mechanism • 16–22 to 16–24
- POOLCHECK parameter • 16–22
- POOL spin lock • 3–13, C–14, C–15, C–19
- Port driver
- See Terminal port driver
- Port driver entry vector table • A–33
- Port driver vector table • 17–4, A–67
- address • 17–8, B–7
 - creating • 17–5 to 17–6, B–68, B–69
 - defining entry in • B–67
 - relocating • B–6
- PORT_ABORT service routine • 17–15
- PORT_CANCEL service routine • 17–15

- PORT_DISCONNECT initiate routine • 17–12
 - PORT_DS_SET initiate routine • 17–12
 - PORT_FDT initiate routine • 17–12
 - PORT_FORKRET initiate routine • 17–13, 17–18
 - PORT_MAINT initiate routine • 17–13, A–68
 - PORT_RESUME service routine • 17–15
 - PORT_SET_LINE initiate routine • 17–13
 - PORT_SET_MODEM initiate routine • 17–14
 - PORT_STARTIO initiate routine • 17–14
 - PORT_STOP service routine • 17–15
 - PORT_XOFF service routine • 17–16
 - PORT_XON service routine • 17–16
 - Position independent code • 5–1
 - Postprocessing
 - See I/O postprocessing
 - Power bit
 - See UCBSV_POWER
 - Power failure
 - blocking • 3–6
 - determining the occurrence of • 8–5
 - occurring when device is busy • A–56
 - on I/O bus • 18–6
 - servicing in an initialization routine • 11–1, 11–5
 - servicing in port driver unit initialization routine • 17–11, 17–20
 - Power failure recovery procedure • A–23, A–24, A–52
 - device timeout forced by • 10–5
 - initialization performed by • 11–4 to 11–5
 - PR\$_ASTLVL processor register • 3–4
 - PR\$_SID processor register • A–16
 - PR\$_SIRR processor register • 3–8, B–62
 - PR\$_TBIA processor register • G–16
 - PR\$_TBIS processor register • G–16
 - Prefetch function of UNIBUS adapter • 12–3, 12–12, 12–13
 - Preprocessing
 - See I/O preprocessing
 - Preprocessing routine
 - See FDT routine
 - Primary bootstrap program (VMB) • G–22
 - Primary processor • G–2, G–22, G–25
 - Printer driver
 - description • 2–1 to 2–7
 - Process
 - See also Process quota
 - current • A–14
 - privilege mask • A–41
 - quantum end event • 3–7
 - returning control from driver to • 4–14
 - scheduling • G–24
 - Process context • 1–7, 2–4, 4–12, 7–1
 - returning to • 4–18
 - Process I/O channel • 11–6, A–11, A–39
 - assigning • 4–3
 - assigning to template device • 11–11
 - deassigning • 11–6, 11–7, 17–12, D–3
 - reference count • A–55, A–56
 - validating • 2–3, 4–4, C–100
 - Processor state
 - See Multiprocessor state
 - Processor status longword
 - See PSL
 - Processor subtype • B–8
 - Processor type • B–8
 - Process quota
 - adjusting • 4–17
 - buffered I/O • 2–3, 2–7, 4–7
 - byte count • 7–7
 - charging • 4–7, 4–10, A–40, D–15
 - direct I/O • 4–7
 - Programmed I/O
 - See PIO transfer
 - \$PRTCTEND macro • 14–11
 - \$PRTCTINI macro • 14–11
 - PSL (processor status longword)
 - examining with XDELTA • 16–10
 - Z condition code • C–27
 - PURDPR macro • 12–24, B–46, C–79
 - detecting memory errors using • 12–25
-
- ## Q
-
- Q22 bus • 1–15, B–3
 - accomplishing a DMA transfer on • 12–15 to 12–16, 12–19 to 12–26
 - address size • 12–5
 - device interrupt dispatching • 12–34 to 12–36, A–20
 - example of driver designed for • E–1 to E–29, F–1 to F–25
 - I/O address space • 18–1, 18–3, 18–6
 - I/O space • 12–4
 - power failure • 18–6
 - rules for configuring • 1–15, 12–35 to 12–36
 - scatter-gather map • 12–4 to 12–7
 - Q22 bus interface
 - functions • 12–1 to 12–14
 - obtaining resources of • 12–15
 - QBUS_MULT_INTR parameter • 12–34

Index

Quantum end event • 3-7
QUEUEAST spin lock • 3-12, C-7
Queue operations
 in multiprocessing environment • G-14 to G-15
Quota
 See Process quota, Job quota

R

Random access device • A-53
Rank
 of spin lock • 3-14
Read check
 enabling • A-53
Read function • A-39, A-40
 FDT routine for • 7-8
 postprocessing for • C-70
READ_SYSTIME macro • B-47, G-15
 example • B-47
Real time device • A-53, A-54
REALTIME_SPTS parameter • 18-8
Record oriented device • A-53
Reentrant code • 5-1
Register dumping routine • 1-4, 11-9, 11-10,
 A-29, A-62, B-46, C-9, C-67, C-79
 address • 6-3, D-14
 context • D-14
 entry point • D-14
 exit method • D-14
 for generic VAXBI device • 14-19
 functions • D-14
 input • D-14
 register usage • D-14
 synchronization requirements • D-14
Registers
 See BIOC registers, Device registers, General
 purpose registers, Map registers
REI instruction
 role in AST delivery • 3-4
Reinitialization table • 6-2, 15-8, A-33, B-24
RELALT macro • 12-26, B-48, C-81
RELCHAN macro • 10-2, 13-14, B-49, C-83
RELDPR macro • 12-25, B-50, C-84
RELMPR macro • 12-25, B-51, C-86
RELSCHAN macro • B-52, C-88
Remote terminal UCB extension • A-54
REQALT macro • 12-10, 12-19, C-89
REQCOM macro • 10-3, B-54, C-91
 required for error logging • 11-9
REQDPR macro • 12-11, 12-17, B-55, C-93
REQMPR macro • 12-10, 12-11, 12-19, B-56,
 C-95
REQPCAN macro • 3-24, 8-3 to 8-4, 13-6,
 13-12, B-57, C-97
REQSCHN macro • 13-6, 13-13, B-58, C-97
Resource wait flag
 See PCB\$V_SSRWAIT
Resource wait mode • 4-7, C-12, C-20, C-22
Resource wait queue • 3-23 to 3-24, G-14
 See also Alternate map register wait queue,
 Device controller data channel wait queue
 See also Map register wait queue, Secondary
 data channel wait queue, Data path wait
 queue
 buffered data path • C-85
Retry count • 10-6
RLO1 driver • E-1 to E-29
RLO2 driver • E-1 to E-29
RL11 driver • E-1 to E-29
RSB instruction • 7-3
RUN processor state • A-15, G-21

S

SAVIPL macro • 3-9, B-59
SBI (synchronous backplane interconnect) • 1-10
 UNIBUS interlock sequence to • 12-10
SBICONF array • 14-6
Scatter-gather map • 12-4
 See also Map registers
SCB (system control block) • 14-9, A-7
 of VAX 6200 series • 14-9
 of VAX 8200/8250/8300/8350 • 14-9
 of VAX 8550/8700/8800/8830/8840 • 14-9
SCH\$GL_COMQS • G-24
SCH\$GL_CURPCB
 replaced in VMS Version 5.0 • G-7
SCH\$GL_PCBVEC • 16-13
SCH\$POSTEF • A-38
SCH\$QAST • 3-4
SCH\$RESCHED • 3-7
SCHED spin lock • 3-4, 3-7, 3-12, C-19, G-24
Scheduler • G-24
 blocking activity of • 3-5
 synchronization of • 3-7
SCS (system communications services) • A-32
SDA
 See System Dump Analyzer

- SDA current process • G-19
- \$SECDEF macro • 18-6
- Secondary bootstrap program (SYSBOOT) • 16-20, G-23
- Secondary controller data channel • 13-12, 13-14, B-52
 - obtaining ownership of • B-58, C-97 to C-98
 - releasing • C-88
- Secondary controller data channel wait queue • C-88, C-98
- Secondary processor • G-21
- Seek operation • 8-6
 - overlapping with data transfer • 8-3
- Selected map register
 - See MBA\$L_SMR
- Self-test status • 14-23
- Sense device characteristics function • 7-8
- Sense device mode function • 7-8
- Set device characteristics function • 7-8, A-54, A-55
- Set device mode function • 7-8, A-54, A-55
- SETIPL macro • 3-8, 3-9, B-60, G-4
 - example • B-61
 - replacing with spin lock synchronization macro • G-13
- Set mode function • A-55
- SET PROCESS command • G-18
- Shareable device • A-53
- SHOW DEVICE command • A-58
- SHOW SPINLOCKS command • G-17
- SIRR (software interrupt request register) • 3-8
- SMP\$ACQNOIPL • 16-25, B-15, G-18
- SMP\$ACQUIRE • 16-25, B-32, B-44, G-18
- SMP\$ACQUIREL • 16-25, B-15, G-18
- SMP\$AR_IPLVEC • B-31, C-26, C-30
- SMP\$AR_SPNLKVEC • 3-12, A-46, B-32, B-44, B-66
- SMP\$GL_FLAGS • 15-10, G-3
- SMP\$RELEASE • 16-25, B-33, B-66, G-18
- SMP\$RELEASEL • 16-25, B-17, G-18
- SMP\$RESTORE • 16-25, 16-26, B-33, B-66, G-18
- SMP\$RESTOREL • 16-25, 16-26, B-17, G-18
- SMP\$SETUP_CPU • G-23
- SMP\$SETUP_SMP • G-23
- SMP\$V_UNMOD_DRIVER • 15-10, G-3
- SMP_CPUS parameter • G-21, G-23
- SOFTINT macro • 3-9, B-62, C-26, C-30
- Software timer • G-25 to G-26
- Software timer interrupt service routine • 3-7, 10-4
- Solicited interrupt
 - See Device interrupt
- Spin lock • 1-7, 3-2, 3-11 to 3-15
 - See also Device lock, Fork lock, SPL, Spin lock index, Spin wait
 - acquisition IPL • 3-10, 3-13, A-46, C-107, G-17, G-19
 - acquisition PC list • A-47, G-17
 - address • G-19
 - dynamic • 3-12, A-47
 - multiple acquisition of • 3-14, B-66, C-111, G-19
 - name • G-19
 - obtaining • 3-9, B-44 to B-45, C-107
 - ownership • 3-14, 16-26, A-46, A-47, G-19
 - rank • 3-12 to 3-13, 3-14, 3-15, A-46, G-17, G-19
 - releasing • 3-9, B-66, C-109
 - restoring • B-66, C-111
 - static • 3-12, A-47
 - status • G-19
 - system • 3-12, A-47
- Spin lock data structure
 - See SPL
- Spin lock index • 3-12, 3-12 to 3-13, G-19
- Spin lock IPL vector
 - See SMP\$AR_IPLVEC
- Spin lock synchronization macros • G-4, G-13
 - See also DEVICELOCK, DEVICEUNLOCK, FORKLOCK, FORKUNLOCK, LOCK, and UNLOCK
- Spin wait • 3-14, A-47, C-106, C-107, C-108
- SPL\$_IPL • 3-8, A-56, G-17
- SPL\$_RANK • G-17
- SPL\$_BUSY_WAITS • G-17
- SPL\$_OWN_PC_VEC • G-17
- SPL\$_ACQ_COUNT • G-17
- SPL (spin lock data structure) • A-45 to A-47
- SPLACQERR bugcheck • 16-25, 16-26, C-107, G-18
- \$SPLCODDEF macro • B-21, B-24, G-9
- SPLIPLHIGH bugcheck • 16-25, C-107, C-108, G-18
- SPLIPLLOW bugcheck • 16-25, C-109, C-110, C-111, C-112, G-18
- SPLRELERR bugcheck • 16-25, 16-26, C-109, C-110, G-18
- SPLRSTERR bugcheck • 16-25, 16-26, C-111, C-112, G-18
- Spooled device • A-53
- SPTREQ parameter • C-16

Index

- SS\$_ABORT • 10–6
- SS\$_ACCVIO • C–32, C–35, C–41, C–43, C–46, C–49, C–50, C–54, C–55, C–58, C–71, C–77, C–78
- SS\$_BADPARAM • C–32, C–35, C–41, C–43, C–45, C–54, C–55, C–57, C–103
- SS\$_CANCEL • 11–6
- SS\$_EXQUOTA • C–6, C–20, C–22, G–6
- SS\$_ILLIOFUNC • C–50
- SS\$_INSFMAPREG • C–62
- SS\$_INSFMEM • C–6, C–12, C–14, C–15, C–16, C–51, C–59
- SS\$_INSFSPTS • C–16, C–103
- SS\$_INSFWSL • C–32, C–35, C–41, C–46, C–58
- SS\$_IVCHAN • C–100
- SS\$_MBFULL • C–51, C–59
- SS\$_MBTOOSML • C–51, C–59
- SS\$_NONSMPPDRV • G–4
- SS\$_NOPRIV • C–51, C–59, C–100
- SS\$_SSFAIL • C–62, C–73, C–82, C–90
- Stack
 - device driver use of • 8–1
 - using for temporary storage • 5–3
- START/CPU command • G–21, G–23
- Start I/O routine • 1–3
 - See also Alternate start I/O routine
 - activating • C–28
 - address • 2–4, 6–3, A–29, D–15
 - checking for zero length buffer • C–32, C–41, C–54
 - context • 4–12 to 4–13, 8–1 to 8–2, D–15
 - entry point • D–15
 - exit method • D–16
 - for connect to interrupt facility • 18–10, 18–15
 - for MASSBUS device • 13–12
 - for MicroVAX I device driver • 12–26
 - functions • 4–13 to 4–14
 - input • D–15
 - of CONINTERR.EXE • 18–13
 - reactivating • 4–15 to 4–16
 - register usage • 8–1, D–15
 - suspending • 4–14
 - synchronization requirements • 3–6, 3–19, 8–5, D–15, G–9 to G–11
 - transferring control to • 4–12 to 4–13, 8–1, 10–3, C–38, C–68 to C–69
 - writing • 8–1 to 8–7
- Static spin lock • 3–12
- Status register
 - See CSR, MBA\$_SR
- STOP/CPU command • G–22
- STOPPED processor state • A–15
- STOPPING processor state • A–15
- Streamlined synchronization image • 16–25
 - loading • G–2
- Subcontroller • A–32
- Swapping I/O function • A–39
- SWI\$GL_FQFL
 - replaced by CPU\$Q_SWIQFL • G–15
- Symbol list
 - defining • B–27 to B–28
- Synchronization image • G–23
 - full-checking • 16–25, G–2, G–17
 - streamlined • 16–25, G–2
 - uniprocessing • 16–25, G–2
- Synchronization techniques • 1–7, 3–1 to 3–24
 - See also IPL, Spin lock, Fork queue, and Resource wait queue
- Synchronous backplane interconnect
 - See SBI
- Synchronous communications device • A–54
- SYS\$ALLOC • A–52, A–56
- SYS\$AL_JOBCTLMB
 - replaced by SYS\$AR_JOBCTLMB • G–7
- SYS\$AL_OPRMBX
 - replaced by SYS\$AR_OPRMBX • G–7
- SYS\$AR_JOBCTLMB • 9–7, G–7
- SYS\$AR_OPRMBX • 10–7, G–7
- SYS\$ASSIGN • 1–6, 2–3, 4–3, 18–8, A–11, A–55, A–56
 - for template device • D–5
- SYS\$CANCEL • 1–4, 11–6, 11–7, 17–15, 18–18, A–29, D–3
- SYS\$CRMPSC • 18–5 to 18–6, 18–7
- SYS\$DALLOC • 11–7, 17–15, A–29, A–55, A–56, D–3
- SYS\$DASSGN • 11–6, 11–7, 17–15, A–29, A–55, D–3
- SYS\$LOADABLE_IMAGES directory • G–8
- SYS\$QIO • 1–1, 2–2 to 2–4, 4–1 to 4–13, A–36
 - device-dependent arguments of • A–40
 - for connect to interrupt facility • 18–8, 18–9 to 18–12
- SYS\$QIOW • 2–7, A–36
- SYS\$SYNCH • 2–7
- SYSBOOT
 - See Secondary bootstrap program
- System buffer
 - See Buffer, Nonpaged pool
- System configuration • 15–9

System context • 1–8
 System control block
 See SCB
 System Dump Analyzer (SDA) • 16–20
 current process • G–18
 SET CPU command • G–19
 SHOW CPU command • G–19
 SHOW CRASH command • G–19
 SHOW SPINLOCKS command • G–19
 using to debug device driver • 16–26
 System failure
 inducing with XDELTA • 16–20
 System Generation Utility (SYSGEN) •
 15–2 to 15–20
 AUTOCONFIGURE command • 11–3 to 11–4,
 14–20, 15–11 to 15–20, A–2, A–33,
 A–47, B–20, D–19
 CONNECT command • 11–3 to 11–4, 14–20,
 15–2, 15–3 to 15–6, A–6, A–24, A–35,
 A–43, A–47, B–20, D–7, D–21, G–3
 /ADAPTER qualifier • 15–5
 /ADPUNIT qualifier • 15–6
 /CSR qualifier • 15–5
 /CSR_OFFSET qualifier • 15–5
 /DRIVERNAME qualifier • 15–6
 /MAXUNITS qualifier • 15–6
 /NOADAPTER qualifier • 15–5
 /NUMVEC qualifier • 12–32, 12–33, 15–6,
 A–21
 /VECTOR qualifier • 15–6
 /VECTOR_OFFSET qualifier • 15–6
 device table • 15–12 to 15–13, 15–20
 LOAD command • 11–3, 15–2 to 15–3, G–3
 loading a VAXBI device driver using •
 14–20 to 14–21
 RELOAD command • 11–4, 15–7 to 15–8, D–9
 SHOW/ADAPTER command • 15–8
 SHOW/CONFIGURATION command • 15–9
 SHOW/DEVICE command • 15–9 to 15–10
 System initialization • G–22 to G–24
 System map (SYS\$SYSTEM:SYS.MAP) • 16–20
 System page-table entry
 allocating • 14–15, C–103, G–7
 allocating permanent • 6–2, A–32, A–58, B–19,
 C–77, C–78
 deallocating • C–104
 System resource
 accessing • B–44 to B–45
 System service dispatcher
 role in servicing I/O request • 4–1
 System spin lock • 3–12
 System time • 3–7, 3–13, C–67, G–14, G–25

System time (cont'd.)
 reading • B–47, G–15, G–26

T

Tape driver • A–53, D–12
 using local tape UCB extension • A–48,
 A–60 to A–61
 Template device • 11–11
 Template for a device driver • 5–6 to 5–15
 Template UCB • A–56, A–57
 Terminal • A–53, A–54
 See also Terminal controller, Terminal class
 driver, Terminal port driver, Terminal UCB
 extension
 detached • A–54
 I/O function for • A–39
 redirected • A–54
 Terminal class driver • 17–1 to 17–21
 binding to port driver • 17–8, B–7
 service routines • 17–17 to 17–21
 structure • 17–6
 Terminal controller • A–19
 Terminal port driver • 17–1 to 17–21, B–6
 aborting output activity in • 17–15
 binding to class driver • 17–8, B–7
 canceling I/O request in • 17–15
 control flags • A–67
 detecting an error on terminal line in • 17–21
 disconnecting a process from a terminal in •
 17–18
 forking in • 17–13, 17–18
 implementing modem functions in • 17–14
 initiate routines • 17–11 to 17–14
 managing data set state transitions in • 17–18
 obtaining characters for output in • 17–19
 passing input characters to class driver from •
 17–19
 resuming stopped output in • 17–15
 service routines • 17–15 to 17–17
 starting output on an inactive line in • 17–14
 startup routines • 17–10 to 17–11
 stopping output in • 17–15
 structure • 17–6
 using input flow control character in • 17–16
 Terminal UCB extension • 17–2 to 17–3, A–48,
 A–62 to A–69
 initializing • 17–20
 remote • A–54

Index

Time
 reading system • B-47

TIMEDWAIT macro • B-64 to B-65
 See also TIMEWAIT macro
 example • B-65

Timekeeping • G-25 to G-26

Timeout • A-56, B-72
 caused by power failure recovery procedure • 10-5
 detecting • A-57
 disabling • 4-14, 10-1, B-40, C-30
 due time • A-57
 expected • A-56, C-102
 logging • 10-6, 11-9

Timeout enable bit
 See UCB\$V_TIM

Timeout handling routine • 1-3, 3-7, 9-4, 10-4 to 10-7, 11-8, B-72, D-4
 aborting an I/O request in • 10-6
 address • 8-7, 10-1, D-17
 context • 10-4, D-17
 entry point • D-17
 exit method • D-18
 functions • 10-5, D-18
 input • D-18
 register usage • D-17
 retrying an I/O operation in • 10-5 to 10-6
 synchronization requirements • 3-19, D-17, G-13

Timeout interval • B-72
 specifying • 10-4

Timer
 See Software timer, Interval clock

Timer queue • 3-13, C-29, G-14, G-25

Timer queue element
 See TQE

TIMER spin lock • 3-7, 3-12, C-29, G-14, G-25

TIMEWAIT macro • B-63
 See also TIMEDWAIT macro
 example • B-63

TIMOUT processor state • A-15, G-21

TIMOUT_CRASH processor state • G-22

TQEQ_TIME • C-29

TQE (timer queue element)
 calling a driver from • G-16
 expiration time • 3-7, C-29
 inserting in timer queue • C-29

Translation buffer
 invalidating • B-38 to B-39, G-16

TTDRIVER.EXE • 17-1

TTY\$V_PC_NOTIME • 17-14

TTY\$V_PC_PORTFDT • 17-13

TTY\$V_TP_ABORT • 17-17

\$TTYDEFS macro • 17-2

\$TTYMACS macro • 17-11, B-6, B-7, B-67, B-68, B-69

\$TTYMDMDEF macro • 17-18

\$TTYMODEMDEF macro • 17-11

\$TTYUCBDEF macro • A-48

U

UBA (UNIBUS adapter) • 1-10
 See also UNIBUS adapter

UBI (UNIBUS interface) • 1-11
 See also UNIBUS adapter

UBMAPEXCED bugcheck • C-73, C-76

UCB\$_DEVCLASS • 6-2, B-24, C-50

UCB\$_DEVTYPE • 6-2, B-24, C-50

UCB\$_DIPL • 3-6, 6-2, 10-4, B-24

UCB\$_ERTCNT • 10-3, C-67, C-91

UCB\$_FIPL • A-51, B-31

UCB\$_FLCK • 3-5, 6-2, 10-1, B-24, B-31
 initializing • G-8

UCB\$_SLAVE • 13-11

UCB\$_SLAVE+1 • 13-11

UCB\$_TP_STAT • 17-17

UCB\$_TT_DEPARI • 17-20

UCB\$_TT_DETTYPE • 17-20

UCB\$_TT_MAINT • 17-13, 17-14

UCB\$_TT_OUTTYPE • 17-14, 17-19, 17-20, 17-21

UCB\$_TT_PARITY • 17-14, 17-20

UCB\$_AFFINITY • C-69

UCB\$_CRB • 11-4, 13-12

UCB\$_DDB • 4-6

UCB\$_DDT • 17-8

UCB\$_DEVCHAR • 6-2, 11-9, B-24

UCB\$_DLCK • 3-20

UCB\$_DUETIM • 4-14, 8-7, 10-5, C-101, C-102

UCB\$_EMB • 10-3, C-8

UCB\$_FPC • 4-13, 4-14, 9-4, 10-1, 10-4

UCB\$_FR3 • 4-13, 4-14, 9-4, 10-1, 10-4

UCB\$_FR4 • 4-13, 4-14, 9-4, 10-1, 10-4

UCB\$_IOQFL • 10-3, C-28, G-14

UCB\$_IRP • 4-4, 10-3, C-69

UCB\$_LINK • 11-4

UCB\$_OPCNT • C-5, C-24, C-91
 adjusted by IOC\$REQCOM • C-92

- UCB\$L_ORB • A-43
- UCB\$L_STS • 2-4, 8-5, 8-7
- UCB\$L_SVAPTE • 4-4, 8-2, 12-21, 13-3, 13-13, 14-16, A-40, C-69, C-77
- UCB\$L_SVFN • B-19, C-65, C-77
- UCB\$L_TT_CLASS • 17-8, B-7
- UCB\$L_TT_GETNXT • 17-8
- UCB\$L_TT_LOGUCB • 17-20
- UCB\$L_TT_OUTADR • 17-14, 17-15, 17-19, 17-20
- UCB\$L_TT_PORT • 17-8, B-7
- UCB\$L_TT_PUTNXT • 17-8
- UCB\$L_TT_RTIMOU • 17-20
- UCB\$L_TT_WFLINK • 17-20
- UCB\$Q_DEVDEPEND • 6-2, C-48, C-50
- UCB\$V_BSY • 2-4, 4-4, 7-5, 10-3, 11-8, C-28, C-66, D-4
- UCB\$V_CANCEL • 10-6, 11-8, C-66, C-69, D-4
- UCB\$V_DELMBX • 17-12
- UCB\$V_ECC • C-65
- UCB\$V_ERLOGIP • 10-3, 11-9, C-8, C-92
- UCB\$V_INT • 8-7, 9-3, 9-7, 10-4, 13-9, 17-14
- UCB\$V_JOB • 9-6, 9-7, 9-8
- UCB\$V_ONLINE • 9-8, 11-2, 14-11, A-35
- UCB\$V_POWER • 8-5, 10-5, 11-1, 17-11
- UCB\$V_TEMPLATE • D-5
- UCB\$V_TIM • 8-7, 10-1, 10-4, B-40, C-30, C-101
- UCB\$V_TIMEOUT • 10-4, C-69, C-101
- UCB\$V_VALID • 9-8
- UCB\$W_BCNT • 8-2, 12-19, 12-21, 13-3, 13-13, 14-16, A-40, A-58, C-62, C-64, C-69
- UCB\$W_BOFF • 8-2, 12-19, 12-21, 12-22, 13-3, 13-13, 14-16, A-40, A-58, C-62, C-64, C-69
- UCB\$W_BUFQUO
 - in mailbox UCB • C-59
- UCB\$W_DEVBUFSIZ • 6-2, C-50
 - in mailbox UCB • C-59
- UCB\$W_DEVSTS • 10-3
- UCB\$W_EC1 • C-65
- UCB\$W_EC2 • C-65
- UCB\$W_ERRCNT • 11-9, C-8
- UCB\$W_QLEN • C-28
- UCB\$W_REFC • 9-6, 9-7, 11-6, D-3
- UCB\$W_TT_CURSOR • 17-20
- UCB\$W_TT_DESPEE • 17-20
- UCB\$W_TT_HOLD • 17-20
- UCB\$W_TT_OUTLEN • 17-14, 17-19, 17-20
- UCB\$W_TT_PRTCTL • 17-13, 17-14
- UCB\$W_TT_SPEED • 17-14, 17-20
- UCB\$W_UNIT • 13-11
- UCB (unit control block) • 1-5, 3-5, 4-4, A-11, A-47 to A-69
 - address • 8-7, 11-4
 - as fork block • 8-7
 - as template • A-57
 - cloned • A-30, A-56
 - creation • 11-3, 13-6, 15-4, 15-18, A-36, A-47
 - dual path extension • A-48
 - error log extension • 11-8, A-48, A-58 to A-60
 - extending • A-48 to A-49
 - initializing • 11-2
 - local disk extension • 11-8, A-48, A-61 to A-62, C-9, C-65
 - local tape extension • 11-8, A-48, A-60 to A-61, C-9
 - logical • A-66
 - number to be created • 6-2
 - physical • A-64
 - reference count • A-56
 - remote terminal extension • A-54
 - size • A-32, A-47 to A-49, A-51, B-20
 - storing data in • 4-4, 5-1
 - synchronizing access to • 2-4, 3-5, 3-15
 - terminal extension • 17-2 to 17-3, A-48, A-62 to A-69
- \$UCBDEF macro • A-48
- UNIBUS
 - accomplishing a DMA transfer on • 12-15 to 12-26
 - address size • 12-5
 - example of driver designed for • E-1 to E-29, F-1 to F-25
 - example of read operation • 12-12 to 12-13, 12-14
 - example of write operation • 12-12, 12-14
 - I/O address space • 18-1, 18-3, 18-6
 - I/O space • 12-4
 - power failure • 18-6
- UNIBUS adapter • 1-11, 1-12
 - error interrupt from • 16-21, 18-6
 - functions • 12-1 to 12-14
 - interrupt service routine • 12-30
 - nexus value of • 15-5
 - obtaining resources of • 12-15
 - prefetch function • 12-12, 12-13
 - registers • 12-15
 - scatter-gather map • 12-4 to 12-7
 - synchronizing access to • 12-2

Index

Uniprocessing device driver
 converting to multiprocessing device driver •
 G-8 to G-20
 incompatibility with multiprocessing device
 driver • 15-10, G-3

Uniprocessing environment
 contrasted with multiprocessing environment •
 3-10, G-1

Uniprocessing synchronization image • 16-25
 loading • G-2

Unit control block
 See UCB

Unit delivery routine • A-2
 address • 6-2, 15-18, A-33, B-20, D-19
 context • 15-18, D-19
 entry point • D-19
 exit method • D-20
 functions • 15-18, D-20
 input • D-19
 output • 15-18
 register usage • D-19
 synchronization requirements • D-19

Unit initialization routine • 1-3, 11-1 to 11-6,
 15-4
 address • 4-4, 6-3, 6-4, 11-1, 12-31,
 A-24, A-29, B-24, D-21
 allocating contiguous physical memory in •
 12-26
 allocating controller data channel in • 8-4, 10-2
 allocating permanent buffered data path in •
 12-18
 allocating permanent map registers in •
 12-20 to 12-21
 context • 11-1, 11-3, D-21
 entry point • D-21
 exit method • D-21
 for connect to interrupt facility • 18-10,
 18-14 to 18-15
 for generic VAXBI device • 14-10, 14-19
 forking in • 3-21, 11-5 to 11-6
 for MASSBUS device • 11-4, 13-11, A-24
 for MicroVAX I device • 12-26
 for terminal port driver • 17-8, 17-11
 functions • 11-2, D-22
 input • 11-3, D-21
 of CONINTERR.EXE • 18-14
 of terminal port driver • B-7
 register usage • D-21
 synchronization requirements • D-21, G-12

UNLOCK macro • 3-9, B-66, C-109, C-111, G-4

Unsolicted interrupt
 See Device interrupt

Unsolicted interrupt service routine • 9-5, 13-14,
 A-29
 address • 6-3, D-23
 context • D-23
 entry point • D-23
 exit method • D-23
 input • D-23
 register usage • D-23
 synchronization requirements • D-23

UNSUPRTCPU bugcheck • B-9

User interface CSR space
 enabling interrupts from • 14-13

V

VAX-11/725
 See VAX-11/730
 booting with XDELTA from • 16-4

VAX-11/730 • 1-12
 booting with XDELTA from • 16-4

VAX-11/750 • 1-11
 booting with XDELTA from • 16-2

VAX-11/780 • 1-10
 booting with XDELTA from • 16-4
 requesting an XDELTA interrupt from • 16-8

VAX-11/785
 See VAX-11/780
 booting with XDELTA from • 16-4
 requesting an XDELTA interrupt from • 16-8

VAX 6200 series • 1-12 to 1-14
 booting with XDELTA from • 16-2
 requesting an XDELTA interrupt from • 16-8

VAX 8200 • 1-12 to 1-14
 booting with XDELTA from • 16-3, 16-8

VAX 8250
 See VAX 8200
 booting with XDELTA from • 16-3, 16-8

VAX 8300
 See VAX 8200
 booting with XDELTA from • 16-3, 16-8

VAX 8350
 See VAX 8200
 booting with XDELTA from • 16-3, 16-8

VAX 8530 • 1-12 to 1-14
 booting with XDELTA from • 16-3
 requesting an XDELTA interrupt from • 16-8

- VAX 8550
 - See VAX 8530
 - booting with XDELTA from • 16–3
 - requesting an XDELTA interrupt from • 16–8
 - VAX 8600 • 1–10
 - booting with XDELTA from • 16–4
 - requesting an XDELTA interrupt from • 16–8
 - VAX 8650
 - See VAX 8600
 - booting with XDELTA from • 16–4
 - requesting an XDELTA interrupt from • 16–8
 - VAX 8670
 - See VAX 8600
 - booting with XDELTA from • 16–4
 - requesting an XDELTA interrupt from • 16–8
 - VAX 8700
 - See VAX 8530
 - booting with XDELTA from • 16–3
 - requesting an XDELTA interrupt from • 16–8
 - VAX 8800 • 1–12 to 1–14
 - booting with XDELTA from • 16–3
 - requesting an XDELTA interrupt from • 16–8
 - VAX 8830
 - booting with XDELTA from • 16–3
 - requesting an XDELTA interrupt from • 16–8
 - VAX 8840
 - booting with XDELTA from • 16–3
 - requesting an XDELTA interrupt from • 16–8
 - VAXBI bus • 1–12
 - address • 14–2 to 14–5
 - arbitration mode of • 14–23
 - errors • 14–24
 - I/O address space • 14–2, 14–14, 18–1
 - master of • 14–8
 - memory space • 14–2
 - VAXBI node
 - See also Generic VAXBI device, Node ID definition • 14–1
 - determining self-test status of • 14–11
 - enabling BIIC options on • 14–13
 - enabling error interrupts from • 14–13
 - mapping window space of • 14–14 to 14–15, C–103
 - setting interrupt destination of • 14–12
 - setting interrupt vector for • 14–13
 - VAX MACRO instructions
 - as used in device driver • 5–1 to 5–4
 - VCB (volume control block) • A–52, A–56
 - VEC\$B_DATAPATH • 12–17, 12–18, 12–21, 12–25
 - VEC\$B_NUMREG • 12–20
 - VEC\$L_IDB • 4–4, 13–12
 - VEC\$L_INITIAL • 4–4, 15–4, D–7
 - VEC\$L_ISR • 4–4, D–12, G–5
 - VEC\$L_RTINTD • 12–35, 12–36
 - VEC\$L_UNITINIT • 4–4, 15–4, D–21
 - VEC\$Q_DISPATCH • A–23
 - VEC\$V_LWAE • 12–14, 12–21, C–76
 - VEC\$V_MAPLOCK • 12–20, C–87
 - VEC\$V_PATHLOCK • 12–17, 12–18, C–84
 - VEC\$W_MAPALT • 12–20, 12–23
 - VEC\$W_MAPREG • 12–20, 12–22
 - VEC\$W_NUMALT • 12–20
 - VEC (interrupt transfer vector) • 12–30, 12–31, 12–31 to 12–33, A–8, A–20 to A–25
 - initializing • 12–32
 - multiple • A–21
 - \$VECEND macro • 17–6, B–68
 - example • B–69
 - \$VECINI macro • 17–6, B–67, B–69
 - example • B–69
 - \$VEC macro • 17–6, B–67
 - example • B–69
 - VECTAB
 - See Adapter dispatch table
 - Vector
 - fixed space • 15–12
 - floating space • 15–12
 - Vector jump table
 - See Adapter dispatch table
 - _VIELD macro • A–48, B–70 to B–71
 - example • B–71
 - \$VIELD macro • B–70 to B–71
 - VIRTCONS spin lock • 3–13
 - Virtual address
 - translating to physical address • 12–26
 - Virtual address register
 - See MBA\$L_VAR
 - Virtual I/O function • A–39, A–41
 - translation to logical function from • 2–3
 - VMB
 - See Primary bootstrap program
 - Volume • A–56
 - Volume valid bit
 - See UCB\$V_VALID
-
- W
-
- Wait for interrupt macro
 - See WFIKPCH macro, WFIRLCH macro
 - WCB (window control block) • 4–8, A–11, A–38

Index

WFIKPCH macro • 4-14, 8-5, 8-6, 10-7, 13-13,
B-61, B-72 to B-73, C-101, D-17, G-11

WFIRLCH macro • 4-14, 8-5, 8-6,
B-72 to B-73, C-101, D-17

Window control block

See WCB

Window space • 14-5

mapping • 14-14 to 14-15

starting address • 14-14

Word count register • 12-23

Working set limit • C-35, C-41

insufficient • C-32

Workstation device • A-54

Write check

enabling • A-53

Write function

FDT routine for • 7-8

X

XADRIVER.MAR • F-1 to F-25

XDELTA

See Delta/XDelta Utility

XDELTA entry IPL • 3-8

XQP (extended QIO processor) • A-11, A-52

default • A-27

Reader's Comments

VMS Device Support
AA-LA88A-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

| I rate this manual's: | Excellent | Good | Fair | Poor |
|--|--------------------------|--------------------------|--------------------------|--------------------------|
| Accuracy (software works as manual says) | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Completeness (enough information) | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Clarity (easy to understand) | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Organization (structure of subject matter) | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Figures (useful) | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Examples (useful) | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Index (ability to find topic) | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Page layout (easy to find information) | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

I would like to see more/less _____

What I like best about this manual is _____

What I like least about this manual is _____

I found the following errors in this manual:

| Page | Description |
|-------|-------------|
| _____ | _____ |
| _____ | _____ |
| _____ | _____ |
| _____ | _____ |
| _____ | _____ |

Additional comments or suggestions to improve this manual:

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

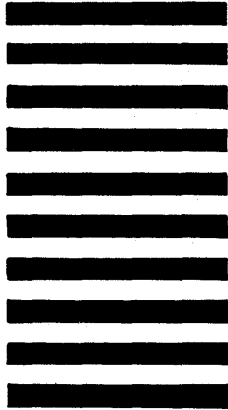
Phone _____

— Do Not Tear - Fold Here and Tape —

digitalTM



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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987



— Do Not Tear - Fold Here —

Cut Along Dotted Line