OpenVMS VAX Device Support Reference Manual

Order Number: AA–PWC9A–TE

March 1994

This manual provides the reference material for the *OpenVMS VAX Device Support Manual*, which describes how to write a driver for a device connected to a VAX processor. This manual describes the data structures, macros, and routines used in device driver programming.

Revision/Update Information: This manual supersedes the *OpenVMS VAX Device Support Reference Manual*, Version 6.0.

Software Version: OpenVMS VAX Version 6.1

Digital Equipment Corporation
Maynard, Massachusetts

March 1994

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

ZK5503

This document is available on CD–ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

Send Us Your Comments

We welcome your comments on this or any other OpenVMS manual. If you have suggestions for improving a particular section or find any errors, please indicate the title, order number, chapter, section, and page number (if available). We also welcome more general comments. Your input is valuable in improving future releases of our documentation.

You can send comments to us in the following ways:

- Internet electronic mail: `OPENVMSDOC@ZKO.MTS.DEC.COM`
- Fax: `603-881-0120 Attn: OpenVMS Documentation, ZKO3-4/U08`
- A completed Reader's Comments form (postage paid, if mailed in the United States), or a letter, via the postal service. Two Reader's Comments forms are located at the back of each printed OpenVMS manual. Please send letters and forms to:

  Digital Equipment Corporation
  Information Design and Consulting
  OpenVMS Documentation
  110 Spit Brook Road, ZKO3-4/U08
  Nashua, NH 03062-2698
  USA

You may also use an online questionnaire to give us feedback. Print or edit the online file SYS$HELP:OPENVMSDOC_SURVEY.TXT. Send the completed online file by electronic mail to our Internet address, or send the completed hardcopy survey by fax or through the postal service.

Thank you.

Contents

1 Data Structures

2 System Macros Invoked by Drivers

3 Operating System Routines

4 Device Driver Entry Points

Index

Figures

Tables

Preface

The *OpenVMS VAX Device Support Reference Manual* provides the reference material for the *OpenVMS VAX Device Support Manual*, which describes how to write a driver for a device connected to a VAX processor. This manual describes the data structures, macros, and routines used in driver programming.

This manual provides information you need to write a device driver that runs under OpenVMS VAX Version 6.1 and to load the driver into the operating system. Digital does not guarantee that drivers written for earlier versions of the operating system 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.

The use of internal executive interfaces other than those described in this manual are discouraged.

Intended Audience

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

Document Structure

This manual contains the following four chapters:

Chapter 1 contains a set of figures and tables that describe the contents of each data structure in the I/O database.

Chapter 2 lists the system macros most frequently called by device drivers.

Chapter 3 describes the context, synchronization, and I/O requirements of the operating system routines used by drivers or called as the result of a driver macro invocation.

Chapter 4 supplies a condensed description of the function and environment of each device driver entry point routine.

Associated Documents

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

- The *OpenVMS VAX Device Support Manual* is the driver programming companion document
- *VAX Hardware Handbook*
- I/O-related portions of the *OpenVMS System Services Reference Manual*

- The section on operating system naming conventions in the *Guide to Creating OpenVMS Modular Procedures*
- *OpenVMS I/O User's Reference Manual*

Other useful information can be found in your processor's hardware documentation, as well as that in the following documents:

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

Conventions

In this manual, every use of OpenVMS VAX means the OpenVMS VAX operating system.

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.

The following conventions are also used in this manual:

. . .	Horizontal ellipsis points in examples indicate one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	Vertical ellipsis points indicate the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In command format descriptions, brackets indicate optional elements. You can choose one, none, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file specification or in the syntax of a substring specification in an assignment statement.)
{ }	In command format descriptions, braces surround a required choice of options; you must choose one of the options listed.

boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason (user action that triggers a callback). Boldface text is also used to show user input in Bookreader versions of the manual.
italic text	Italic text emphasizes important information and indicates complete titles of manuals and variables. Variables include information that varies in system messages (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *device-name* contains up to five alphanumeric characters).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
-	A hyphen in code examples indicates that additional arguments to the request are provided on the line that follows.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1

Data Structures

This chapter 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 chapter—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 1–1.) The structures in the **I/O database** help the operating system and device drivers monitor the status 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

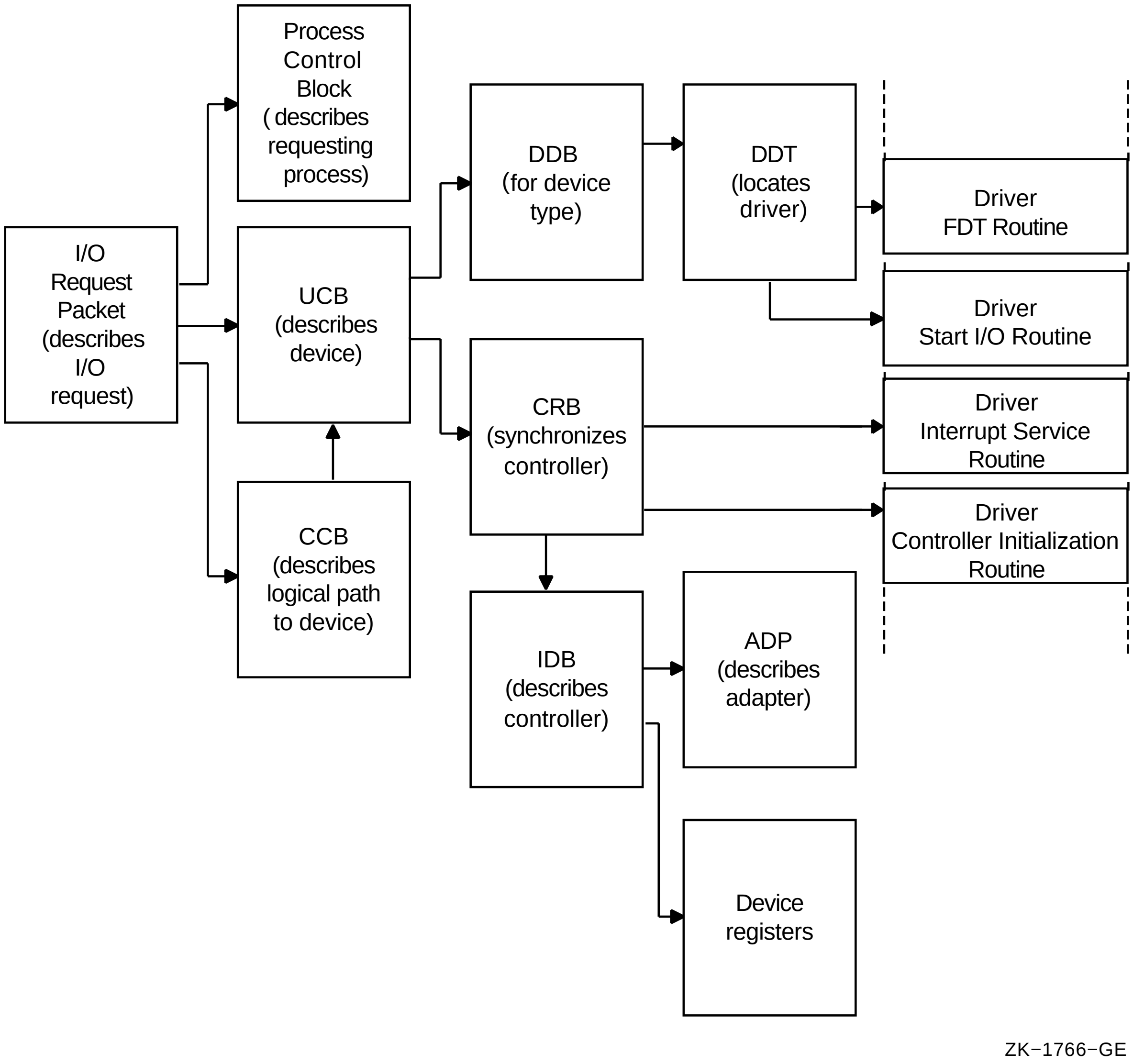
Aside from the I/O database structures, this chapter includes descriptions of those data structures used by the operating system to maintain multiprocessing synchronization and record processor-specific information: the spinlock data structure (SPL) and the per-CPU database structure (CPU), respectively. This chapter also describes the structures that implement the SCSI port interface that supports the creation of SCSI class driver, as well as those structures used to access the control registers of a device attached to a remote bus (CRAM and CRAMH).

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, never numeric offsets, from the beginning of the structure. Numeric offsets are likely to change with each new release of the operating system. The figures in this chapter list numeric offsets only as an aid in driver debugging.

Figure 1–1 I/O Database



1.1 Configuration Control Block (ACF)

The configuration control block (ACF) is used by the System Generation utility (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 1–2 and described in Table 1–1. An asterisk (*) indicates a read-only field in tables and figures.

Figure 1–2 Configuration Control Block (ACF)

<table>
<tr><td colspan="4">ACF$L_ADAPTER*</td><td>0</td></tr>
<tr><td colspan="4">ACF$L_CONFIGREG*</td><td>4</td></tr>
<tr><td>ACF$B_AFLAG*</td><td>ACF$B_AUNIT*</td><td colspan="2">ACF$W_AVECTOR*</td><td>8</td></tr>
<tr><td colspan="4">ACF$L_CONTRLREG*</td><td>12</td></tr>
<tr><td colspan="2">ACF$W_CUNIT*</td><td colspan="2">ACF$W_CVECTOR*</td><td>16</td></tr>
<tr><td colspan="4">ACF$L_DEVNAME*</td><td>20</td></tr>
<tr><td colspan="4">ACF$L_DRVNAME*</td><td>24</td></tr>
<tr><td>ACF$B_COMBO_VEC*</td><td>ACF$B_CNUMVEC*</td><td colspan="2">ACF$W_MAXUNITS*</td><td>28</td></tr>
<tr><td colspan="2">Unused</td><td>ACF$B_NUMUNIT*</td><td>ACF$B_COMBO_CSR*</td><td>32</td></tr>
<tr><td colspan="4">ACF$L_DLVR_SCRH</td><td>36</td></tr>
</table>

*A read-only field

Table 1–1 Contents of Configuration Control Block

Field Name	Contents	
ACF$L_ADAPTER*	Address of ADP for adapter currently being configured.	
ACF$L_CONFIGREG*	Address of configuration register for adapter currently being configured.	
ACF$W_AVECTOR*	Offset from base of SCB to interrupt vector of adapter currently being configured.	
ACF$B_AUNIT*	Adapter unit number of device or controller currently being configured.	
ACF$B_AFLAG*	Flags associated with autoconfiguration operation. Flags defined in this field include the following:	
	ACF$V_RELOAD	Reloading driver code.
	ACF$V_CRBBLT	CRB and IDB already built for device.
	ACF$V_SCBVEC	CVECTOR is offset into SCB.
	ACF$V_NOLOAD_DB	Do not load I/O database, only load driver.
	ACF$V_SUPPORT	VMS-supported device.
	ACF$V_GETDONE	Addresses of data structures in I/O database have been obtained.
	ACF$V_BVP	Multiport BVP adapter.
ACF$L_CONTRLREG*	Address of CSR for controller currently being configured.	
ACF$W_CVECTOR*	Offset into ADP vector table to longword that contains transfer address of interrupt vector used by controller currently being configured (if ACF$V_SCBVEC is not set). If ACF$V_SCBVEC is set, this field is the offset from the SCB base to the interrupt vector of the controller currently being configured.	
ACF$B_CUNIT*	Unit number of device currently being configured.	
ACF$L_DEVNAME*	Address of counted ASCII string that gives name of controller currently being configured.	
ACF$L_DRVNAME*	Address of counted ASCII string that gives driver name for controller currently being configured.	
ACF$W_MAXUNITS*	Maximum number of units that can be connected to controller currently being configured.	
ACF$B_CNUMVEC*	Number of interrupt vectors to configure for controller currently being configured.	
ACF$B_COMBO_VEC*	Offset to vectors for combo device. (The name of this field is ACF$B_COMBO_VECTOR_OFFSET.)	
ACF$B_COMBO_CSR*	Offset to start of control registers of combo device. (The name of this field is ACF$B_COMBO_CSR_OFFSET.)	
ACF$B_NUMUNIT*	Number of units to be configured for controller currently being configured.	
ACF$L_DLVR_SCRH	Field available for use by unit delivery routine. SYSGEN never alters this field.	

1.2 Adapter Control Block (ADP)

Each MASSBUS adapter, UNIBUS adapter, Q22–bus, and VAXBI node configured in a VAX system is represented to the operating system 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.

Depending upon the type of I/O adapter being described, the ADP size is variable and subject to the length of the bus-specific ADP extension. Table 1–2 defines the fields that appear in a UNIBUS ADP; these fields are pictured in Figure 1–3. Bus-specific extensions start at offset ADP$L_HOSTNODE in the ADP.

Figure 1–3 Adapter Control Block (ADP)

<table>
<tr><td colspan="4">ADP$L_CSR*</td><td>0</td></tr>
<tr><td colspan="4">ADP$L_LINK*</td><td>4</td></tr>
<tr><td>ADP$B_NUMBER*</td><td>ADP$B_TYPE*</td><td colspan="2">ADP$W_SIZE*</td><td>8</td></tr>
<tr><td colspan="2">ADP$W_ADPTYPE*</td><td colspan="2">ADP$W_TR*</td><td>12</td></tr>
<tr><td colspan="4">ADP$L_VECTOR*</td><td>16</td></tr>
<tr><td colspan="4">ADP$L_DPQFL*</td><td>20</td></tr>
<tr><td colspan="4">ADP$L_DPQBL*</td><td>24</td></tr>
<tr><td colspan="4">ADP$L_AVECTOR*</td><td>28</td></tr>
<tr><td colspan="4">ADP$L_BI_IDR*</td><td>32</td></tr>
<tr><td colspan="2">ADP$W_BI_VECTOR*</td><td colspan="2">ADP$W_BI_FLAGS*</td><td>36</td></tr>
<tr><td colspan="4">ADP$L_SCB_PAGE*</td><td>40</td></tr>
<tr><td colspan="4">ADP$L_BIMASTER*</td><td>44</td></tr>
<tr><td>ADP$B_ADDR_BITS*</td><td>Unused</td><td colspan="2">ADP$W_ADPDISP_FLAGS*</td><td>48</td></tr>
<tr><td colspan="4">ADP$L_PSWITCH_CALLBACK*</td><td>52</td></tr>
</table>

Note that the bus-specific ADP extension begins here (for example, a UNIBUS Adapter).

<table>
<tr><td>ADP$L_MRQFL (ADP$L_HOSTNODE)*</td><td>56</td></tr>
<tr><td>ADP$L_MRQBL*</td><td>60</td></tr>
<tr><td>≈ ADP$L_INTD_UBA* (12 bytes) ≈</td><td>64</td></tr>
</table>

(continued on next page)

ADP$L_UBASCB* (16 bytes)		76
ADP$L_UBASPTE*		92
ADP$L_MRACTMDRS*		100
ADP$W_MRNFENCE*	ADP$W_DPBITMAP*	104
ADP$W_MRNREGARY* (248 bytes)		108
	ADP$W_MRFFENCE*	356
ADP$W_MRFREGARY* (248 bytes)		
ADP$W_UMR_DIS*		604
ADP$L_MR2QFL*		608
ADP$L_MR2QBL*		612
ADP$L_MR2ACTMDR*		616
ADP$W_MR2NFENCE*	Unused	620
ADP$W_MR2NREGAR* (248 bytes)		624
	ADP$W_MR2FFENCE*	872
ADP$W_MR2FREGAR* (248 bytes)		
ADP$W_UMR2_DIS*		1120
ADP$L_MR2ADDR*		1124
ADP$L_VMEADP*		1128

*A read-only field

Table 1–2 Contents of Adapter Control Block

Field Name	Contents
ADP$L_CSR*	Virtual address of adapter configuration register. For a generic VAXBI adapter, this field contains the address of the base of the adapter's node space. The system adapter initialization routine writes this field. The configuration register marks the base of adapter register space, an area that contains data path registers, map registers, or any other registers appropriate to the implementation of the adapter. If the adapter resides on a remote bus connected to a VAX 7000-series or VAX 10000-series system, this field contains the pseudo CSR address (PCA) of the configuration register. The PCA uniquely describes a specific register of a specific node on a specific bus.
ADP$L_LINK*	Address of next ADP. The system adapter initialization routine writes this field. A value of 0 indicates that this is the last ADP.
ADP$W_SIZE*	Size of ADP. The system adapter initialization routine writes this field when the routine creates the ADP. For nondirect-vector UNIBUS adapters, ADP$W_SIZE includes the space allocated for the four UNIBUS interrupt service routines (for BR4 to BR7) and the vector jump table.
ADP$B_TYPE*	Type of data structure. The system adapter initialization routine writes the symbolic constant DYN$C_ADP into this field when the routine creates the ADP.
ADP$B_NUMBER*	Number of this type of adapter (for example, the number for a third MASSBUS adapter is 2). The system adapter initialization routine writes this field when the routine creates the ADP.
ADP$W_TR*	Nexus number of adapter. The system 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$W_ADPTYPE*	Type of adapter. The system adapter initialization routine writes the symbolic constant AT$_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.
ADP$L_VECTOR*	Address of adapter dispatch table. The table is 512 bytes of longword vectors that correspond to device interrupt vectors (0_8–777_8). On VAX processors that handle direct-vector interrupts, ADP$L_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$L_INTD). (The actual stored value is CRB$L_INTD+1, the set low bit of the address indicating that the interrupt stack is to be used in servicing interrupts.) On VAX processors that handle non-direct-vector interrupts, ADP$L_VECTOR points to a page allocated from nonpaged pool called the adapter dispatch table (or vector jump table). Each longword in the page that corresponds to a vector in use contains the address of the controller's interrupt dispatcher (CRB$L_INTD+2). When the UNIBUS adapter interrupts on behalf of a UNIBUS device, the UNIBUS adapter interrupt service routine saves R0 through R5, determines the vector address of the interrupting device, indexes into the vector-jump table, and jumps to the instruction at CRB$L_INTD+2.

(continued on next page)

Table 1–2 (Cont.) Contents of Adapter Control Block

Field Name	Contents
	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.
ADP$L_DPQFL*	Data path wait queue forward link. IOC$REQDATAP and IOC$RELDATAP read and write this field. When a driver fork process requests a buffered data path and none is currently available, IOC$REQDATAP saves driver context in the device's UCB fork block, inserts the fork block address in the data path wait queue, and suspends the driver fork process. When another driver calls IOC$RELDATAP to release a buffered data path, the routine dequeues a UCB fork block address from the data path wait queue, allocates a data path to the driver, and reactivates that driver fork process. This field is also known as ADP$L_MBASCB. For MASSBUS adapters and generic VAXBI adapters, the system 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$L_MBASCB to refresh the SCB vectors. The actual stored value is CRB$L_INTD+1, the set low bit of the address indicating that the interrupt stack is to be used in servicing interrupts.
ADP$L_DPQBL*	Data path wait queue backward link. IOC$REQDATAP and IOC$RELDATAP read and write this field. This field is also known as ADP$L_MBASPTE. For generic VAXBI adapters, the system 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$L_MBASPTE to restore SPTE values and remap node space following a power failure.
ADP$L_AVECTOR*	Address of first SCB vector for adapter.
ADP$L_BI_IDR*	Longword mask specifying, by a single set bit, which VAXBI node is the destination of interrupts from this adapter. In VAX 82*x*0/83*x*0 systems, the VAXBI node of the primary processor becomes the destination for interrupts; in VAX 85*x*0/8700/88*x*0 and VAX 6000-series systems, it is the VAXBI node at which the memory-interconnect-to-VAXBI adapter (NBIB, PBIB, or DWMBA/B) resides.
ADP$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 6000-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: ADP$V_ADPDISP_INIT — ADPDISP flags have been initialized ADP$V_ADAP_MAPPING — Adapter mapping supported

(continued on next page)

Table 1–2 (Cont.) Contents of Adapter Control Block

Field Name	Contents	
	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$V_QBUS	Q22–bus adapter
	ADP$V_PSWITCH_COMMIT	XMI adapters committed to primary switch
	ADP$V_CRAMIO	Performs CRAM I/O
	<15:11>	Reserved to Digital
ADP$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_PSWITCH_ CALLBACK*	Address of primary switch XMI callback	

Field Name	Contents
UNIBUS Adapter Extension	
ADP$L_HOSTNODE*	The offset to the bus-specific ADP extension.
ADP$L_MRQFL*	Standard map register wait queue's forward link and the first longword in the UNIBUS adapter extension. 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_UBA*	Interrupt transfer vector. The system 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.
ADP$L_UBASCB*	Series of four longwords that contain SCB entry values, one for each bus request (BR) level or interrupt vector. The UNIBUS adapter power failure recovery procedure uses these values.
ADP$L_UBASPTE*	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.

(continued on next page)

Table 1–2 (Cont.) Contents of Adapter Control Block

Field Name	Contents
UNIBUS Adapter Extension	
ADP$L_MRACTMDRS*	Number of active standard map register descriptors in arrays to which ADP$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 system 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$W_MRNREGARY*	Standard map register "number of registers" array of 124 words. The number of words, or cells, that are active in this array is contained in ADP$L_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$L_MRACTMDRS. Each active cell gives a number of the first free map register within a block of free map registers. For each active cell in this array, there is a corresponding cell in the "number of registers" array (ADP$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$W_UMR_DIS*	Number of disabled standard map registers. During system initialization, some standard map registers can be disabled so that their corresponding UNIBUS and Q22–bus addresses can be accessed directly through UNIBUS-space or Q22–bus-space physical addresses.
ADP$L_MR2QFL*	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. 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.

(continued on next page)

Table 1–2 (Cont.) Contents of Adapter Control Block

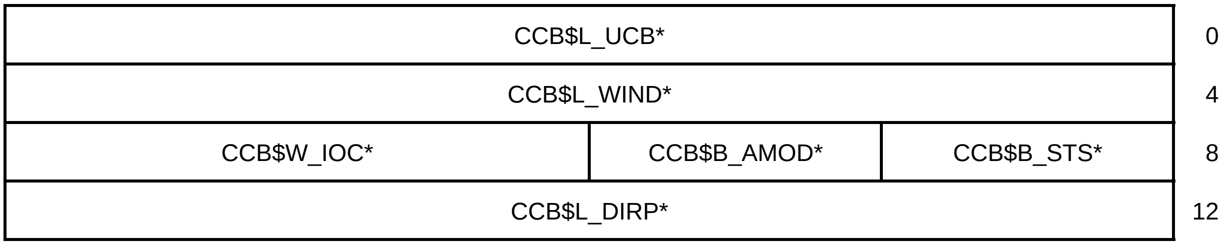
Field Name	Contents
UNIBUS Adapter Extension	
ADP$L_MR2QBL*	Alternate-map-register wait queue's backward link. IOC$ALOALTMAP, IOC$REQALTMAP, and IOC$RELALTMAP read and write this field when allocating and deallocating from the set of Q22–bus alternate map registers.
ADP$L_MR2ACTMDR*	Number of active map register descriptors in arrays to which ADP$W_MR2NREGAR and ADP$W_MR2FREGAR point. IOC$ALOALTMAP, IOC$REQALTMAP, and IOC$RELMAPREG use these fields when allocating and deallocating Q22–bus alternate map registers.
ADP$W_MR2NFENCE*	Boundary marker for the array specified by ADP$W_MR2NREGAR; contains –1.
ADP$W_MR2NREGAR*	Alternate-map-register "number of registers" array of 124 words. The number of words, or cells, that are active in this array is contained in ADP$L_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$W_MR2FREGAR. Together, these values give the base map register and the number of free map registers for a block of free alternate map registers. IOC$ALOALTMAP, IOC$REQALTMAP, and IOC$RELALTMAP use this information when allocating and deallocating from Q22–bus alternate map registers.
ADP$W_MR2FFENCE*	Boundary marker for the array specified by ADP$W_MR2NREGAR; contains –1.
ADP$W_MR2FREGAR*	Alternate map register "first register" array of 124 words. The number of words, or cells, that are active in this array is contained in ADP$L_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$W_MR2NREGAR. Together, these values give the base map register and the number of free map registers for a block of free map registers.
ADP$W_UMR2_DIS*	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.
ADP$L_MR2ADDR	Address of the first Q22–bus alternate map register mapped in CPU node private space. The value varies for each processor with alternate map registers. IOC$LOADUBAMAP reads this field when accessing alternate map registers.
ADP$L_VMEADP*	VME adapter type identifier.

1.3 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 preallocated channel control blocks (CCBs) of the process. EXE$ASSIGN then writes into the CCB a description of the device attached to the CCB's channel.

The channel control block is the only data structure described in this chapter that exists in the control (P1) region of a process address space. It is illustrated in Figure 1–4 and described in Table 1–3.

Figure 1–4 Channel Control Block (CCB)



*A read-only field

Table 1–3 Contents of Channel Control Block

Field Name	Contents
CCB$L_UCB*	Address of UCB of assigned device unit. EXE$ASSIGN writes a value into this field. EXE$QIO reads this field to determine that the I/O request specifies a process I/O channel assigned to a device and to obtain the device's UCB address.
CCB$L_WIND*	Address of window control block (WCB) for file-structured device assignment. This field is written by an ACP or XQP and read by EXE$QIO. A file-structured device's XQP or ACP creates a WCB when a process accesses a file on a device assigned to a process I/O channel. The WCB maps the virtual block numbers of the file to a series of physical locations on the device.
CCB$B_STS*	Channel status.
CCB$B_AMOD*	Access mode plus 1 of the channel. EXE$ASSIGN writes the access mode value into this field.
CCB$W_IOC*	Number of outstanding I/O requests on channel. EXE$QIO increases this field when it begins to process an I/O request that specifies the channel. During I/O postprocessing, the special kernel-mode AST routine decrements this field. Some FDT routines and EXE$DASSGN read this field.
CCB$L_DIRP*	Address of IRP for requested deaccess. A number of outstanding I/O requests can be pending on the same process I/O channel at one time. If the process that owns the channel issues an I/O request to deaccess the device, EXE$QIO holds the deaccess request until all other outstanding I/O requests are processed.

1.4 Per-CPU Database (CPU)

A per-CPU database structure exists for each processor in a 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, the operating system 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 1–5 and described in Table 1–4.

Figure 1–5 Per-CPU Database (CPU)

CPU$L_CURPCB*				0
CPU$L_REALSTACK*				4
CPU$B_SUBTYPE*	CPU$B_TYPE*	CPU$W_SIZE*		8
CPU$B_CUR_PRI*	CPU$B_CPUMTX*	CPU$B_STATE*	CPU$B_BUSYWAIT	12
CPU$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_P0BR*				56

(continued on next page)

CPU$L_P0LR*	60
CPU$L_P1BR*	64
CPU$L_P1LR*	68
CPU$L_BUGCODE*	72
CPU$B_CPUDATA* (16 bytes)	76
CPU$L_MCHK_MASK*	92
CPU$L_MCHK_SP*	96
CPU$L_P0PT_PAGE*	100
Reserved (408 bytes)	104
CPU$Q_SWIQFL* (48 bytes)	512
CPU$L_PSFL*	560
CPU$L_PSBL*	564
CPU$Q_WORK_FQFL*	568
CPU$L_QLOST_FQFL*	576
CPU$L_QLOST_FQBL*	580
CPU$B_QLOST_FLCK* CPU$B_QLOST_TYPE* CPU$W_QLOST_SIZE*	584
CPU$L_QLOST_FPC*	588
CPU$L_QLOST_FR3*	592
CPU$L_QLOST_FR4*	596
CPU$Q_BOOT_TIME*	600
CPU$Q_CPUID_MASK*	608
CPU$L_PHY_CPUID*	616
CPU$L_CAPABILITY*	620

(continued on next page)

CPU$L_TENUSEC*		624
CPU$L_UBDELAY*		628
CPU$L_KERNEL* (28 bytes)		632
CPU$L_NULLCPU*		660
CPU$W_UKERNEL* (14 bytes)		664
CPU$W_UNULLCPU*		676
CPU$W_HARDAFF*	CPU$W_CLKUTICS*	680
CPU$L_RANK_VEC*		684
CPU$L_IPL_VEC*		688
CPU$L_IPL_ARRAY* (128 bytes)		692
CPU$L_TPOINTER*		820
CPU$W_SANITY_TICKS*	CPU$W_SANITY_TIMER*	824
CPU$L_VP_OWNER*		828
CPU$L_VP_VARIANT_EXIT*		832
CPU$L_VP_FLAGS*		836
CPU$L_VP_CPUTIM*		840
Reserved	CPU$B_FLAGS*	844
CPU$L_INTFLAGS*		848

*A read-only field

Table 1–4 Contents of Per-CPU Database

Field Name	Contents	
CPU$L_CURPCB*	Address of current PCB. The scheduler writes this field.	
CPU$L_REALSTACK*	Physical address of boot stack.	
CPU$W_SIZE*	Size of the per-CPU database, including the size of the boot stack but not the interrupt stack or the interrupt stack's guard pages.	
CPU$B_TYPE*	Type of data structure. The operating system writes the value DYN$C_MP into this field when it creates the per-CPU database.	
CPU$B_SUBTYPE*	Structure subtype. The operating system writes the value DYN$C_MP_CPU into this field when it creates the per-CPU database.	
CPU$B_BUSYWAIT*	Concurrent busy-wait count for this processor.	
CPU$B_STATE*	State of this processor. The following processor states are defined:	
	CPU$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_TIMOUT	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

(continued on next page)

Table 1–4 (Cont.) Contents of Per-CPU Database

Field Name	Contents	
	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
	CPU$V_IOPOST	Request to request IPL 4 software interrupt
	CPU$V_INV_ISTREAM	Invalidate instruction cache
	<31:29>	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.	
CPU$L_SAVED_ISP*	Saved ISP for restart.	
CPU$L_PCBB*	PCBB from power down.	
CPU$L_SCBB*	SCBB from power down.	
CPU$L_SISR*	SISR from power down.	
CPU$L_P0BR*	P0 base register (used by system power failure and bugcheck routines).	
CPU$L_P0LR*	P0 length register (used by system power failure and bugcheck routines).	
CPU$L_P1BR*	P1 base register (used by system power failure and bugcheck routines).	
CPU$L_P1LR*	P1 length register (used by system power failure and bugcheck routines).	
CPU$L_BUGCODE*	Bugcheck code.	
CPU$B_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$L_SID.	
CPU$L_MCHK_MASK*	Function mask for current machine check recovery block.	
CPU$L_MCHK_SP*	Saved SP for return at end of machine check recovery block. This field is zero if there is no current recovery block.	
CPU$L_P0PT_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$Q_SWIQFL*	Twelve longwords representing the forward and backward links for the software interrupt queues (fork IPLs 6 through 11).	
CPU$L_PSFL*	CPU-specific I/O postprocessing queue forward link.	
CPU$L_PSBL*	CPU-specific I/O postprocessing queue backward link.	

(continued on next page)

Table 1–4 (Cont.) Contents of Per-CPU Database

<table>
<tr><th>Field Name</th><th>Contents</th></tr>
<tr><td>CPU$Q_WORK_FQFL*</td><td>Work packet queue. This field is also called CPU$Q_WORK_IFQ.</td></tr>
<tr><td>CPU$L_QLOST_FQFL*</td><td>Quorum loss fork queue forward link.</td></tr>
<tr><td>CPU$L_QLOST_FQBL*</td><td>Quorum loss fork queue blink link.</td></tr>
<tr><td>CPU$W_QLOST_SIZE*</td><td>Quorum loss fork block size.</td></tr>
<tr><td>CPU$B_QLOST_TYPE*</td><td>Quorum loss fork block type.</td></tr>
<tr><td>CPU$B_QLOST_FLCK*</td><td>Quorum loss fork lock.</td></tr>
<tr><td>CPU$L_QLOST_FPC*</td><td>Quorum loss fork PC.</td></tr>
<tr><td>CPU$L_QLOST_FR3*</td><td>Quorum loss fork R3.</td></tr>
<tr><td>CPU$L_QLOST_FR4*</td><td>Quorum loss fork R4.</td></tr>
<tr><td>CPU$Q_BOOT_TIME*</td><td>System time at which this processor was bootstrapped.</td></tr>
<tr><td>CPU$Q_CPUID_MASK*</td><td>Bit mask representing this processor's CPU ID.</td></tr>
<tr><td>CPU$L_PHY_CPUID*</td><td>Integer that uniquely identifies the local processor in a multiprocessor configuration. This value is system specific. (For example, in a VAX 8300/8350 configuration, it is the VAXBI node ID. For a VAX 8800, it is the left or right bit from the processor's system ID register (PR$_SID); for a VAX 8810/8820/8830 it is the CPU number (0 to 3) from PR$_SID. In a VAX 6000-series configuration, it is the XMI node ID. The operating system uses the physical ID principally to locate the per-CPU database and interrupt stack of a processor that it is restarting.)</td></tr>
<tr><td>CPU$L_CAPABILITY*</td><td>Bit mask of this processor's capabilities.
The following capabilities are defined in $CPBDEF:
<table>
<tr><td>CPB$C_PRIMARY</td><td>Primary CPU.</td></tr>
<tr><td>CPB$C_NS</td><td>Reserved to Digital.</td></tr>
<tr><td>CPB$C_QUORUM</td><td>Quorum required.</td></tr>
<tr><td>CPB$C_HARDAFF</td><td>Hard affinity. Reserved for diagnostics software.</td></tr>
</table></td></tr>
<tr><td>CPU$L_TENUSEC*</td><td>10-microsecond delay value.</td></tr>
<tr><td>CPU$L_UBDELAY*</td><td>UNIBUS delay counter.</td></tr>
<tr><td>CPU$L_KERNEL*</td><td>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.</td></tr>
<tr><td>CPU$L_NULLCPU*</td><td>Clock ticks during which the null job has been the current process on this processor.</td></tr>
<tr><td>CPU$W_UKERNEL*</td><td>Reserved to Digital.</td></tr>
<tr><td>CPU$W_UNULLCPU*</td><td>Reserved to Digital.</td></tr>
<tr><td>CPU$W_CLKUTICS*</td><td>Reserved to Digital.</td></tr>
<tr><td>CPU$W_HARDAFF*</td><td>Count of processes with hard affinity for this processor.</td></tr>
<tr><td>CPU$L_RANK_VEC*</td><td>Longword recording the ranks of all spinlocks currently held by the processor. Spinlock 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.</td></tr>
</table>

(continued on next page)

Table 1–4 (Cont.) Contents of Per-CPU Database

Field Name	Contents	
CPU$L_IPL_VEC*	Vector recording, in inverse order, the IPLs of all spinlocks currently held by the processor (that is, bit 0 represents IPL 31).	
CPU$L_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 spinlock acquisition by this processor, the IPL vector corresponding to the spinlock's synchronization IPL (SPL$B_IPL) is incremented.	
CPU$L_TPOINTER*	Address of the sanity timer (CPU$W_SANITY_TIMER) of the active processor with the next highest CPU ID.	
CPU$W_SANITY_TIMER*	Number of sanity cycles before this processor times out.	
CPU$W_SANITY_TICKS*	Number of clock ticks until the next sanity cycle.	
CPU$L_VP_OWNER*	PCB address of the vector consumer.	
CPU$L_VP_VARIANT_EXIT*	Variant exit address to the disabled fault handler.	
CPU$L_VP_FLAGS*	Vector processing flags. The following fields are defined within CPU$L_VP_FLAGS:	
	CPU$V_VP_POWERFAIL	Powerfail variant
	CPU$V_VP_BUGCHECK	Bugcheck variant
	CPU$V_VP_CTX_INIT	Initialization in progress for vector context
	CPU$V_VP_CTX_SAVE	Save in progress for vector context
	CPU$V_VP_CTX_RESTORE	Restore in progress for vector context
CPU$L_VP_CPUTIM*	Scheduled time for a vector consumer.	
CPU$B_FLAGS*	Miscellaneous processor flags. The following fields are defined within CPU$B_FLAGS:	
	CPU$V_SCHED	Idle loop in wait for CPU scheduler
	CPU$V_FOREVER	STOP/CPU with /FOREVER qualifier
	CPU$V_NEWPRIM	Primary-to-be CPU
	CPU$V_PSWITCH	Live primary switch requested by primary CPU
	CPU$V_MMG_HELD	CPU owns MMG spinlock
	CPU$V_VBSS_TRAN	VBSS transition in progress
CPU$L_INTFLAGS*	Interlocked flags. This word contains one flag bit: CPU$V_STOPPING for the CPU stopping indicator.	

1.5 Control Register Access Mailbox (CRAM)

The control register access mailbox (CRAM) holds information that describes a mailbox I/O access of a control and status register of a device attached to a remote bus. The CRAM contains information required by the software as well as the hardware itself. For example, mailbox I/O transactions require the physical address of the hardware mailbox, as well as the virtual address of the corresponding mailbox pointer register (MBPR). Timeout values for the transaction are also stored in the CRAM.

CRAMs are preallocated from system nonpaged memory. Once they have been allocated, they are managed privately by the CRAM allocation and deallocation routines (IOC$ALLOCATE_CRAM and IOC$DEALLOCATE_CRAM). Each block of CRAMs begins with a structure known as a control register access mailbox header (CRAMH). The blocks of CRAMs are maintained as a linked list starting at system global location IOC$GQ_CRAMQ and linked through location CRAM$L_FLINK of the CRAM.

The control register access mailbox is shown in Figure 1–6 and described in Table 1–5.

Figure 1–6 Control Register Access Mailbox (CRAM)

<table>
<tr><td colspan="4">CRAM$L_FLINK</td><td>0</td></tr>
<tr><td colspan="4">CRAM$L_BLINK</td><td>4</td></tr>
<tr><td>CRAM$B_SUBTYPE*</td><td>CRAM$B_TYPE*</td><td colspan="2">CRAM$W_SIZE*</td><td>8</td></tr>
<tr><td colspan="4">CRAM$L_MBPR*</td><td>12</td></tr>
<tr><td colspan="4">CRAM$Q_HW_MBX*</td><td>16</td></tr>
<tr><td colspan="4">CRAM$Q_QUEUE_TIME</td><td>24</td></tr>
<tr><td colspan="4">CRAM$Q_WAIT_TIME</td><td>32</td></tr>
<tr><td colspan="4">CRAM$L_DRIVER</td><td>40</td></tr>
<tr><td colspan="4">CRAM$L_IDB*</td><td>44</td></tr>
<tr><td colspan="4">CRAM$L_UCB*</td><td>48</td></tr>
<tr><td colspan="3">Unused</td><td>CRAM$B_CRAM_FLAGS</td><td>52</td></tr>
<tr><td colspan="4">CRAM$L_CRAMHADDR*</td><td>56</td></tr>
<tr><td colspan="4">Unused</td><td>60</td></tr>
<tr><td colspan="4">≈ Hardware Mailbox Structure (64 bytes) ≈</td><td>64</td></tr>
</table>

*A read-only field

Table 1–5 Contents of Control Register Access Mailbox

Field Name	Contents
CRAM$L_FLINK	Forward link. Reserved for driver use.
CRAM$L_BLINK	Backward link. Reserved for driver use.
CRAM$W_SIZE*	CRAM structure size in bytes.
CRAM$B_TYPE*	Structure type. Set to DYN$C_MISC when the CRAM is allocated.
CRAM$B_SUBTYPE*	Structure subtype. Set to DYN$C_CRAM when the CRAM is allocated.
CRAM$L_MBPR*	Address of mailbox pointer register (MBPR).
CRAM$Q_HW_MBX*	Physical address of hardware mailbox structure at the end of the CRAM.
CRAM$Q_QUEUE_TIME	Timeout value (in nanoseconds) for queuing mailbox operations. This field is initialized to a default value, but can be changed by the driver.
CRAM$Q_WAIT_TIME	Timeout value (in nanoseconds) for waiting for mailbox operation completion. This field is initialized to a default value, but can be changed by the driver.
CRAM$L_DRIVER	This field is reserved for driver use.
CRAM$L_IDB*	Address of Interrupt Dispatch Block (IDB).
CRAM$L_UCB*	Address of Unit Control Block (UCB).
CRAM$B_CRAM_FLAGS	CRAM flags. The only flag defined is CRAM$V_CRAM_IN_USE.
CRAM$L_CRAMHADDR*	Virtual address of the CRAMH structure associated with this CRAM.
Hardware Mailbox Structure	This 16-longword field (described in Section 1.5.1) is used by the hardware to control the physical I/O operation.

1.5.1 Hardware Mailbox Structure

The hardware mailbox structure is part of the control register access mailbox (CRAM), described in Section 1.5. This structure is used by the I/O processor (IOP) hardware to perform the actual CSR access.

The hardware mailbox structure is shown in Figure 1–7 and described in Table 1–6.

Figure 1–7 Hardware Mailbox Structure

<table>
<tr><td colspan="4">HW_CRAM$L_COMMAND</td><td>0</td></tr>
<tr><td>Unused</td><td>HW_CRAM$B_HOSE</td><td>Unused</td><td>HW_CRAM$B_MASK</td><td>4</td></tr>
<tr><td colspan="4">HW_CRAM$Q_RBADR</td><td>8</td></tr>
<tr><td colspan="4">HW_CRAM$Q_WDATA</td><td>16</td></tr>
<tr><td colspan="4">Unused</td><td>24</td></tr>
<tr><td colspan="4">HW_CRAM$Q_RDATA</td><td>32</td></tr>
<tr><td colspan="2"></td><td colspan="2">HW_CRAM$W_MBX_FLAGS</td><td>40</td></tr>
<tr><td colspan="4">HW_CRAM$W_ERROR_BITS</td><td></td></tr>
<tr><td colspan="4">≈ Unused (16 bytes) ≈</td><td>48</td></tr>
</table>

Table 1–6 Contents of the Hardware Mailbox Structure

Field Name	Contents	
HW_CRAM$L_COMMAND	Bus command to the remote I/O interconnect. Specifies either a read or write transaction. The local I/O adapter delivers this command to the remote interconnect to which the target device is connected. The command may also include fields such as address width and data width.	
HW_CRAM$B_MASK	Active byte mask indicating which bytes within the remote bus address are to be written. (The full name of this field is HW_CRAM$B_BYTE_MASK.)	
HW_CRAM$B_HOSE	I/O bus (or hose) number specifying the remote I/O interconnect to be accessed.	
HW_CRAM$Q_RBADR	Remote bus address specifying the physical address of the device interface register.	
HW_CRAM$Q_WDATA	Data to be written.	
HW_CRAM$Q_RDATA	Returned read data.	
HW_CRAM$W_MBX_FLAGS	Flags bitmask. The following flags are defined:	
	HW_CRAM$V_MBX_DONE	Mailbox operation has completed. This bit, when set, indicates that any error bits and the read data field are valid. Note, however, it does not guarantee that a remote write operation has actually completed at the remote device.
	HW_CRAM$V_MBX_ERROR	There was an error in the operation. The CRAM$W_ERROR_BITS field contains additional information.
HW_CRAM$W_ERROR_BITS	Device-specific error bits.	

1.6 Control Register Access Mailbox Header (CRAMH)

The control register access mailbox header (CRAMH) describes a block of control register access mailboxes (CRAMs).

CRAMs are preallocated in a pool in nonpaged system memory, four pages at a time. Each contiguous section of memory is divided into one header (CRAMH) and 15 CRAMs. If a driver attempts to allocate a CRAM when there are none available, another four-page section of nonpaged memory is allocated and divided into one header and 15 CRAMs.

The control register access mailbox header is shown in Figure 1–8 and described in Table 1–7.

Figure 1–8 Control Register Access Mailbox Header (CRAMH)

<table>
<tr><td colspan="3">CRAMH$L_FLINK</td><td>0</td></tr>
<tr><td colspan="3">CRAMH$L_BLINK</td><td>4</td></tr>
<tr><td>CRAMH$B_SUBTYPE</td><td>CRAMH$B_TYPE</td><td>CRAMH$W_SIZE</td><td>8</td></tr>
<tr><td colspan="3">CRAMH$L_MAX</td><td>12</td></tr>
<tr><td colspan="3">CRAMH$Q_PA_BASE</td><td>16</td></tr>
<tr><td colspan="3">CRAMH$L_AVAIL</td><td>24</td></tr>
<tr><td colspan="3">CRAMH$B_MAP (64 bytes)</td><td>28</td></tr>
<tr><td colspan="3">Unused (36 bytes)</td><td>92</td></tr>
</table>

*A read-only field

Table 1–7 Contents of the Control Register Access Mailbox Header

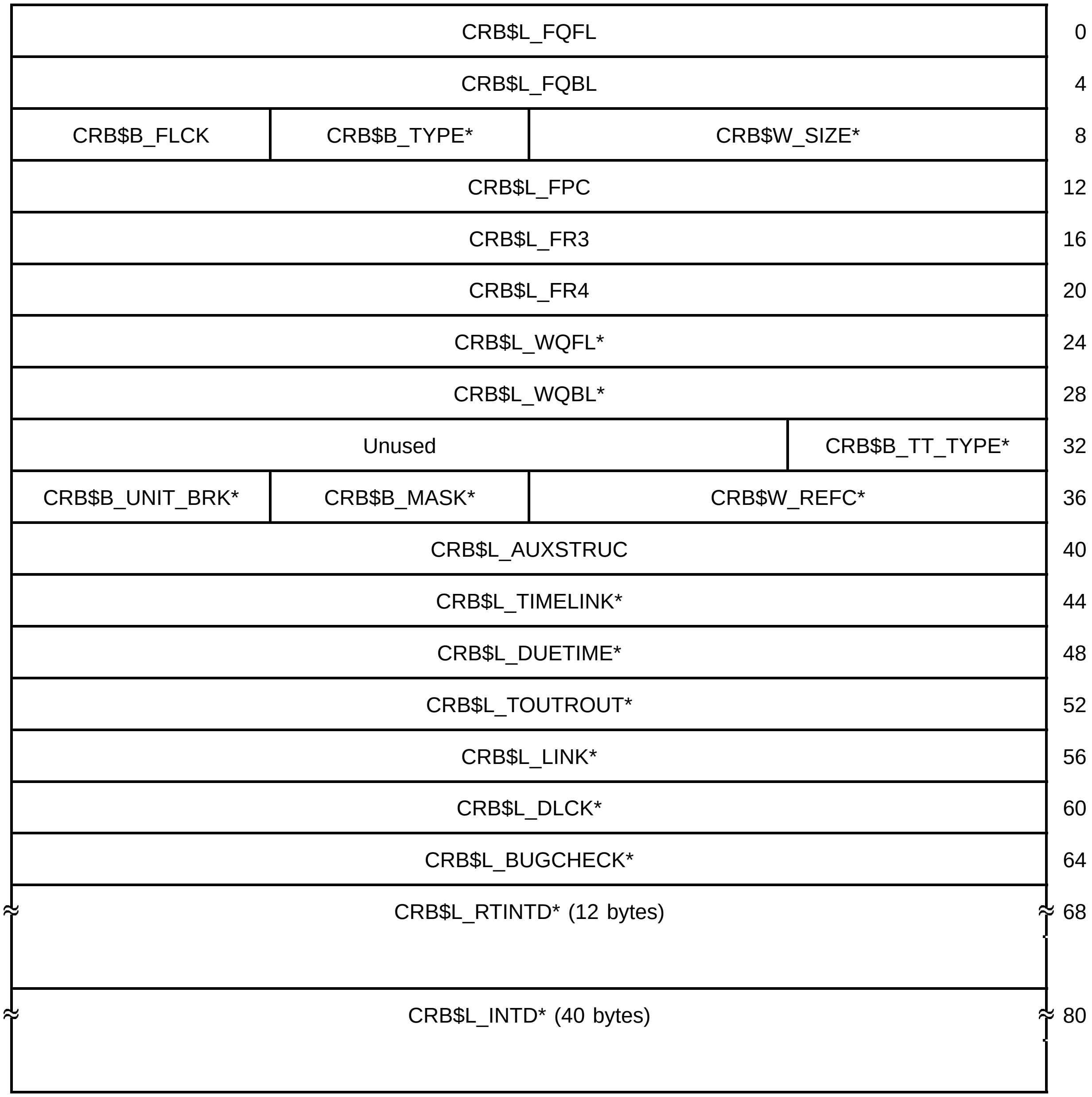
Field Name	Contents
CRAMH$L_FLINK	Forward link. This link points to the CRAMH for the next group of CRAMs.
CRAMH$L_BLINK	Backward link. This link points to the CRAMH for the previous group of CRAMs.
CRAMH$W_SIZE	Structure size in bytes. The CRAMH structure is the same size as a CRAM and takes the first CRAM slot in a page of CRAMs.
CRAMH$B_TYPE	Structure type. Set to DYN$C_MISC when the CRAMH is allocated.
CRAMH$B_SUBTYPE	Structure subtype. Set to DYN$C_CRAMH when the CRAMH is allocated.
CRAMH$L_MAX	Maximum number of CRAMs linked to this CRAMH.
CRAMH$Q_PA_BASE	Base physical address of this block of CRAMs.
CRAMH$L_AVAIL	Total number of mailboxes available in this block.
CRAMH$B_MAP	Usage bitmap. If a bit is set, its corresponding CRAM is available. Up to 64KB of CRAMs can be mapped.

1.7 Channel Request Block (CRB)

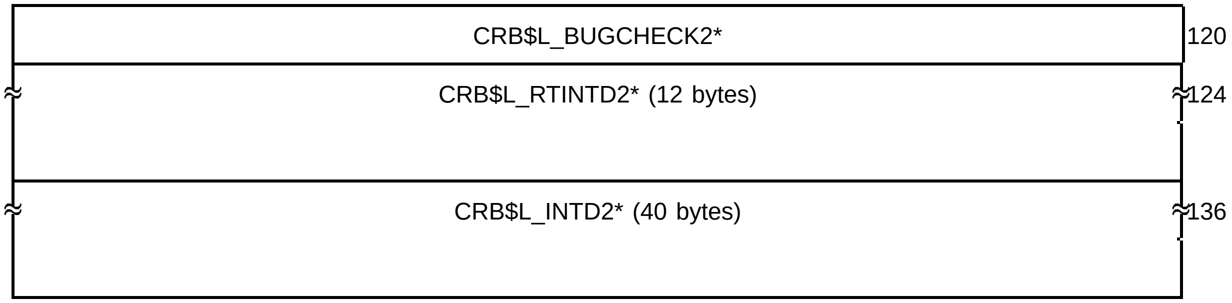
The activity of each controller in a configuration is described in a channel request block (CRB). This data structure contains pointers to the wait queue of drivers ready to gain access to a device through the controller. The CRB also stores the entry points to the driver's interrupt service routines and the unit or controller initialization routine.

The channel request block is illustrated in Figure 1–9 and described in Table 1–8.

Figure 1–9 Channel Request Block (CRB)



(continued on next page)



*A read-only field

Table 1–8 Contents of Channel Request Block

Field Name	Contents
CRB$L_FQFL	Fork queue forward link. The link points to the next entry in the fork queue. Controller initialization routines write this field when they must drop IPL to utilize certain executive routines, such as those that allocate memory, that must be called at a lower IPL. The CRB timeout mechanism also uses the CRB fork block to lower IPL prior to calling the CRB timeout routine.
CRB$L_FQBL	Fork queue backward link. The link points to the previous entry in the fork queue.
CRB$W_SIZE*	Size of CRB. The driver-loading procedure writes this field when it creates the CRB.
CRB$B_TYPE*	Type of data structure. The driver-loading procedure writes the symbolic constant DYN$C_CRB into this field when it creates the CRB.
CRB$B_FLCK	Fork lock at which the controller's fork operations are synchronized. If it must use the CRB fork block, a driver either uses a DPT_STORE macro to initialize this field or explicitly sets its value within the controller initialization routine.
CRB$L_FPC	Address of instruction at which execution resumes when the fork dispatcher dequeues the fork block. EXE$FORK writes this field when called to suspend driver execution.
CRB$L_FR3	Value of R3 at the time that the executing code requests creation of a fork block. EXE$FORK writes this field when called to suspend driver execution.
CRB$L_FR4	Value of R4 at the time that the executing code requests creation of a fork block. EXE$FORK writes this field when called to suspend driver execution.
CRB$L_WQFL*	Controller data channel wait queue forward link. IOC$REQ*x*CHAN*y* and IOC$REL*x*CHAN insert and remove driver fork block addresses in this field. A channel wait queue contains addresses of driver fork blocks that record the context of suspended drivers waiting to gain control of a controller data channel. If a channel is busy when a driver requests access to the channel, IOC$REQ*x*CHAN*y* suspends the driver by saving the driver's context in the device's UCB fork block and inserting the fork block address in the channel wait queue. When a driver releases a channel because an I/O operation no longer needs the channel, IOC$REL*x*CHAN dequeues a driver fork block, allocates the channel to the driver, and reactivates the suspended driver fork process. If no drivers are awaiting the channel, IOC$REL*x*CHAN clears the channel busy bit.

(continued on next page)

Table 1–8 (Cont.) Contents of Channel Request Block

Field Name	Contents
CRB$L_WQBL*	Controller channel wait queue backward link. IOC$REQ*x*CHAN*y* and IOC$REL*x*CHAN read and write this field.
CRB$B_TT_TYPE*	Type of controller (for instance, DZ11 or DZ32) for terminals. A terminal port driver fills in this field.
CRB$W_REFC*	UCB reference count. The driver-loading procedure increases the value in this field each time it creates a UCB for a device attached to the controller.
CRB$B_MASK*	Mask that describes controller status. The following fields are defined in CRB$B_MASK:
	CRB$V_BSY Busy bit. IOC$REQ*x*CHAN*y* reads the busy bit to determine whether the controller is free and sets this bit when it allocates the controller data channel to a driver. IOC$REL*x*CHAN clears the busy bit if no driver is waiting to acquire the channel.
	CRB$V_UNINIT Indication, when set, that the system adapter initialization routine has created a CRB for a generic VAXBI device, but has not yet called its controller initialization routine. SYSGEN reads this bit to determine whether to call the controller initialization routine and clears it when the initialization routine completes. This facilitates SYSGEN's processing of multiunit generic VAXBI devices.
CRB$B_UNIT_BRK*	Break bits for terminal lines. Used by the 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$REQSCHAN*x* and IOC$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 ILLQBUSCFG bugcheck when the multilevel interrupt dispatching code (at CRB$L_RTINTD) determines that a Q22–bus is illegally configured.

(continued on next page)

Table 1–8 (Cont.) Contents of Channel Request Block

Field Name	Contents
CRB$L_RTINTD*	Portion of interrupt transfer vector created at system initialization when a MicroVAX system implements multilevel device interrupt dispatching. The code stored in this 12-byte field implements a conditional lowering to device IPL. See Section 1.7.1 for a description of the contents of the interrupt transfer vector.
CRB$L_INTD*	Portion of the interrupt transfer vector block that stores executable code, driver entry points, and I/O adapter information. This 10-longword area is overlaid with the contents of the interrupt transfer vector block that starts at VEC$L_INTD (offset 16) as described in Section 1.7.1. 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. Although certain of the symbolic offsets defined in the data structure definition macro $VECDEF have negative values, driver code can uniformly refer to the contents of the VEC structure in the following form: CRB$L_INTD+VEC$*x_symbol*.
CRB$L_BUGCHECK2*	Bugcheck data used to issue an ILLQBUSCFG bugcheck when the multilevel interrupt dispatching code (at CRB$L_RTINTD2) determines that the Q22–bus is illegally configured.
CRB$L_RTINTD2*	Portion of second interrupt transfer vector initialized and used if multilevel interrupt dispatching is enabled in a MicroVAX system. See Section 1.7.1 for a description of the contents of the interrupt transfer vector.
CRB$L_INTD2*	Portion of the second interrupt transfer vector block for devices with multiple interrupt vectors. The data structure definition macro $CRBDEF supplies symbolic offsets for only the first two interrupt transfer vector structures.

1.7.1 Interrupt Transfer Vector Block (VEC)

The operating system creates the appropriate number of interrupt transfer vector blocks (VEC) within a CRB if a driver specifies that the addresses of additional interrupt service routines be loaded into these structures. For example:

```
DPT_STORE,CRB,CRB$L_INTD2+VEC$L_ISR,D,isr_for_vec2
DPT_STORE,CRB,CRB$L_INTD+<2*VEC$K_LENGTH>+VEC$L_ISR,D,isr_for_vec3
```

The offset of the *n*th vector located in the CRB is equal to the result of the following formula:

CRB$L_INTD+(*n**VEC$K_LENGTH)

The operating system 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. A single interrupt transfer vector block (VEC) is shown in Figure 1–10. Table 1–9 describes the fields in a VEC block.

Figure 1–10 Interrupt Transfer Vector Block (VEC)

<table>
<tr><td colspan="4">VEC$L_BUGCHECK*</td><td>0</td></tr>
<tr><td colspan="4">VEC$L_RTINTD* (12 bytes)</td><td>4</td></tr>
<tr><td colspan="4">VEC$L_INTD*</td><td>16</td></tr>
<tr><td colspan="4">VEC$L_ISR</td><td>20</td></tr>
<tr><td colspan="4">VEC$L_IDB*</td><td>24</td></tr>
<tr><td colspan="4">VEC$L_INITIAL</td><td>28</td></tr>
<tr><td>VEC$B_DATAPATH</td><td>VEC$B_NUMREG</td><td colspan="2">VEC$W_MAPREG</td><td>32</td></tr>
<tr><td colspan="4">VEC$L_ADP*</td><td>36</td></tr>
<tr><td colspan="4">VEC$L_UNITINIT*</td><td>40</td></tr>
<tr><td colspan="4">VEC$L_START*</td><td>44</td></tr>
<tr><td colspan="4">VEC$L_UNITDISC*</td><td>48</td></tr>
<tr><td colspan="2">VEC$W_NUMALT</td><td colspan="2">VEC$W_MAPALT</td><td>52</td></tr>
</table>

*A read-only field

Table 1–9 Contents of Interrupt Transfer Vector Block (VEC)

Field Name	Contents
VEC$L_BUGCHECK*	Bugcheck data used to issue an ILLQBUSCFG bugcheck when the multilevel interrupt dispatching code determines that the Q22–bus is illegally configured.
VEC$L_RTINTD*	Portion of interrupt transfer vector created at system initialization when a MicroVAX system implements multilevel device interrupt dispatching. The code stored in this 12-byte field implements a conditional lowering to device IPL, as follows:

```
CMPZV   #PSL$V_IPL, #PSL$S_IPL,-
        4(SP), S^#DIPL
BGEQ    BUGCHECK
SETIPL  S^#DIPL
```

(continued on next page)

Table 1–9 (Cont.) Contents of Interrupt Transfer Vector Block (VEC)

Field Name	Contents	
VEC$L_INTD*	Interrupt dispatching code, written by the driver-loading procedure as follows: PUSHR #^M<R0,R1,R2,R3,R4,R5> JSB @# The destination of the JSB instruction is the driver's interrupt service routine, as indicated at offset VEC$L_ISR. Under normal operations, direct-vector UNIBUS or Q22–bus adapters—as well as VAXBI system interrupt dispatching—transfer control to CRB$L_INTD. The code located here causes the processor to execute the PUSHR instruction to save R0 through R5 on the stack and execute a JSB instruction to transfer control to the driver's interrupt service routine. In dispatching interrupts from non-direct-vector UNIBUS adapters, the UNIBUS adapter interrupt service routine transfers control to CRB$L_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. This field, plus VEC$L_ISR, is also known as VEC$Q_DISPATCH.	
VEC$L_ISR	The DPT in every driver for an interrupting device specifies the address of a driver interrupt service routine.	
VEC$L_IDB*	Address of IDB for controller. The driver-loading procedure creates an IDB for each CRB and loads the address of the IDB in this field. Device drivers use the IDB address to obtain the virtual addresses of device registers. When a driver's interrupt service routine gains control, the top of the stack contains a pointer to this field.	
VEC$L_INITIAL	Address of controller initialization routine. If a device controller requires initialization at driver-loading time and during recovery from a power failure, the driver specifies a value for this field in the DPT. The driver-loading procedure calls this routine each time the procedure loads the driver. The power failure recovery procedure also calls this routine to initialize a controller after a power failure.	
VEC$W_MAPREG	The following bits are defined within VEC$W_MAPREG:	
	VEC$V_MAPREG	Number of first 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/Q22–bus transfer.
	VEC$V_MAPLOCK	Map register set is permanently allocated (when set).
VEC$B_NUMREG	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.	

(continued on next page)

Table 1–9 (Cont.) Contents of Interrupt Transfer Vector Block (VEC)

Field Name	Contents	
VEC$B_DATAPATH	Data path specifier. The bits that make up this field are used as follows:	
	VEC$V_DATAPATH	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 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$L_UNITINIT). Other drivers can use either field.	
VEC$L_START*	Address of system start protocol routine. Use of this field is reserved to Digital.	
VEC$L_UNITDISC*	Address of unit disconnect routine. Use of this field is reserved to Digital.	

(continued on next page)

Table 1–9 (Cont.) Contents of Interrupt Transfer Vector Block (VEC)

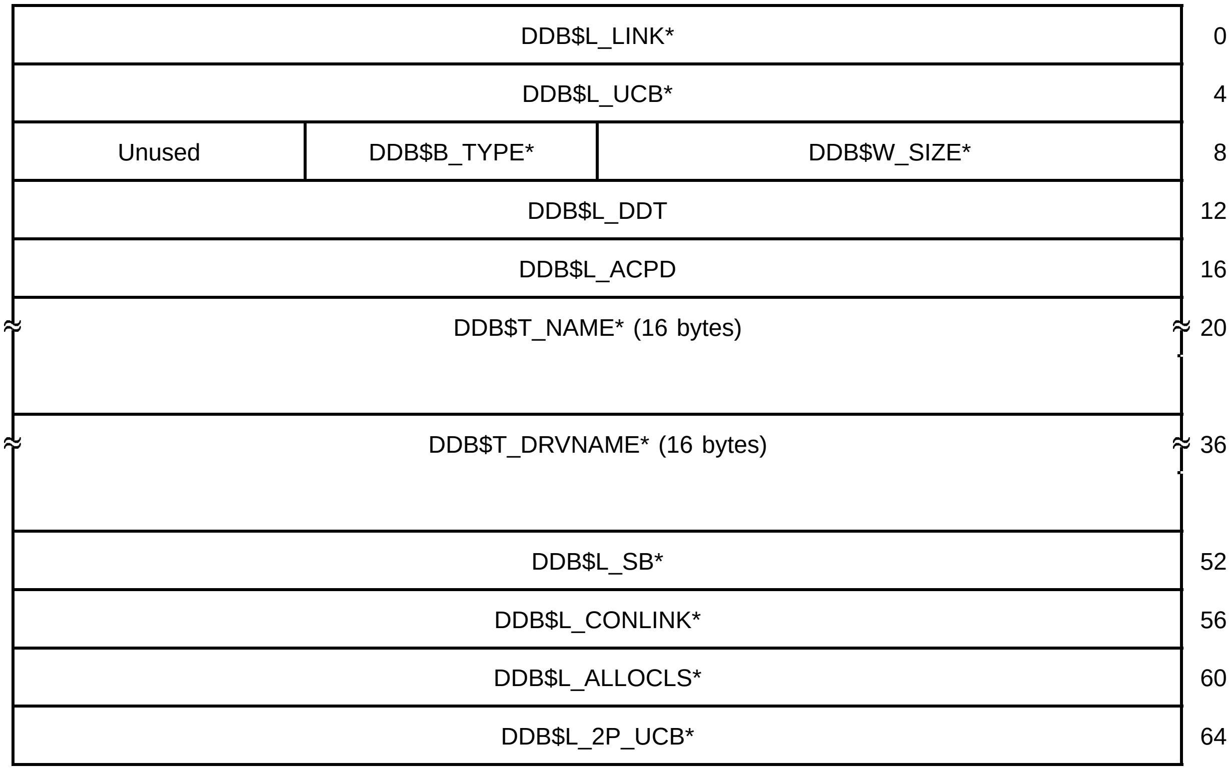
Field Name	Contents	
VEC$W_MAPALT	The following bits are defined within VEC$W_MAPALT:	
	VEC$V_MAPALT	Number of first Q22–bus alternate map register allocated to driver that owns controller data channel. IOC$REQALTMAP writes this field when the routine allocates a set of Q22–bus alternate map registers to a driver fork process for a DMA transfer. IOC$RELMAPREG reads the field to deallocate a set of map registers. Device drivers read this 15-bit field in calculating the starting address of a MicroVAX Q22–bus transfer that uses a set of alternate map registers.
	VEC$V_ALTLOCK	Alternate map register set is permanently allocated (when set).
VEC$W_NUMALT	Number of Q22–bus alternate map registers allocated to driver. IOC$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.	

1.8 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. System routines and device drivers refer to the DDB. 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.

The device data block is illustrated in Figure 1–11 and described in Table 1–10.

Figure 1–11 Device Data Block (DDB)



*A read-only field

Table 1–10 Contents of Device Data Block

Field Name	Contents
DDB$L_LINK*	Address of next DDB. A zero indicates that this is the last DDB in the DDB chain.
DDB$L_UCB*	Address of UCB for first unit attached to controller.
DDB$W_SIZE*	Size of DDB.
DDB$B_TYPE*	Type of data structure. The driver-loading procedure writes the constant DYN$C_DDB into this field when the procedure creates the DDB.
DDB$L_DDT	Address of DDT. The operating system can transfer control to a device driver only through addresses listed in the DDT, the CRB, and the UCB fork block. The DPT of every device driver must specify a value for this field.
DDB$L_ACPD	Name of default ACP (or XQP) for controller. ACPs that control access to file-structured devices (or the XQP) use the high-order byte of this field, DDB$B_ACPCLASS, to indicate the class of the file-structured device. If the ACP_MULTIPLE system parameter is set, the initialization procedure creates a unique ACP for each class of file-structured device. Drivers initialize DDB$B_ACPCLASS by invoking a DPT_STORE macro. Values for DDB$B_ACPCLASS are as follows: DDB$K_PACK Standard disk pack DDB$K_CART Cartridge disk pack DDB$K_SLOW Floppy disk DDB$K_TAPE Magnetic tape that simulates file-structured device
DDB$T_NAME*	Generic name for the devices attached to controller. The first byte of this field is the number of characters in the generic name. The remainder of the field consists of a string of up to 15 characters that, suffixed by a device unit number, identifies devices on the controller.
DDB$T_DRVNAME*	Name of device driver for controller. The first byte of this field is the number of characters in the driver name. The remainder of the field contains a string of up to 15 characters taken from the DPT in the driver.
DDB$L_SB*	Address of system block.
DDB$L_CONLINK*	Address of next DDB in the connection subchain.
DDB$L_ALLOCLS*	Allocation class of device.
DDB$L_2P_UCB*	Address of the first UCB on the secondary path. Another name for this field is DDB$L_DP_UCB.

1.9 Driver Dispatch Table (DDT)

Each device driver contains a driver dispatch table (DDT). The DDT lists entry points in the driver that system routines call, for instance, the entry point for the driver start-I/O routine.

A device driver creates a DDT by invoking the system macro DDTAB. The fields in the driver dispatch table are illustrated in Figure 1–12 and described in Table 1–11.

Figure 1–12 Driver Dispatch Table (DDT)

DDT$L_START		0
DDT$L_UNSOLINT		4
DDT$L_FDT		8
DDT$L_CANCEL		12
DDT$L_REGDUMP		16
DDT$W_ERRORBUF	DDT$W_DIAGBUF	20
DDT$L_UNITINIT		24
DDT$L_ALTSTART		28
DDT$L_MNTVER		32
DDT$L_CLONEDUCB		36
Unused	DDT$W_FDTSIZE*	40
DDT$L_MNTV_SSSC*		44
DDT$L_MNTV_FOR*		48
DDT$L_MNTV_SQD*		52
DDT$L_AUX_STORAGE*		56
DDT$L_AUX_ROUTINE*		60
DDT$L_CHANNEL_ASSIGN*		64
DDT$L_CANCEL_SELECTIVE*		68

*A read-only field

Table 1–11 Contents of Driver Dispatch Table

Field Name	Contents
DDT$L_START	Entry point to the driver's start-I/O routine. Every driver must specify this address in the **start** argument to the DDTAB macro. When a device unit is idle and an I/O request is pending for that unit, IOC$INITIATE transfers control to the address contained in this field.
DDT$L_UNSOLINT	Entry point to a MASSBUS driver's unsolicited-interrupt service routine. The driver specifies this address in the **unsolic** argument to the DDTAB macro. This field contains the address of a routine that analyzes unexpected interrupts from a device. The standard interrupt service routine, the address of which is stored in the CRB, determines whether an interrupt was solicited by a driver. If the interrupt is unsolicited, the interrupt service routine can call the unsolicited-interrupt service routine.
DDT$L_FDT	Address of the driver's FDT. Every driver must specify this address in the **functb** argument to the DDTAB macro. EXE$QIO refers to the FDT to validate I/O function codes, decide which functions are buffered, and call FDT routines associated with function codes.
DDT$L_CANCEL	Entry point to the driver's cancel-I/O routine. The driver specifies this address in the **cancel** argument to the DDTAB macro. Some devices require special cleanup processing when a process or a system routine cancels an I/O request before the I/O operation completes or when the last channel is deassigned. The $DASSGN, $DALLOC, and $CANCEL system services cancel I/O requests.
DDT$L_REGDUMP	Entry point to the driver's register dumping routine. The driver specifies this address in the **regdmp** argument to the DDTAB macro. IOC$DIAGBUFILL, ERL$DEVICERR, and ERL$DEVICTMO call the address contained in this field to write device register contents into a diagnostic buffer or error message buffer.
DDT$W_DIAGBUF	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. 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.
DDT$W_ERRORBUF	Size of error message buffer. The driver specifies this value in the **erlgbf** argument to the DDTAB macro. The value is the size in bytes of an error message buffer for the device. If error logging is enabled and an error occurs during an I/O operation, the driver calls ERL$DEVICERR or ERL$DEVICTMO to allocate and write error-logging data into the error message buffer. IOC$INITIATE and IOC$REQCOM write values into the buffer if an error has occurred.
DDT$L_UNITINIT	Address of the device's unit initialization routine, if one exists. Drivers for MASSBUS devices use this field rather than CRB$L_INTD+VEC$L_UNITINIT. Drivers for UNIBUS, VAXBI, and Q22–bus devices can use either field.
DDT$L_ALTSTART	Address of a driver's alternate start-I/O routine. The EXE$ALTQUEPKT routine transfers control to the alternate start-I/O routine at this address.

(continued on next page)

Table 1–11 (Cont.) Contents of Driver Dispatch Table

Field Name	Contents
DDT$L_MNTVER	Address of the system routine (IOC$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$MNTVER is reserved to Digital.
DDT$L_CLONEDUCB	Address of routine to call when UCB is cloned.
DDT$W_FDTSIZE*	Number of bytes in FDT. The driver-loading procedure uses this field to relocate addresses in the FDT to system virtual addresses.
DDT$L_MNTV_SSSC*	Address of routine to call when performing mount verification for a shadow-set state change. Use of this field is reserved to Digital.
DDT$L_MNTV_FOR*	Address of routine to call when performing mount verification for a foreign device. Use of this field is reserved to Digital.
DDT$L_MNTV_SQD*	Address of routine to call when performing mount verification for a sequential device. Use of this field is reserved to Digital.
DDT$L_AUX_STORAGE*	Address of auxiliary storage area. Use of this field is reserved to Digital.
DDT$L_AUX_ROUTINE*	Address of auxiliary routine. Use of this field is reserved to Digital.
DDT$L_CHANNEL_ASSIGN*	Address of routine to call from $ASSIGN.
DDT$L_CANCEL_SELECTIVE*	Address of selective cancel I/O entry point.

1.10 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 system macros DPTAB and DPT_STORE. The driver prologue table is illustrated in Figure 1–13 and described in Table 1–12.

Figure 1–13 Driver Prologue Table (DPT)

<table>
<tr><td colspan="4">DPT$L_FLINK*</td><td>0</td></tr>
<tr><td colspan="4">DPT$L_BLINK*</td><td>4</td></tr>
<tr><td>DPT$B_REFC*</td><td>DPT$B_TYPE*</td><td colspan="2">DPT$W_SIZE</td><td>8</td></tr>
<tr><td colspan="2">DPT$W_UCBSIZE</td><td>Unused</td><td>DPT$B_ADPTYPE</td><td>12</td></tr>
<tr><td colspan="4">DPT$L_FLAGS</td><td>16</td></tr>
<tr><td colspan="2">DPT$W_REINITTAB</td><td colspan="2">DPT$W_INITTAB</td><td>20</td></tr>
<tr><td colspan="2">DPT$W_MAXUNITS</td><td colspan="2">DPT$W_UNLOAD</td><td>24</td></tr>
</table>

(continued on next page)

DPT$W_DEFUNITS	DPT$W_VERSION*	28
DPT$W_VECTOR	DPT$W_DELIVER	32
≈ DPT$T_NAME (12 bytes) ≈		36
DPT$Q_LINKTIME*		48
DPT$L_ECOLEVEL*		56
DPT$L_UCODE*		60
DPT$Q_LMF_1*		64
DPT$Q_LMF_2*		72
DPT$Q_LMF_3*		80
DPT$Q_LMF_4*		88
DPT$Q_LMF_5*		96
DPT$Q_LMF_6*		104
DPT$Q_LMF_7*		112
DPT$Q_LMF_8*		120
	DPT$W_DECW_SNAME*	

*A read-only field

Table 1–12 Contents of Driver Prologue Table

<table>
<tr><th>Field Name</th><th>Contents</th></tr>
<tr><td>DPT$L_FLINK*</td><td>Forward link to next DPT. The driver-loading procedure writes this field. The procedure links all DPTs in the system in a doubly linked list.</td></tr>
<tr><td>DPT$L_BLINK*</td><td>Backward link to previous DPT. The driver-loading procedure writes this field.</td></tr>
<tr><td>DPT$W_SIZE</td><td>Size in bytes of the driver. The DPTAB macro writes this field by subtracting the address of the beginning of the DPT from the address specified as the end argument to the DPTAB macro. The driver-loading procedure uses this value to determine the space needed in nonpaged system memory to load the driver.</td></tr>
<tr><td>DPT$B_TYPE*</td><td>Type of data structure. The DPTAB macro always writes the symbolic constant DYN$C_DPT into this field.</td></tr>
<tr><td>DPT$B_REFC*</td><td>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.</td></tr>
<tr><td>DPT$B_ADPTYPE</td><td>Type of adapter used by the devices using this driver. Every driver must specify the string "UBA", "MBA", "GENBI", "NULL", or "DR" as the value of the adapter argument to the DPTAB macro. Q22–bus drivers should specify "UBA" as the adapter type. The macro writes the value AT$_UBA, AT$_MBA, or AT$_GENBI in this field.</td></tr>
<tr><td>DPT$W_UCBSIZE</td><td>Size in bytes of the unit control block for a device that uses this driver. Every driver must specify a value for this field in the ucbsize argument to the DPTAB macro.
The driver-loading procedure allocates blocks of nonpaged system memory of the specified size when creating UCBs for devices associated with the driver.</td></tr>
<tr><td>DPT$L_FLAGS</td><td>Driver-loading flags. This field is also known as DPT$B_FLAGS. The driver can specify any of a set of flags as the value of the flags argument to the DPTAB macro. The driver-loading procedure modifies its loading and reloading algorithm based on the settings of these flags.
Flags defined in the flag field include the following:
<table>
<tr><td>DPT$V_SUBCNTRL</td><td>Device is a subcontroller.</td></tr>
<tr><td>DPT$V_SVP</td><td>Device requires permanent system page to be allocated during driver loading.</td></tr>
<tr><td>DPT$V_NOUNLOAD</td><td>Driver cannot be reloaded.</td></tr>
<tr><td>DPT$V_SCS</td><td>SCS code must be loaded with this driver.</td></tr>
<tr><td>DPT$V_DUSHADOW</td><td>Driver is the shadowing disk class driver.</td></tr>
<tr><td>DPT$V_SCSCI</td><td>Common SCS/CI subroutines must be loaded with this driver.</td></tr>
<tr><td>DPT$V_BVPSUBS</td><td>Common BVP subroutines must be loaded with this driver.</td></tr>
<tr><td>DPT$V_UCODE</td><td>Driver has an associated microcode image.</td></tr>
<tr><td>DPT$V_SMPMOD</td><td>Driver has been designed to run in a multiprocessing environment.</td></tr>
<tr><td>DPT$V_DECW_DECODE</td><td>Driver is a decoding class driver.</td></tr>
</table></td></tr>
</table>

(continued on next page)

Table 1–12 (Cont.) Contents of Driver Prologue Table

Field Name	Contents	
	DPT$V_TPALLOC	Select the tape allocation class parameter.
	DPT$V_SNAPSHOT	Driver is certified for system snapshot.
	DPT$V_NO_IDB_DISPATCH	Do not select IDB$L_UCBLST for UCB vectors.
	DPT$V_EXTENDED_DDT	Do not allocate an extended DDT.
	DPT$V_XPAMOD	Driver is compliant with extended addressing (XA).
	DPT$V_VERSION_SAFE	Driver is exempt from version checks.
DPT$W_INITTAB	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. The driver invokes the system macro DPT_STORE to specify these fields and their values.	
DPT$W_REINITTAB	Offset to driver-reinitialization table. Every driver must specify a list of data structure fields and values to be written into these fields at the time that the driver-loading procedure creates the driver's data structures and loads the driver or the driver is reloaded. The driver invokes the system macro DPT_STORE to specify these fields and their values.	
DPT$W_UNLOAD	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.	
DPT$W_MAXUNITS	Maximum number of units on controller that this driver supports. Specify this value in the **maxunits** argument to the DPTAB macro. If no value is specified, the default is eight units.	
DPT$W_VERSION*	Version number that identifies format of DPT. The DPTAB macro automatically inserts a value in this field. SYSGEN checks its copy of the version number against the value stored in this field. If the values do not match, an error is generated. To correct the error, reassemble and relink the driver.	
DPT$W_DEFUNITS	Number of UCBs that the autoconfiguration facility will automatically create. Drivers specify this number with the **defunits** argument to the DPTAB macro. If the driver also gives a value to DPT$W_DELIVER, this field is also the number of times that the autoconfiguration facility calls the unit delivery routine.	
DPT$W_DELIVER	Relative address of the unit delivery routine that the autoconfiguration facility calls for the number of UCBs specified in DPT$W_DEFUNITS. The driver supplies the address of the unit delivery routine in the **deliver** argument to the DPTAB macro.	
DPT$W_VECTOR	Relative address of a driver-specific vector. A terminal class or port driver stores the address of its class or port entry vector table in this field.	
DPT$T_NAME	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.	

(continued on next page)

Table 1–12 (Cont.) Contents of Driver Prologue Table

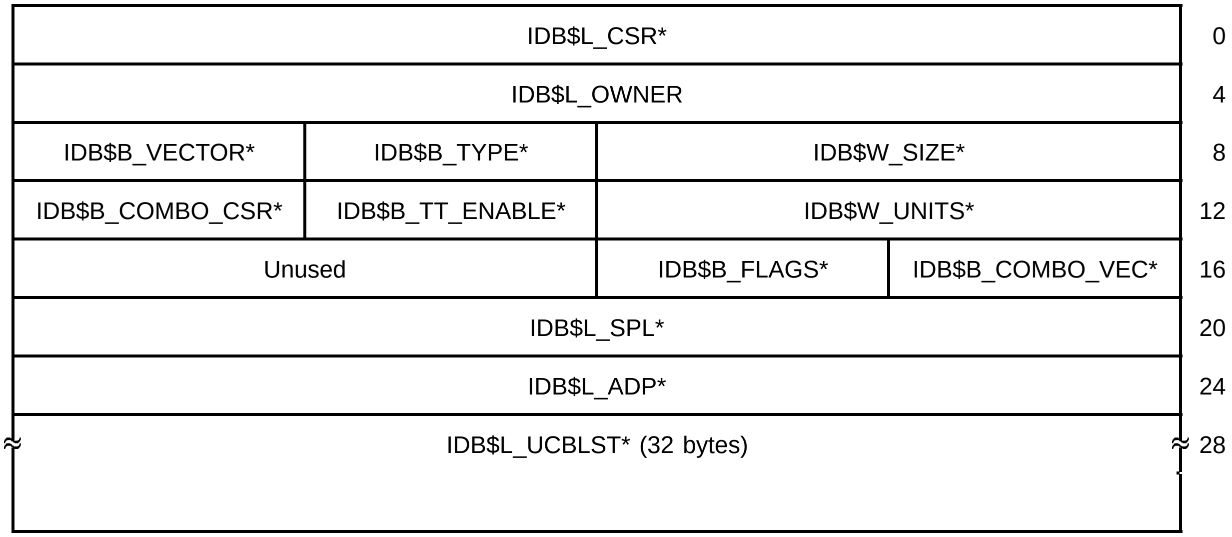
Field Name	Contents
	The driver-loading procedure compares the name of a driver to be loaded with the values in this field in all DPTs already loaded into system memory to ensure that it loads only one copy of a driver at a time.
DPT$Q_LINKTIME*	Time and date at which driver was linked, taken from its image header.
DPT$L_ECOLEVEL*	ECO level of driver, taken from its image header.
DPT$L_UCODE*	Address of associated microcode image, if DPT$V_UCODE is set in DPT$L_FLAGS. Use of this field is reserved to Digital.
DPT$Q_LMF_1*	First of eight quadwords reserved to Digital for the use of the license management facility. (The others are DPTQ_LMF_2, DPTQ_LMF_3, DPTQ_LMF_4, DPTQ_LMF_5, DPTQ_LMF_6, DPTQ_LMF_7, and DPT$Q_LMF_8.)
DPT$W_DECW_SNAME*	Offset to counted ASCII string used by decoding drivers.

1.11 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 1–14 and described in Table 1–13.

Figure 1–14 Interrupt Dispatch Block (IDB)



*A read-only field

Table 1–13 Contents of Interrupt Dispatch Block

Field Name	Contents
IDB$L_CSR*	Address of CSR. The SYSGEN command CONNECT specifies the address of a device's CSR. The driver-loading procedure writes the system virtual equivalent of this address into the IDB$L_CSR field. Device drivers set and clear bits in device registers by referencing all device registers at fixed offsets from the CSR address. If the controller resides on a remote bus connected to a VAX 7000-series or VAX 10000-series system, this field contains the pseudo CSR address (PCA) of the base register. The PCA uniquely describes a specific register of a specific node on a specific bus. The driver-loading procedure tests the value of this field. If the value is not a CSR address or a PCA, it sets IDB$V_NO_CSR in IDB$B_FLAGS and places the device offline by clearing UCB$V_ONLINE in UCB$L_STS. In this event, it does not call the driver's controller and unit initialization routines.
IDB$L_OWNER	Address of UCB of device that owns controller data channel. IOC$REQ *x* CHAN*y* writes a UCB address into this field when the routine allocates a controller data channel to a driver. IOC$REL*x* CHAN confirms that the proper driver fork process is releasing a channel by comparing the driver's UCB with the UCB stored in the IDB$L_OWNER field. If the UCB addresses are the same, IOC$REL*x* CHAN allocates the channel to a waiting driver by writing a new UCB address into the field. If no driver fork processes are waiting for the channel, IOC$REL*x*CHAN clears the field. If the controller is a single-unit controller, the unit or controller initialization routine should write the UCB address of the single device into this field.
IDB$W_SIZE*	Size of IDB. The driver-loading procedure writes the constant IDB$K_LENGTH into this field when the procedure creates the IDB.
IDB$B_TYPE*	Type of data structure. The driver-loading procedure writes the symbolic constant DYN$C_IDB into this field when the procedure creates the IDB.
IDB$B_VECTOR*	Interrupt vector number of the device, right-shifted by two bits. SYSGEN writes a value into this field using either the autoconfiguration database or the value specified in the /VECTOR qualifier to the CONNECT command. Drivers for devices that define the interrupt vector address through a device register must use this field to load that register during unit initialization and reinitialization after a power failure.
IDB$W_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$B_TT_ENABLE*	Reserved for use by the terminal driver.
IDB$B_COMBO_CSR*	Address of the start of CSRs for a multicontroller device such as the DMF32. (The name of this field is IDB$B_COMBO_CSR_OFFSET.)
IDB$B_COMBO_VEC*	Address of the start of interrupt vectors for a multicontroller device. (The name of this field is IDB$B_COMBO_VECTOR_OFFSET.)
IDB$B_FLAGS*	Flags associated with the IDB. The only flag currently defined is IDB$V_NO_CSR. The driver loading procedure sets this flag if IDB$L_CSR does not contain the address of a CSR.
IDB$L_SPL*	Address of the device lock that—in a multiprocessing environment—synchronizes access to device registers and those fields in the UCB accessed at device IPL.

(continued on next page)

Table 1–13 (Cont.) Contents of Interrupt Dispatch Block

Field Name	Contents
IDB$L_ADP*	Address of the adapter's ADP. The SYSGEN CONNECT command must specify the nexus number of the I/O adapter used by a device. The driver-loading procedure writes the address of the ADP for the specified I/O adapter into the IDB$L_ADP field.
IDB$L_UCBLST*	List of UCB addresses. The size of this field is the maximum number of units supported by the controller, as defined in the DPT. The maximum specified in the DPT can be overridden at driver load time. The driver-loading procedure writes a UCB address into this field every time the routine creates a new UCB associated with the controller.

1.12 I/O Request Packet (IRP)

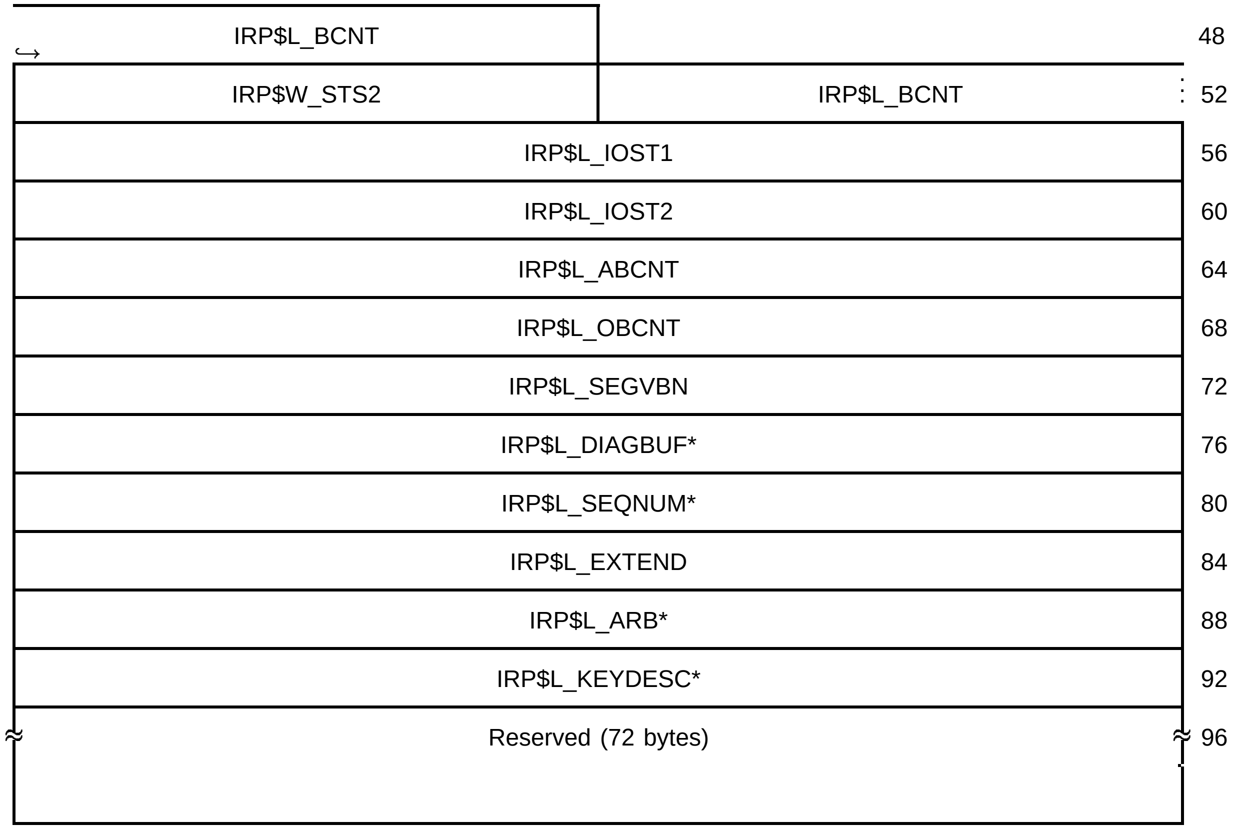
When a user process queues a valid I/O request by issuing a $QIO or $QIOW system service, the service creates an I/O request packet (IRP). The IRP contains a description of the request and receives the status of the I/O processing as it proceeds.

The I/O request packet is illustrated in Figure 1–15 and described in Table 1–14. Note that the standard IRP contains space for fields required by multiprocessing and the class drivers. Under no circumstances should a driver not supplied by Digital use these fields.

Figure 1–15 I/O Request Packet (IRP)

<table>
<tr><td colspan="4">IRP$L_IOQFL</td><td>0</td></tr>
<tr><td colspan="4">IRP$L_IOQBL</td><td>4</td></tr>
<tr><td>IRP$B_RMOD*</td><td>IRP$B_TYPE*</td><td colspan="2">IRP$W_SIZE*</td><td>8</td></tr>
<tr><td colspan="4">IRP$L_PID*</td><td>12</td></tr>
<tr><td colspan="4">IRP$L_AST*</td><td>16</td></tr>
<tr><td colspan="4">IRP$L_ASTPRM*</td><td>20</td></tr>
<tr><td colspan="4">IRP$L_WIND*</td><td>24</td></tr>
<tr><td colspan="4">IRP$L_UCB*</td><td>28</td></tr>
<tr><td>IRP$B_PRI*</td><td>IRP$B_EFN*</td><td colspan="2">IRP$W_FUNC</td><td>32</td></tr>
<tr><td colspan="4">IRP$L_IOSB*</td><td>36</td></tr>
<tr><td colspan="2">IRP$W_STS</td><td colspan="2">IRP$W_CHAN*</td><td>40</td></tr>
<tr><td colspan="4">IRP$L_SVAPTE</td><td>44</td></tr>
<tr><td colspan="2"></td><td colspan="2">IRP$W_BOFF</td><td></td></tr>
</table>

(continued on next page)



*A read-only field

Table 1–14 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. Note that the MSB is set for shared memory support, otherwise it must be zero.
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.

(continued on next page)

Table 1–14 (Cont.) Contents of an I/O Request Packet

Field Name	Contents
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$QIO calls FDT action routines to preprocess an I/O request. Six bits of the function code describe the basic function. The remaining 10 bits modify the function.
IRP$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.

(continued on next page)

Table 1–14 (Cont.) Contents of an I/O Request Packet

Field Name	Contents	
IRP$W_STS	Status of I/O request. EXE$QIO initializes this field to 0. EXE$QIO, FDT routines, and driver fork processes modify this field according to the current status of the I/O request. I/O postprocessing reads this field to determine what sort of postprocessing is necessary (for example, deallocate system buffers and adjust quota usage). Bits in the IRP$W_STS field describe the type of I/O function, as follows:	
	IRP$V_BUFIO	Buffered-I/O function
	IRP$V_FUNC	Read function
	IRP$V_PAGIO	Paging-I/O function
	IRP$V_COMPLX	Complex-buffered-I/O function
	IRP$V_VIRTUAL	Virtual-I/O function
	IRP$V_CHAINED	Chained-buffered-I/O function
	IRP$V_SWAPIO	Swapping-I/O function
	IRP$V_DIAGBUF	Diagnostic buffer is present
	IRP$V_PHYSIO	Physical-I/O function
	IRP$V_TERMIO	Terminal I/O (for priority increment calculation)
	IRP$V_MBXIO	Mailbox-I/O function
	IRP$V_EXTEND	An extended IRP is linked to this IRP
	IRP$V_FILACP	File ACP I/O
	IRP$V_MVIRP	Mount-verification I/O function
	IRP$V_SRVIO	Server-type I/O
	IRP$V_KEY	Encrypted function (encryption key address at IRP$L_KEYDESC)
IRP$L_SVAPTE	For a direct-I/O transfer, virtual address of the first page-table entry (PTE) of the I/O-transfer buffer, written here by the FDT routine locking process pages; for buffered-I/O transfer, address of a buffer in system address space, written here by the FDT routine allocating buffer. IOC$INITIATE copies this field into UCB$L_SVAPTE before transferring control to a device driver start-I/O routine. I/O postprocessing uses this field to deallocate the system buffer for a buffered-I/O transfer or to unlock pages locked for a direct-I/O transfer.	
IRP$W_BOFF	Byte offset into the first page of a direct-I/O transfer. FDT routines calculate this offset and write the field. For buffered-I/O transfers, FDT routines must write the number of bytes to be charged to the process in this field because these bytes are being used for a system buffer. IOC$INITIATE copies this field into UCB$W_BOFF before calling a device driver start-I/O routine. I/O postprocessing uses IRP$W_BOFF in conjunction with IRP$L_BCNT and IRP$L_SVAPTE to unlock pages locked for direct I/O. For buffered I/O, I/O postprocessing adds the value of IRP$W_BOFF to the process byte count quota.	

(continued on next page)

Table 1–14 (Cont.) Contents of an I/O Request Packet

<table>
<tr><th>Field Name</th><th>Contents</th></tr>
<tr><td>IRP$L_BCNT</td><td>Byte count of the I/O transfer. FDT routines calculate the count value and write the field. IOC$INITIATE copies the low-order word of this field into UCB$W_BCNT before calling a device driver's start-I/O routine.
For a buffered-I/O-read function, I/O postprocessing uses IRP$L_BCNT to determine how many bytes of data to write to the user's buffer.
The field IRP$W_BCNT points to the low-order word of this field to provide compatibility with previous versions of the operating system.</td></tr>
<tr><td>IRP$W_STS2</td><td>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.
Bits in the IRP$W_STS2 field describe the type of I/O function, as follows:
<table>
<tr><td>IRP$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>
<tr><td>IRP$V_SHDIO</td><td>Shadowing IRP.</td></tr>
<tr><td>IRP$V_CACHEIO</td><td>I/O using VBN cache buffers.</td></tr>
<tr><td>IRP$V_WLE</td><td>I/O uses a write log entry.</td></tr>
<tr><td>IRP$V_CACHE_SAFE</td><td>Request has been checked through cache.</td></tr>
<tr><td>IRP$V_NOCACHE</td><td>IO$M_NOVCACHE was set in the QIO function.</td></tr>
</table></td></tr>
<tr><td>IRP$L_IOST1</td><td>First I/O status longword. IOC$REQCOM and EXE$FINISHIO(C) write the contents of R0 into this field. The I/O postprocessing routine copies the contents of this field into the user's IOSB.
EXE$ZEROPARM copies a 0 and EXE$ONEPARM copies **p1** into this field. This field is a good place to put a $QIO request argument (**p1** through **p6**) or a computed value.
This field is also called IRP$L_MEDIA.</td></tr>
<tr><td>IRP$L_IOST2</td><td>Second I/O status longword. IOC$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.
The low byte of this field is also known as IRP$B_CARCON. IRP$B_CARCON contains carriage control instructions to the driver. EXE$READ and EXE$WRITE copy the contents of **p4** of the user's I/O request into this field.</td></tr>
<tr><td>IRP$L_ABCNT</td><td>Accumulated bytes transferred in virtual I/O transfer. IOC$IOPOST reads and writes this field after a partial virtual transfer.
The symbol IRP$W_ABCNT points to the low-order word of this field to provide compatibility with previous versions of the operating system.</td></tr>
<tr><td>IRP$L_OBCNT</td><td>Original transfer byte count in a virtual I/O transfer. IOC$IOPOST reads this field to determine whether a virtual transfer is complete, or whether another I/O request is necessary to transfer the remaining bytes.
The symbol IRP$W_OBCNT points to the low-order word of this field to provide compatibility with previous versions of the operating system.</td></tr>
</table>

(continued on next page)

Table 1–14 (Cont.) Contents of an I/O Request Packet

Field Name	Contents	
IRP$L_SEGVBN	Virtual block number of the current segment of a virtual I/O transfer. IOC$IOPOST writes this field after a partial virtual transfer.	
IRP$L_DIAGBUF*	Address of a diagnostic buffer in system address space. If the I/O request call specifies a diagnostic buffer and if a diagnostic buffer length is specified in the DDT, and if the process has diagnostic privilege, EXE$QIO copies the buffer address into this field. EXE$QIO allocates a diagnostic buffer in system address space to be filled by IOC$DIAGBUFILL during I/O processing. During I/O postprocessing, the special kernel-mode AST routine copies diagnostic data from the system buffer into the process diagnostic buffer.	
IRP$L_SEQNUM*	I/O transaction sequence number. If an error is logged for the request, this field contains the universal error log sequence number.	
IRP$L_EXTEND	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$W_STS is set if this extension address is used.	
IRP$L_ARB*	Address of access rights block (ARB). This block is located in the PCB and contains the process privilege mask and UIC, which are set up as follows:	
	ARB$Q_PRIV	Quadword containing process privilege mask
	SPARE$L	Unused longword
	ARB$L_UIC	Longword containing process UIC
IRP$L_KEYDESC	Address of encryption key.	

1.13 I/O Request Packet Extension (IRPE)

I/O request packet extensions (IRPEs) hold additional I/O request information for devices that require more context than the standard IRP can accommodate. IRP extensions are also used when more than one buffer (region) must be locked into memory for a direct-I/O operation, or when a transfer requires a buffer that is larger than 64K. An IRPE provides space for two buffer regions, each with a 32-bit byte count.

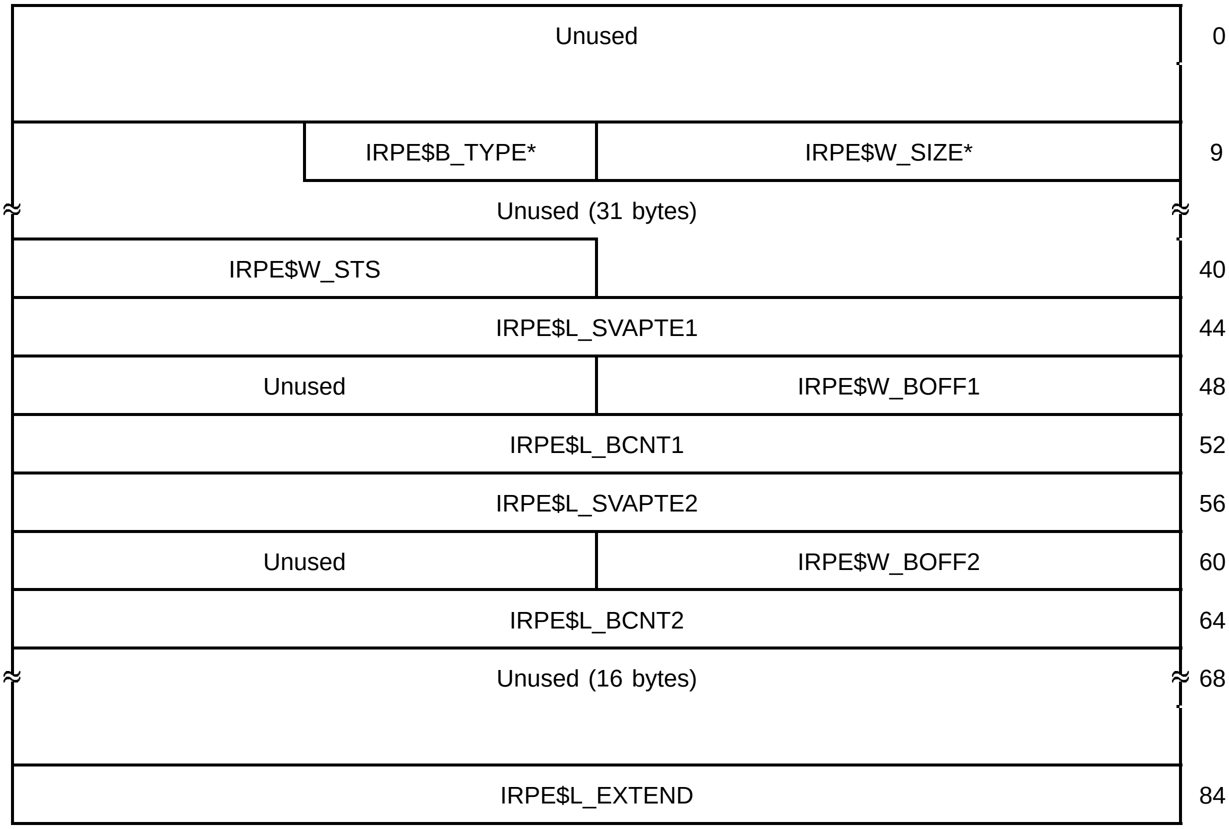
FDT routines allocate IRPEs by calling EXE$ALLOCIRP. Driver routines link the IRPE to the IRP, store the IRPE's address in IRP$L_EXTEND, and set the bit field IRP$V_EXTEND in IRP$W_STS to show that an IRPE exists for the IRP. The FDT routine initializes the contents of the IRPE. Any fields within the extension not described in Table 1–15 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 1–16 and described in Table 1–15.

Figure 1–16 I/O Request Packet Extension (IRPE)



*A read-only field

Table 1–15 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_EXTEND 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.

1.14 Object Rights Block (ORB)

The object rights block (ORB) is a data structure that describes the rights a process must have to access the object with which the ORB is associated. The ORB is not normally accessed by device drivers.

The ORB is usually allocated when the device is connected by means of the SYSGEN command CONNECT. SYSGEN also sets the address of the ORB in UCB$L_ORB at that time. The object name is normally stored at the end of the ORB (ORB$T_OBJECT_NAME).

The object rights block is illustrated in Figure 1–17 and described in Table 1–16.

Figure 1–17 Object Rights Block (ORB)

ORB$L_OWNER				0
ORB$L_ACL_MUTEX				4
↪ ORB$W_FLAGS	ORB$B_TYPE*	ORB$W_SIZE*		8
ORB$W_REFCOUNT		Unused	ORB$W_FLAGS :	12
ORB$Q_MODE_PROT				16
ORB$L_SYS_PROT				24
ORB$L_OWN_PROT				28
ORB$L_GRP_PROT				32
ORB$L_WOR_PROT				36
ORB$L_ACLFL				40
ORB$L_ACLBL				44
≈ ORB$R_MIN_CLASS (20 bytes) ≈				48
≈ ORB$R_MAX_CLASS (20 bytes) ≈				68
Unused		ORB$W_NAME_LENGTH		88
ORB$L_NAME_POINTER				92
ORB$L_OCB				96
ORB$L_TEMPLATE_ORB				100
ORB$L_OBJECT_SPECIFIC				104
ORB$L_ORIGINAL_ORB				108
Reserved		ORB$W_UPDSEQ		112
ORB$L_MUTEX_ADDRESS				116
Reserved				120

*A read-only field

Table 1–16 Contents of Object Rights Block

Field Name	Contents	
ORB$L_OWNER	UIC of the object’s owner.	
ORB$L_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 0000FFFF$_{16}$.	
ORB$W_SIZE*	Size in bytes of ORB. The driver-loading procedure writes the symbolic constant ORB$K_LENGTH into this field when it creates an ORB.	
ORB$B_TYPE*	Type of data structure. The driver-loading procedure writes the symbolic constant DYN$C_ORB into this field when it creates an ORB.	
ORB$W_FLAGS	Flags needed for interpreting portions of the ORB that can have alternate meanings. The following fields are defined within ORB$B_FLAGS:	
	ORB$V_PROT_16	When this flag is set, protection is stored as one word rather than four longwords.
	ORB$V_ACL_QUEUE	This flag represents the existence of an ACL queue.
	ORB$V_MODE_VECTOR	Use vector mode protection, not byte mode.
	ORB$V_NOACL	This object cannot have an ACL.
	ORB$V_CLASS_PROT	Security classification is valid.
	ORB$V_NOAUDIT	Disables $CHKPRO auditing.
	ORB$V_MODE_VALID	Access mode protection is valid.
	ORB$V_PROFILE_LOCKED	Object locked, no modification allowed. This flag is intended to be set when the profile cannot be modified. The protection of a volume set may only be altered if the root volume of the set is mounted, though mounting a selected volume from a volume set is supported.
	ORB$V_INDIRECT_ACL	Use the ACL from the template (ORB$L_TEMPLATE).
	ORB$V_BOOTTIME	ORB created prior to security object initialization.
	ORB$V_UNMODIFIED	ORB has not been explicitly modified.
	ORB$V_DAMAGED	Deny access to all but system (BADACL).
	ORB$V_TEMPLATE	This ORB is a template.
	ORB$V_TRANSITION	Profile content is uncertain.
ORB$W_REFCOUNT	Reference count.	
ORB$Q_MODE_PROT	Mode protection vector. The low byte of this quadword is known as ORB$B_MODE.	
ORB$L_SYS_PROT	System protection field. The low word of this field is known as ORB$W_PROT and may contain the standard SOGW protection.	
ORB$L_OWN_PROT	Owner protection field.	
ORB$L_GRP_PROT	Group protection field.	
ORB$L_WOR_PROT	World protection field.	
ORB$L_ACLFL	ACL queue forward link.	
ORB$L_ACLBL	ACL queue backward link.	

(continued on next page)

Table 1–16 (Cont.) Contents of Object Rights Block

Field Name	Contents
ORB$R_MIN_CLASS	Minimum classification mask.
ORB$R_MAX_CLASS	Maximum classification mask.
ORB$W_NAME_LENGTH	Length of object name.
ORB$L_NAME_POINTER	Pointer to object name.
ORB$L_OCB	Pointer to object class block.
ORB$L_TEMPLATE_ORB	Pointer to template ORB.
ORB$L_OBJECT_SPECIFIC	Object class specific usage cell.
ORB$L_ORIGINAL_ORB	Pointer to another ORB.
ORB$W_UPDSEQ	Update sequence number.
ORB$L_MUTEX_ADDRESS	Address of mutex for $CHKPRO.

1.15 SCSI Class Driver Request Packet (SCDRP)

The SCSI class driver allocates and builds a SCSI class driver request packet (SCDRP) for each I/O request it services, passing it to the SCSI port driver. The class driver routine initializes the SCDRP with the addresses of the UCB, SCDT, and IRP and copies to it data obtained from the IRP. The SCDRP also contains the addresses of the SCSI command buffer and status buffer.

The SCSI class driver passes the address of the SCDRP to the port driver in the call to SPI$SEND_COMMAND.

The SCDRP is illustrated in Figure 1–18 and described in Table 1–17.

Figure 1–18 SCSI Class Driver Request Packet (SCDRP)

<table>
<tr><td colspan="4">SCDRP$L_FQFL</td><td>0</td></tr>
<tr><td colspan="4">SCDRP$L_FQBL</td><td>4</td></tr>
<tr><td>SCDRP$B_FLCK</td><td>SCDRP$B_CD_TYPE</td><td colspan="2">SCDRP$W_SCDRPSIZE</td><td>8</td></tr>
<tr><td colspan="4">SCDRP$L_FPC</td><td>12</td></tr>
<tr><td colspan="4">SCDRP$L_FR3</td><td>16</td></tr>
<tr><td colspan="4">SCDRP$L_FR4</td><td>20</td></tr>
<tr><td colspan="4">SCDRP$L_PORT_UCB</td><td>24</td></tr>
<tr><td colspan="4">SCDRP$L_UCB</td><td>28</td></tr>
<tr><td colspan="2">SCDRP$W_STS</td><td colspan="2">SCDRP$W_FUNC</td><td>32</td></tr>
<tr><td colspan="4">SCDRP$L_SVAPTE</td><td>36</td></tr>
</table>

(continued on next page)

Reserved	SCDRP$W_BOFF	40
SCDRP$L_BCNT		44
SCDRP$L_MEDIA		48
SCDRP$L_ABCNT		52
SCDRP$L_SAVD_RTN		56
SCDRP$L_MSG_BUF		60
SCDRP$L_RSPID		64
SCDRP$L_CDT		68
SCDRP$L_RWCPTR		72
SCDRP$L_IRP		76
SCDRP$L_SVA_USER		80
SCDRP$L_CMD_BUF		84
SCDRP$L_CMD_BUF_LEN		88
SCDRP$L_CMD_PTR		92
SCDRP$L_STS_PTR		96
SCDRP$L_SCSI_FLAGS		100
SCDRP$L_DATACHECK		104
SCDRP$L_SCSI_STK_PTR		108
≈ SCDRP$L_SCSI_STK (32 bytes) ≈		112
SCDRP$L_CL_RETRY		144
SCDRP$L_DMA_TIMEOUT		148
SCDRP$L_DISCON_TIMEOUT		152
Reserved	SCDRP$W_PAD_BCNT	156
≈ SCDRP$B_TQE* (52 bytes) ≈		160
SCDRP$L_TQE_DELAY*		212
SCDRP$L_SVA_DMA*		216
SCDRP$L_SVA_CMD*		220

(continued on next page)

SCDRP$W_CMD_MAPREG*	SCDRP$W_MAPREG*	224
SCDRP$W_CMD_NUMREG*	SCDRP$W_NUMREG*	228
SCDRP$L_SVA_SPTE*		232
SCDRP$L_SCSIMSGO_PTR*		236
SCDRP$L_SCSIMSGI_PTR*		240
SCDRP$B_SCSIMSGO_BUF*		244
		248
SCDRP$B_SCSIMSGI_BUF*		
SCDRP$L_MSGO_PENDING*		256
SCDRP$L_MSGI_PENDING*		260
Reserved	SCDRP$B_LAST_MSGO*	264
SCDRP$L_DATA_PTR*		268
SCDRP$L_TRANS_CNT*		272
SCDRP$L_SAVE_DATA_CNT*		276
SCDRP$L_SAVE_DATA_PTR*		280
SCDRP$L_SDP_DATA_CNT*		284
SCDRP$L_SDP_DATA_PTR*		288
SCDRP$L_DUETIME*		292
SCDRP$L_TIMEOUT_ADDR*		296
SCDRP$W_BUSY_RETRY_CNT*	SCDRP$W_CMD_BCNT*	300
SCDRP$W_SEL_RETRY_CNT*	SCDRP$W_ARB_RETRY_CNT*	304
SCDRP$W_SEL_TQE_RETRY_CNT*	SCDRP$W_CMD_RETRY_CNT*	308
SCDRP$L_SAVER3*		312
SCDRP$L_SAVER6*		316
SCDRP$L_SAVER7*		320
SCDRP$L_SAVER3CL*		324
SCDRP$L_SAVEPCCL*		328
SCDRP$L_ABORTPCCL*		332
SCDRP$L_PO_STK_PTR*		336

(continued on next page)

SCDRP$L_PO_STK* (24 bytes)	340
SCDRP$L_TAG*	364
Reserved (44 bytes)	368

Note that the SCSI-2 data fields begin here.

SCDRP$W_QUEUE_CHAR*	SCDRP$W_QUEUE_TAG*	412
SCDRP$L_CLASS_STACK_PTR*		416
SCDRP$L_CLASS_STACK* (40 bytes)		420
SCDRP$L_PQFL*		460
SCDRP$L_PQBL*		464
SCDRP$L_BUS_PHASE*		468
SCDRP$L_OLD_PHASES*		472
SCDRP$L_EVENTS_SEEN*		476
SCDRP$L_CNX_STS*		480
SCDRP$L_SEQUENCE*		484
SCDRP$W_PHASES* (44 bytes)		488
SCDRP$L_PHASE_STK_PTR*		532
SCDRP$L_PHASE_END_STK_PTR*		536
SCDRP$L_SCDRP_SAV2*		540
SCDRP$L_ADDNL_INFO*		544
SCDRP$L_SENSE_KEY*		548

*A read-only field from the class driver point of view

Table 1–17 Contents of SCSI Class Driver Request Packet

Field Name	Contents
SCDRP$L_FQFL	Fork queue forward link. This field points to the next entry in the SCSI adapter's command buffer wait queue (ADP$L_BVPWAITFL), map register wait queue (ADP$L_MRQFL), port wait queue (SPDT$L_PORT_WQFL), or system fork queue.
SCDRP$L_FQBL	Fork queue backward link. This field points to the previous entry in the SCSI adapter's command buffer wait queue (ADP$L_BVPWAITFL), map register wait queue (ADP$L_MRQFL), port wait queue (SPDT$L_PORT_WQFL), or system fork queue.
SCDRP$W_SCDRPSIZE	Size of SCDRP. A SCSI class driver, after allocating sufficient nonpaged pool for the SCDRP, writes the constant SCDRP$C_LENGTH into this field.
SCDRP$B_CD_TYPE	Class driver type. This field is currently unused.
SCDRP$B_FLCK	Index of the fork lock that synchronizes access to this SCDRP at fork level. A SCSI class driver, after allocating sufficient nonpaged pool for the SCDRP, copies to this field the value of UCB$B_FLCK. All devices controlled by a single SCSI adapter and actively competing for shared adapter resources must specify the same value for this field.
SCDRP$L_FPC	Address of instruction at which processing resumes when SCSI adapter resources become available to satisfy a request stalled in an adapter resource wait queue.
SCDRP$L_FR3	Value of R3 when the request is stalled to wait for SCSI adapter resources. When the request is satisfied, this value is restored to R3 before the driver resumes execution at SCDRP$L_FPC.
SCDRP$L_FR4	Value of R4 when the request is stalled to wait for SCSI adapter resources. When the request is satisfied, this value is restored to R4 before the driver resumes execution art SCDRP$L_FPC.
SCDRP$L_PORT_UCB	SCSI adapter's UCB address. The SCSI port driver reads and writes this field in order to manage ownership of the SCSI port across bus reselection.
SCDRP$L_UCB	SCSI device's UCB address. The SCSI class driver initializes this field to indicate that the SCDRP is active.
SCDRP$W_FUNC	I/O function code that identifies the function to be performed for the I/O request. The SCSI class driver's start-I/O routine copies the contents of IRP$W_FUNC to this field.
SCDRP$W_STS	Status of I/O request. The SCSI class driver's start-I/O routine copies the contents of IRP$W_STS to this field. Bits in the SCDRP$W_STS field correspond to the bits in the IRP$W_STS field that 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 present

(continued on next page)

Table 1–17 (Cont.) Contents of SCSI Class Driver Request Packet

Field Name	Contents	
	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$L_KEYDESC)
SCDRP$L_SVAPTE	For a direct-I/O transfer, virtual address of the first page-table entry (PTE) of the I/O transfer buffer. This address is originally written to IRP$L_SVAPTE by the FDT routine that locks process pages. For a buffered-I/O transfer, address of a buffer in system address space. This address is originally written to IRP$L_SVAPTE by the class driver FDT routine that allocates the buffer. The class driver's start-I/O routine copies the address from the IRP to this field.	
SCDRP$W_BOFF	For a direct-I/O transfer, byte offset into the first page of the buffer; for a buffered-I/O transfer, number of bytes to be charged to the process requesting the transfer. FDT routines calculate this value and write it to IRP$W_BOFF. The class driver's start-I/O routine copies the value from the IRP to this field.	
SCDRP$L_BCNT	Byte count of the I/O transfer. Class driver FDT routines calculate this value and write it to IRP$L_BCNT. The class driver's start-I/O routine copies the value from the IRP to this field.	
SCDRP$L_MEDIA	Address of the media.	
SCDRP$L_ABCNT	Accumulated count of bytes transferred. The SCSI class driver maintains this field to accomplish segmented transfers.	
SCDRP$L_SAVD_RTN	Saved return address from Level 1 JSB.	
SCDRP$L_MSG_BUF	Address of allocated MSCP buffer.	
SCDRP$L_RSPID	Allocated request ID.	
SCDRP$L_CDT	Address of the SCSI connection descriptor table (SCDT). When the SCSI class driver's unit initialization routine invokes the SPI$CONNECT macro, the macro returns the address of the SCDT describing the connection it established to the SCSI port. The class driver stores that address in SCDRP$L_CDT.	
SCDRP$L_RWCPTR	RWAITCNT pointer.	
SCDRP$L_IRP	Address of I/O request block. The SCSI class driver copies the address of the IRP to this field.	
SCDRP$L_SVA_USER	System virtual address of a process buffer as mapped in system space (S0 space). The SCSI port driver initializes this field as the result of a class driver call to SPI$MAP_BUFFER.	
SCDRP$L_CMD_BUF	Address of the port command buffer. The SCSI class driver initializes this field with the address returned from a call to SPI$ALLOCATE_COMMAND_BUFFER.	

(continued on next page)

Table 1–17 (Cont.) Contents of SCSI Class Driver Request Packet

Field Name	Contents	
SCDRP$L_CMD_BUF_LEN	Length of SCSI command buffer.	
SCDRP$L_CMD_PTR	Address of the SCSI command descriptor block (its length byte) in the SCSI command buffer allocated by the SCSI port driver. The SCSI class driver initializes this field.	
SCDRP$L_STS_PTR	Address of SCSI status byte in the port command buffer. The SCSI class driver initializes this field.	
SCDRP$L_SCSI_FLAGS	SCSI flags. The SCSI class and port drivers use the following bits:	
	SCDRP$V_S0BUF	System buffer mapped. A SCSI class driver sets this bit, before invoking SPI$MAP_BUFFER, if the data transfer buffer is in system space (S0).
	SCDRP$V_BUFFER_MAPPED	Data transfer buffer mapped. A SCSI class driver sets this bit, after invoking SPI$MAP_BUFFER, to indicate that the data transfer buffer (either a system or process space buffer) has been mapped.
	SCDRP$V_DISK_SPUN_UP	START UNIT command issued. The SCSI disk class sets this bit.
	SCDRP$V_LOCK	Fork block in use.
	SCDRP$V_MREG_DONE	Mapping registers are loaded to control this transfer (set by the port driver).
	SCDRP$V_ONEBYTE	One byte transfer in progress.
	SCDRP$V_QUEUE_FULL	Indicates a full queue to port driver when the port is sending I/O to device.
	SCDRP$V_QUEUED_IO	Indicates a queued I/O characteristic when set. Indicates a not-queued I/O and error recovery when zero.
	SCDRP$V_ERROR_REC_IO	Indicates an error recovery I/O.
SCDRP$L_DATACHECK	Address of buffer for datacheck operations. A SCSI class driver maintains this field.	
SCDRP$L_SCSI_STK_PTR	Stack pointer of the class driver’s return address stack.	
SCDRP$L_SCSI_STK	Class driver’s return address stack. This stack is 32 bytes long.	
SCDRP$L_CL_RETRY	Retry count.	
SCDRP$L_DMA_TIMEOUT	Maximum number of seconds for a target to change the SCSI bus phase or complete a data transfer. Upon sending the last command byte, the port driver waits this many seconds for the target to change the bus phase lines and assert REQ (indicating a new phase). Or, if the target enters the DATA IN or DATA OUT phase, the transfer must be completed within this interval. A class driver can initialize this field to specify a per-request DMA timeout value.	
SCDRP$L_DISCON_TIMEOUT	Maximum number of seconds, from the time the initiator receives the DISCONNECT message, for a target to reselect the initiator so that it can proceed with the disconnected I/O transfer. A class driver can initialize this field to specify a per-request disconnect timeout value.	

(continued on next page)

Table 1–17 (Cont.) Contents of SCSI Class Driver Request Packet

Field Name	Contents
SCDRP$W_PAD_BCNT	Pad byte count. This field contains the number of bytes required to make the size of the user buffer equal to the data length value required by a specific SCSI command. A SCSI class driver uses this field to accommodate SCSI device classes that require that the transfer length be specified in terms of a larger data unit than the count of bytes expressed in the SCDRP$L_BCNT. If the total amount of data requested in the SCSI command does not match that specified in the SCDRP$L_BCNT, this field must account for the difference.
SCDRP$B_TQE*	Timer queue element, used by the port driver to time out pending disconnected I/O transfers. When this TQE expires, the timer thread times out expired pending I/O transfers.
SCDRP$L_TQE_DELAY*	Delay time for next TQE delay.
SCDRP$L_SVA_DMA*	System address of the section of the port DMA buffer allocated for the data transfer.
SCDRP$L_SVA_CMD*	System address of the segment of the port DMA buffer allocated for the port command buffer.
SCDRP$W_MAPREG*	Page number of the first port DMA buffer page allocated for the data transfer.
SCDRP$W_CMD_MAPREG*	Page number of the first port DMA buffer page allocated for the port command buffer.
SCDRP$W_NUMREG*	Number of port DMA buffer pages allocated for the data transfer.
SCDRP$W_CMD_NUMREG*	Number of port DMA buffer pages allocated for the port DMA buffer.
SCDRP$L_SVA_SPTE*	System virtual address of the system page-table entry that maps the first page of the process buffer in S0 space.
SCDRP$L_SCSIMSGO_PTR*	SCSI output message pointer.
SCDRP$L_SCSIMSGI_PTR*	SCSI input message pointer.
SCDRP$B_SCSIMSGO_BUF*	SCSI output message buffer.
SCDRP$B_SCSIMSGI_BUF*	SCSI input message buffer.

(continued on next page)

Table 1–17 (Cont.) Contents of SCSI Class Driver Request Packet

Field Name	Contents	
SCDRP$L_MSGO_PENDING*	Output message pending flags. One or more of the following bits are set in this longword if the port driver is to send the corresponding message:	
	SCDRP$V_IDENTIFY	IDENTIFY message
	SCDRP$V_SYNC_OUT	SYNCHRONOUS DATA TRANSFER REQUEST (out) message
	SCDRP$V_QUEUE_TAG	
	SCDRP$V_BUS_DEVICE_RESET	BUS DEVICE RESET message
	SCDRP$V_MESSAGE_PARITY_ERROR	MESSAGE PARITY ERROR message
	SCDRP$V_ID_ERROR	ID ERROR message
	SCDRP$V_ABORT	ABORT message
	SCDRP$V_NOP	NO OPERATION message
	SCDRP$V_MESSAGE_REJECT	MESSAGE REJECT message
SCDRP$L_MSGI_PENDING*	Input message pending flags. The only currently defined bit is SCDRP$V_SYNC_IN, which is set when the port driver expects to receive a SYCHRONOUS DATA TRANSFER REQUEST (in) message.	
SCDRP$B_LAST_MSGO*	Last message sent.	
SCDRP$L_DATA_PTR*	Current data pointer address.	
SCDRP$L_TRANS_CNT*	Actual number of bytes sent or received by the port driver. The port driver returns a value in this field to the class driver when it completes a SCSI data transfer.	
SCDRP$L_SAVE_DATA_CNT*	Running count of bytes (in two's-complement form) to be transferred. The port driver maintains this count.	
SCDRP$L_SAVE_DATA_PTR*	Pointer to current port DMA buffer segment. The SCSI port driver maintains this pointer.	
SCDRP$L_SDP_DATA_CNT*	Storage for SDP data count.	
SCDRP$L_SDP_DATA_PTR*	Storage for SDP data pointer.	
SCDRP$L_DUETIME*	Timeout time for a disconnected I/O transfer.	
SCDRP$L_TIMEOUT_ADDR*	Address of timeout routine.	
SCDRP$W_CMD_BCNT*	Command byte count.	
SCDRP$W_BUSY_RETRY_CNT*	Count of remaining busy retries.	
SCDRP$W_ARB_RETRY_CNT*	Count of remaining arbitration retries.	
SCDRP$W_SEL_RETRY_CNT*	Count of remaining selection retries.	
SCDRP$W_CMD_RETRY_CNT*	Count of remaining command retries.	
SCDRP$W_SEL_TQE_RETRY_CNT*	Count of remaining TQE retries.	

(continued on next page)

Table 1–17 (Cont.) Contents of SCSI Class Driver Request Packet

Field Name	Contents
SCDRP$L_SAVER3*	Reserved to Digital.
SCDRP$L_SAVER6*	Reserved to Digital.
SCDRP$L_SAVER7*	Reserved to Digital.
SCDRP$L_SAVER3CL*	Reserved to Digital.
SCDRP$L_SAVEPCCL*	Reserved to Digital.
SCDRP$L_ABORTPCCL*	Reserved to Digital.
SCDRP$L_PO_STK_PTR*	Stack pointer of the port driver's return address stack.
SCDRP$L_PO_STK*	Port driver's return address stack. This stack is 24 bytes long.
SCDRP$L_TAG*	Reserved to Digital.
SCDRP$W_QUEUE_TAG	SCSI-2 queue tag for this I/O allocated by the port driver. For more complex ports where the tag allocation is done by adapter firmware, this field is undefined.
SCDRP$W_QUEUE_CHAR	SCSI-2 queuing characteristic specified for this I/O. This field is filled in by the SPI$QUEUE_COMMAND as the class driver specifies one of the following values: 0 SCDRP$K_UNORDERED 1 SCDRP$K_HEAD 2 SCDRP$K_ORDERED 3 SCDRP$K_NOT_QUEUED 4 SCDRP$K_ERROR_RECOVERY
SCDRP$L_CLASS_STACK_PTR	Class driver return address stack pointer for SCSI-2 devices only.
SCDRP$L_CLASS_STACK	Class driver return address stack for SCSI-2 devices only (replaces UCB$L_STACK).
SCDRP$L_PQFL	Port (incoming and in-device) queue forward link used by INSQUE and REMQUE.
SCDRP$L_PQBL	Port (incoming and in-device) queue backward link.

(continued on next page)

Table 1–17 (Cont.) Contents of SCSI Class Driver Request Packet

Field Name	Contents	
SCDRP$L_BUS_PHASE*	Current SCSI bus phase. The SCSI port driver defines the following flags in this longword bit map:	
	SCDRP$V_DATAOUT	DATA OUT phase
	SCDRP$V_DATAIN	DATA IN phase
	SCDRP$V_CMD	COMMAND phase
	SCDRP$V_STS	STATUS phase
	SCDRP$V_INV1	Invalid phase 1
	SCDRP$V_INV2	Invalid phase 2
	SCDRP$V_MSGOUT	MESSAGE OUT phase
	SCDRP$V_MSGIN	MESSAGE IN phase
	SCDRP$V_ARB	ARBITRATION phase
	SCDRP$V_SEL	SELECTION phase
	SCDRP$V_RESEL	RESELECTION phase
	SCDRP$V_DISCON	DISCONNECT message seen
	SCDRP$V_CMD_CMPL	COMMAND COMPLETE message received
	SCDRP$V_TMODISCON	Disconnect operation timed out
	SCDRP$V_FREE	BUS FREE phase
SCDRP$L_OLD_PHASES*	Bus phase tracking information.	
SCDRP$L_EVENTS_SEEN*	Longword bit mask of bus events seen by the SCSI port driver. The following bits are defined:	
	SCDRP$V_PARERR	Parity error
	SCDRP$V_BSYERR	Bus lost during command
	SCDRP$V_MISPHS	Missing bus phase
	SCDRP$V_BADPHS	Bad phase transition
	SCDRP$V_RST	Bus reset during command
	SCDRP$V_CTLERR	SCSI controller error
	SCDRP$V_BUSERR	SCSI bus error
	SCDRP$V_ABORT	I/O has been aborted
	SCDRP$V_MSGERR	Error during message send

(continued on next page)

Table 1–17 (Cont.) Contents of SCSI Class Driver Request Packet

Field Name	Contents	
SCDRP$L_CNX_STS	Longword bit mask for the connection status. The following bits are defined:	
	SCDRP$V_ABORT_PND	Abort pending on connection
	SCDRP$V_ABORT_CMPL	Abort completed on connection
	SCDRP$V_ABORT_INPROG	Abort in progress
	SCDRP$V_ABORT_RESEL	Port was reselected while abort was in progress
	SCDRP$V_PND_RESEL	Reselection interrupt pending
	SCDRP$V_DSCN	Connection is disconnected
	SCDRP$V_TMODSCN	Connection timed out
SCDRP$L_SEQUENCE	Sequence number for this I/O.	
SCDRP$W_PHASES*	Bus phase tracking information. This field is 44 bytes long.	
SCDRP$L_PHASE_STK_PTR*	Address of the top of the bus phase stack. The SCSI port driver uses the bus phase stack to maintain a phase histogram.	
SCDRP$L_PHASE_END_STK_PTR*	Address of the bottom of the bus phase stack. The SCSI port driver uses the bus phase stack to maintain a phase histogram.	
SCDRP$L_SCDRP_SAV2	Saved address of original SCDRP request sense.	
SCDRP$L_ADDNL_INFO	Information bytes from sense data.	
SCDRP$L_SENSE_KEY	Request sense key from check condition.	

1.16 SCSI Connection Descriptor Table (SCDT)

The SCSI connection descriptor table (SCDT) contains information specific to a connection established between a SCSI class driver and the port, such as phase records, timeout values, and error counters. The SCSI port driver creates an SCDT each time a SCSI class driver, by invoking the SPI$CONNECT macro, connects to a device on the SCSI bus. The class driver stores the address of the SCDT in the SCSI device's UCB.

The SCSI port driver has exclusive access to the SCDT. A SCSI class driver has no access to this structure.

The SCDT is illustrated in Figure 1–19 and described in Table 1–18.

Figure 1–19 SCSI Connection Descriptor Table (SCDT)

SCDT$L_FLINK*				0
SCDT$B_SUBTYP*	SCDT$B_TYPE*	SCDT$W_SIZE*		4
SCDT$B_FLCK*	Reserved			8
SCDT$L_FPC*				12
SCDT$L_FR3*				16
SCDT$L_FR4*				20
SCDT$L_STS*				24
SCDT$W_STATE*		SCDT$W_SCDT_TYPE*		28
SCDT$L_SPDT*				32
SCDT$L_SCSI_PORT_ID*				36
SCDT$L_SCSI_BUS_ID*				40
SCDT$L_SCSI_LUN*				44
SCDT$L_AUXSTRUC				48
SCDT$L_SCDTLST				52
SCDT$L_SCDRP_ADDR*				56
SCDT$L_BUS_PHASE*				60
SCDT$L_OLD_PHASES*				64
SCDT$W_PHASES* (44 bytes)				68
SCDT$L_PHASE_STK_PTR*				112

(continued on next page)

SCDT$L_PHASE_END_STK_PTR*		116
SCDT$L_EVENTS_SEEN*		120
SCDT$L_ARB_FAIL_CNT*		124
SCDT$L_SEL_FAIL_CNT*		128
SCDT$L_PARERR_CNT*		132
SCDT$L_MISPHS_CNT*		136
SCDT$L_BADPHS_CNT*		140
SCDT$L_RETRY_CNT*		144
SCDT$L_RST_CNT*		148
SCDT$L_CTLERR_CNT*		152
SCDT$L_BUSERR_CNT*		156
SCDT$L_CMDSENT*		160
SCDT$L_MSGSENT*		164
SCDT$L_BYTSENT*		168
SCDT$L_CON_FLAGS*		172
SCDT$L_SYNCHRONOUS*		176
SCDT$W_TRANSFER_PERIOD*	SCDT$W_REQACK_OFFSET*	180
SCDT$W_ARB_RETRY_CNT*	SCDT$W_BUSY_RETRY_CNT*	184
SCDT$W_CMD_RETRY_CNT*	SCDT$W_SEL_RETRY_CNT*	188
SCDT$L_DMA_TIMEOUT*		192
SCDT$L_DISCON_TIMEOUT*		196
SCDT$L_SEL_CALLBACK*		200
SCDT$L_SEL_CONTEXT*		204
≈ Reserved (36 bytes) ≈		208
SCDT$L_PORT_QFL*		244
SCDT$L_PORT_QBL*		248
SCDT$L_DEV_QFL*		252
SCDT$L_DEV_QBL*		256

(continued on next page)

SCDT$L_QUEUE_FLAGS*		260
SCDT$Q_TAG_MAP*		264
SCDT$W_PORT_IO_COUNT*	SCDT$W_DEV_IO_COUNT*	272
SCDT$W_MAX_TAG_USED*	SCDT$W_WAIT_TAG*	276
Reserved	SCDT$W_MAX_QUEUE_DEPTH*	280
SCDT$L_SEQUENCE*		284
SCDT$L_NEXT_SEQUENCE*		288
SCDT$L_SCDRP_MAP*		292

*A read-only field from a class driver point of view

Table 1–18 Contents of SCSI Connection Descriptor Table

Field Name	Contents
SCDT$L_FLINK*	SCDT forward link. This field points to the next SCDT in the port's SCDT list (at SPDT$L_SCDT_VECTOR). The SCSI port driver initializes this field when it creates the SCDT in response to an SPI$CONNECT call.
SCDT$W_SIZE*	Size of SCDT. The port driver, after allocating sufficient nonpaged pool for the SCDT, writes the constant SCDT$C_LENGTH into this field.
SCDT$B_TYPE	SCS structure type.
SCDT$B_SUBTYP	SCSI structure subtype for CDT.
SCDT$B_FLCK*	Index of the fork lock that synchronizes access to this SCDT at fork level. The SCSI port driver, when creating the SCDT, initializes this field with SPL$C_IOLOCK8. The SCDT fork block is used during an ABORT command request on the connection.
SCDT$L_FPC*	Address of instruction at which the suspended port driver thread is to be resumed.
SCDT$L_FR3*	Value of R3 when the request is stalled during disconnection. The value in R3 is restored before a suspended driver thread is resumed.
SCDT$L_FR4*	Value of R4 when the request is stalled during disconnection. The value in R4 is restored before a suspended driver thread is resumed.

(continued on next page)

Table 1–18 (Cont.) Contents of SCSI Connection Descriptor Table

Field Name	Contents	
SCDT$L_STS*	Connection status. This field is a bit map, maintained by the port driver. The following bits are defined:	
	SCDT$V_BSY	Connection busy.
	SCDT$V_ABORT_PND	Abort pending on connection.
	SCDT$V_ABORT_CMPL	Abort completed on connection.
	SCDT$V_ABORT_INPROG	Abort is in progess.
	SCDT$V_ABORT_RESEL	Port was reselected while abort was in progress.
	SCDT$V_PND_RESEL	Reselection interrupt pending.
	SCDT$V_DSCN	Connection is disconnected.
	SCDT$V_TMODSCN	Connection timed out.
SCDT$W_SCDT_TYPE*	Type of SCDT.	
SCDT$W_STATE*	SCSI connection state. The SCSI port driver maintains this field, using the following constants:	
	SCDT$C_CLOSED	Closed
	SCDT$C_OPEN	Open
	SCDT$C_FAIL	Failed
SCDT$L_SPDT*	Address of port descriptor table with which this SCDT is associated.	
SCDT$L_SCSI_PORT_ID*	SCSI port ID of the port to which this connection is established.	
SCDT$L_SCSI_BUS_ID*	SCSI device ID of the device unit to which this connection is established.	
SCDT$L_SCSI_LUN*	SCSI logical unit number (LUN) of the device unit to which this connection is established.	
SCDT$L_AUXSTRUC	Address of auxiliary structure.	
SCDT$L_SCDTLST	Link for SCDT list from SPDT.	
SCDT$L_SCDRP_ADDR*	Address of SCDRP current on the connection.	

(continued on next page)

Table 1–18 (Cont.) Contents of SCSI Connection Descriptor Table

Field Name	Contents	
SCDT$L_BUS_PHASE*	Current SCSI bus phase. The SCSI port driver defines the following flags in this longword bit map:	
	SCDT$V_DATAOUT	DATA OUT phase
	SCDT$V_DATAIN	DATA IN phase
	SCDT$V_CMD	COMMAND phase
	SCDT$V_STS	STATUS phase
	SCDT$V_INV1	Invalid phase 1
	SCDT$V_INV2	Invalid phase 2
	SCDT$V_MSGOUT	MESSAGE OUT phase
	SCDT$V_MSGIN	MESSAGE IN phase
	SCDT$V_ARB	ARBITRATION phase
	SCDT$V_SEL	SELECTION phase
	SCDT$V_RESEL	RESELECTION phase
	SCDT$V_DISCON	DISCONNECT message seen
	SCDT$V_CMD_CMPL	COMMAND COMPLETE message received
	SCDT$V_TMODISCON	Disconnect operation timed out
	SCDT$V_FREE	BUS FREE phase
SCDT$L_OLD_PHASES*	Bus phase tracking information.	
SCDT$W_PHASES*	Bus phase tracking information. This field is 44 bytes long.	
SCDT$L_PHASE_STK_PTR*	Address of the top of the bus phase stack. The SCSI port driver uses the bus phase stack to maintain a phase histogram.	
SCDT$L_PHASE_END_STK_PTR*	Address of the bottom of the bus phase stack. The SCSI port driver uses the bus phase stack to maintain a phase histogram.	
SCDT$L_EVENTS_SEEN*	Longword bit mask of bus events seen by the SCSI port driver. The following bits are defined:	
	SCDT$V_PARERR	Parity error
	SCDT$V_BSYERR	Bus lost during command
	SCDT$V_MISPHS	Missing bus phase
	SCDT$V_BADPHS	Bad phase transition
	SCDT$V_RST	Bus reset during command
	SCDT$V_CTLERR	SCSI controller error
	SCDT$V_BUSERR	SCSI bus error
	SCDT$V_ABORT	I/O has been aborted
	SCDT$V_MSGERR	Error during message send
SCDT$L_ARB_FAIL_CNT*	Count of arbitration failures.	
SCDT$L_SEL_FAIL_CNT*	Count of selection failures.	
SCDT$L_PARERR_CNT*	Count of parity errors.	
SCDT$L_MISPHS_CNT*	Count of missing phases errors.	

(continued on next page)

Table 1–18 (Cont.) Contents of SCSI Connection Descriptor Table

<table>
<tr><th>Field Name</th><th>Contents</th></tr>
<tr><td>SCDT$L_BADPHS_CNT*</td><td>Count of bad phase errors.</td></tr>
<tr><td>SCDT$L_RETRY_CNT*</td><td>Count of retries.</td></tr>
<tr><td>SCDT$L_RST_CNT*</td><td>Count of bus resets.</td></tr>
<tr><td>SCDT$L_CTLERR_CNT*</td><td>Count of controller errors.</td></tr>
<tr><td>SCDT$L_BUSERR_CNT*</td><td>Count of bus errors.</td></tr>
<tr><td>SCDT$L_CMDSENT*</td><td>Number of commands sent on this connection.</td></tr>
<tr><td>SCDT$L_MSGSENT*</td><td>Number of messages sent on this connection.</td></tr>
<tr><td>SCDT$L_BYTSENT*</td><td>Number of bytes sent during DATA OUT phase.</td></tr>
<tr><td>SCDT$L_CON_FLAGS*</td><td>Connection-specific flags. The SCSI port driver sets or clears these flags according to information the SCSI class driver supplies to the SPI$SET_CONNECTION_CHAR macro. The following bits are defined:
<table>
<tr><td>SCDT$V_ENA_DISCON</td><td>Enable disconnect</td></tr>
<tr><td>SCDT$V_DIS_RETRY</td><td>Disable command retry</td></tr>
<tr><td>SCDT$V_TARGET_MODE</td><td>Enable asynchronous event notification from target</td></tr>
</table></td></tr>
<tr><td>SCDT$L_SYNCHRONOUS*</td><td>Synchronous data transfer enabled field. This longword contains 1 if synchronous data transfers are enabled for this connection; otherwise it contains a 0. The SCSI port driver writes this field according to information the SCSI class driver supplies to the SPI$SET_CONNECTION_CHAR macro.</td></tr>
<tr><td>SCDT$W_REQACK_OFFSET*</td><td>For synchronous data transfers, maximum number of REQs outstanding on the connection before an ACK is transmitted. The SCSI port driver writes this field according to information the SCSI class driver supplies to the SPI$SET_CONNECTION_CHAR macro.</td></tr>
<tr><td>SCDT$W_TRANSFER_PERIOD*</td><td>Number of 4-nanosecond ticks between a REQ and an ACK on this connection. The SCSI port driver writes this field according to information the SCSI class driver supplies to the SPI$SET_CONNECTION_CHAR macro.</td></tr>
<tr><td>SCDT$W_BUSY_RETRY_CNT*</td><td>Remaining number of retries allowed on this connection to successfully send a command to the target device. The SCSI port driver initially writes this field according to information the SCSI class driver supplies to the SPI$SET_CONNECTION_CHAR macro.</td></tr>
<tr><td>SCDT$W_ARB_RETRY_CNT*</td><td>Remaining number of retries allowed on this connection while waiting for the port to win arbitration of the bus. The SCSI port driver initially writes this field according to information the SCSI class driver supplies to the SPI$SET_CONNECTION_CHAR macro.</td></tr>
<tr><td>SCDT$W_SEL_RETRY_CNT*</td><td>Select retry count. Remaining number of retries allowed on this connection while waiting for the port to be selected by the target device. The SCSI port driver initially writes this field according to information the SCSI class driver supplies to the SPI$SET_CONNECTION_CHAR macro.</td></tr>
<tr><td>SCDT$W_CMD_RETRY_CNT*</td><td>Remaining number of retries allowed on this connection to successfully send a command to the target device. The SCSI port driver initially writes this field according to information the SCSI class driver supplies to the SPI$SET_CONNECTION_CHAR macro.</td></tr>
</table>

(continued on next page)

Table 1–18 (Cont.) Contents of SCSI Connection Descriptor Table

Field Name	Contents	
SCDT$L_DMA_TIMEOUT*	Timeout value (in seconds) for a target to change the SCSI bus phase or complete a data transfer. The SCSI port driver initially writes this field according to information the SCSI class driver supplies to the SPI$SET_CONNECTION_CHAR macro.	
SCDT$L_DISCON_TIMEOUT*	Disconnect timeout. Default timeout value (in seconds) for a target to reselect the initiator to proceed with a disconnected I/O transfer. The SCSI port driver initially writes this field according to information the SCSI class driver supplies to the SPI$SET_CONNECTION_CHAR macro.	
SCDT$L_SEL_CALLBACK*	Address of class driver's asynchronous event notification callback routine.	
SCDT$L_SEL_CONTEXT	Context for class driver callback.	
SCDT$L_PORT_QFL	Forward queue header link in managing the incoming port queue. As each I/O is sent to the device, it is removed from the port queue and placed on the in-device queue (SPDT$L_DEV_QBL).	
SCDT$L_PORT_QBL	Backward (tail) queue header link in managing the incoming port queue.	
SCDT$L_DEV_QFL	Forward queue header link in managing the in-device queue of I/O requests that were sent to the device.	
SCDT$L_DEV_QBL	Backward (tail) queue link in managing the in-device port queue.	
SCDT$L_QUEUE_FLAGS	Port queue flags for queue management are as follows:	
	SCDT$V_CMDQ	Indicates this connection supports command queuing.
	SCDT$V_FLUSHQ	Indicates this connection is to flush on error.
	SCDT$V_SCSI_2	Indicates the device conforms to SCSI-2.
	SCDT$V_QUEUE_WAIT	Indicates the port queue is currently waiting for a command to complete in the device. The I/O causing the wait is identified in the SCDT$W_WAIT_TAG field. When this I/O completes, the SCDT$V_QUEUE_WAIT bit is cleared unblocking the queue.
	SCDT$V_QUEUE_FROZEN	Indicates a queued command has terminated with a CHECK_CONDITION SCSI status. Used by the port driver to know when the port is waiting for the class driver to complete its error recovery. Cleared by the SPI$RELEASE_QUEUE call. Any I/O received while this bit is set is immediately returned to the class driver with failure status. This bit is also set by the SPI$FREEZE_QUEUE call.
	SCDT$V_FLUSHING_QUEUE	Indicates the port is flushing the device and port queues. I/O received while this bit is set is immediately returned to the class driver with failure status. Cleared by a count 0 in the SCDT$W_DEV_IO_COUNT and SCDT$W_PORT_IO_COUNT fields. This bit is also set by the SPI$FLUSH_QUEUE call.
	SCDT$V_QUEUE_FULL	Queue full status detected.

(continued on next page)

Table 1–18 (Cont.) Contents of SCSI Connection Descriptor Table

Field Name	Contents
SCDTW_TAG_MAP	Quadword bitmap of allocated tags. A bit set in this map indicates a tag value in use. Tag values in the map can be from 0 to 63 allowing 64 outstanding I/Os in a device.
SCDT$W_DEV_IO_COUNT	Number of I/Os currently outstanding on the in-device queue.
SCDT$W_PORT_IO_COUNT	Number of I/Os currently outstanding on the incoming port queue.
SCDT$W_WAIT_TAG	Synchronizes the port queue for a non-queued I/O request. A tag value is still allocated though the command is not sent as a tagged command to the device. The port will not initiate queued I/O if the SCDT$V_QUEUE_WAIT bit is set until the I/O in SCDT$W_WAIT_TAG has completed.
SCDT$W_MAX_TAG	Largest tag value used.
SCDT$W_MAX_QUEUE	Class driver imposed limit.
SCDT$L_SEQUENCE	Next sequence to be used.
SCDT$L_NEXT_SEQUENCE	Next sequence ID to be sent device.
SCDT$L_SCDRP_MAP	Pointer to a list of SCDRPs indexed by the tag value. Reduces searching by SCDT$L_DEV_QFL.

1.17 SCSI Port Descriptor Table (SPDT)

The SCSI port descriptor table (SPDT) contains information specific to a SCSI port, such as the port driver connection database. The SPDT also includes a set of vectors, corresponding to the SPI macros invoked by SCSI class drivers, that point to service routines within the port driver. The SCSI port driver's unit initialization routine creates an SPDT for each SCSI port defined for a specific MicroVAX or VAXstation system and initializes each SPI vector.

The port driver reads and writes fields in the SPDT. The class driver reads the SPDT indirectly when it invokes an SPI macro.

The SPDT is illustrated in Figure 1–20 and described in Table 1–19.

Figure 1–20 SCSI Port Descriptor Table (SPDT)

SPDT$L_FLINK*			0
SPDT$B_SUBTYP*	SPDT$B_TYPE*	SPDT$W_SIZE*	4
SPDT$B_FLCK*	SPDT$B_SCSI_INT_MSK*	SPDT$W_SPDT_TYPE*	8
SPDT$L_FPC*			12
SPDT$L_FR3*			16
SPDT$L_FR4*			20
SPDT$L_SCSI_PORT_ID*			24
SPDT$L_SCSI_BUS_ID*			28
SPDT$L_STS*			32
SPDT$L_PORT_WQFL*			36
SPDT$L_PORT_WQBL*			40
SPDT$L_MAXBYTECNT*			44
SPDT$L_WAITQFL			48
SPDT$L_WAITQBL			52
SPDT$L_PORT_UCB*			56
SPDT$L_PORT_CSR*			60
SPDT$L_PORT_IDB*			64
SPDT$L_DMA_BASE*			68
SPDT$L_SPTE_BASE*			72
SPDT$L_SPTE_SVAPTE*			76
SPDT$L_ADP*			80
≈ SPDT$L_PORT_RING* (64 bytes) ≈			84
SPDT$L_PORT_RING_PTR*			148
SPDT$L_OWNERSCDT*			152
≈ SPDT$L_SCDT_VECTOR* (256 bytes) ≈			156

(continued on next page)

SPDT$L_DLCK*		412
Reserved	SPDT$B_DIPL*	416
SPDT$L_AUXSTRUC		420
SPDT$L_SEL_SCDRP*		424
SPDT$L_ENB_SEL_SCDRP*		428
SPDT$L_MAP_BUFFER*		432
SPDT$L_UNMAP*		436
SPDT$L_SEND*		440
SPDT$L_SET_CONN_CHAR*		444
SPDT$L_GET_CONN_CHAR*		448
SPDT$L_RESET*		452
SPDT$L_CONNECT*		456
SPDT$L_DISCONNECT*		460
SPDT$L_ALLOC_COMMAND_BUFFER*		464
SPDT$L_DEALLOC_COMMAND_BUFFER*		468
SPDT$L_ABORT*		472
SPDT$L_SET_PHASE*		476
SPDT$L_SENSE_PHASE*		480
SPDT$L_SEND_BYTES*		484
SPDT$L_RECEIVE_BYTES*		488
SPDT$L_FINISH_CMD*		492
SPDT$L_RELEASE_BUS*		496
SPDT$L_QUEUE_CMD*		500
SPDT$L_FREEZE_QUEUE*		504
SPDT$L_RELEASE_QUEUE*		508
SPDT$L_FLUSH_QUEUE*		512
≈ Reserved (52 bytes) ≈		516
Reserved	BUS_HUNG_VEC*	568

(continued on next page)

SPDT$B_TQE* (52 bytes)				572
SPDT$L_TQE_DELAY*				624
SPDT$L_BUS_HUNG_CNT*				628
SPDT$L_TARRST_CNT*				632
SPDT$L_RETRY_CNT*				636
SPDT$L_STRAY_INT_CNT*				640
SPDT$L_UNEXP_INT_CNT*				644
SPDT$L_NODISCON_CNT*				648
SPDT$W_DISCON_CNT*		Reserved		652
SPDT$L_PORT_FLAGS*				656
SPDT$L_VERSION_CHECK*				660
Reserved (36 bytes)				664
SPDT$B_EVENT_CNT*	SPDT$B_MODE*	SPDT$B_STATUS*	SPDT$B_CUR_STAT*	700
SPDT$L_TVDRV_ISR*				704
SPDT$L_TVDRV_DMA_BASE*				708
SPDT$L_TVDRV_DMA_SIZE*				712
SPDT$L_TVDRV_UCB*				716
SPDT$L_CUR_SCDT_VEC*				720
SPDT$L_QMAN_RESUME*				724
SPDT$W_Reserved		SPDT$W_OWNER_TAG*		728
SPDT$L_PORT_IO_COUNT*				732
SPDT$L_QUEUE_SPINS*				736
SPDT$L_QUEUE_EXITS*				740

*A read-only field from a class driver point of view

Table 1–19 Contents of SCSI Port Descriptor Table

Field Name	Contents
SPDT$L_FLINK*	SPDT forward link. This field points to the next SPDT in the system SPDT list. The SCSI port driver initializes this field when it creates the SPDT.
SPDT$W_SIZE*	Size of SPDT. The SCSI port driver initializes this field to SPDT$C_PKNLENGTH or SPDT$C_PKSLENGTH when creating the SPDT.
SPDT$B_TYPE	Structure type.
SPDT$B_SUBTYP	Structure subtype.
SPDT$W_SPDT_TYPE*	SPDT type. The SCSI port driver initializes this field to SPDT$C_PKN or SPDT$C_PKS when creating the SPDT.
SPDT$B_SCSI_INT_MSK*	Port-specific interrupt mask.
SPDT$B_FLCK*	Index of the fork lock that synchronizes access to this SPDT at fork level. The SCSI port driver, when creating the SPDT, copies to this field the value of UCB$B_FLCK. The SPDT fork block is used during reselection and disconnection.
SPDT$L_FPC*	Address of instruction at which the suspended port driver thread is to be resumed.
SPDT$L_FR3*	Value of R3 when the request is stalled during disconnection. The value in R3 is restored before a suspended driver thread is resumed.
SPDT$L_FR4*	Value of R4 when the request is stalled during disconnection. The value in R4 is restored before a suspended driver thread is resumed.
SPDT$L_SCSI_PORT_ID*	SCSI port ID, an alphabetic value from A to Z.
SPDT$L_SCSI_BUS_ID*	SCSI device ID of the port, a numeric value from 0 to 7.
SPDT$L_STS*	Port device status. This field is a bit map maintained by the port driver. The following bits are defined: SPDT$V_ONLINE Online SPDT$V_TIMOUT Timed out SPDT$V_ERLOGIP Error log in progress SPDT$V_CANCEL Cancel I/O SPDT$V_POWER Power failed while unit busy SPDT$V_BSY Busy SPDT$V_FAILED Port failed operation or initialization SPDT$V_FIFOLCK FIFO buffer is use
SPDT$L_PORT_WQFL*	Port wait queue forward link. This field points to the first SCDRP waiting for the port to be free.
SPDT$L_PORT_WQBL*	Port wait queue backward link. This field points to the last SCDRP waiting for the port to be free.
SPDT$L_MAXBYTECNT*	Maximum byte count for a transfer using this port.
SPDT$L_WAITQFL SPDT$L_WAITQBL	List head for fork blocks waiting for nonpaged pool.
SPDT$L_PORT_UCB*	Address of port UCB.
SPDT$L_PORT_CSR*	Address of the port hardware's CSR.
SPDT$L_PORT_IDB*	Address of the port IDB.
SPDT$L_DMA_BASE*	Base address of the port's DMA buffer.

(continued on next page)

Table 1–19 (Cont.) Contents of SCSI Port Descriptor Table

Field Name	Contents
SPDT$L_SPTE_BASE*	System virtual address of the system page-table entry mapping the first page of the port's DMA buffer.
SPDT$L_SPTE_SVAPTE*	System virtual address of the system page-table entry that double-maps the data transfer buffer.
SPDT$L_ADP*	Address of the adapter control block managing port resources.
SPDT$L_PORT_RING*	64-byte field recording the PCs of port channel request and release transactions.
SPDT$L_PORT_RING_PTR*	Pointer to the current port channel ring buffer entry.
SPDT$L_OWNERSCDT*	Address of the SCDT of the connection that currently owns the port.
SPDT$L_SCDT_VECTOR*	256-byte vector, recording the SCDT addresses associated with connections active for a given SCSI device ID (0 through 7).
SPDT$L_DLCK*	Address of device lock that—in a multiprocessing environment—synchronizes access to device registers and those fields at the SPDT accessed at device IPL. The port driver initializes this field from UCB$L_DLCK when it creates the SPDT.
SPDT$B_DIPL*	Interrupt priority level (IPL) at which the device requests hardware interrupts. The port driver initializes this field from UCB$L_DLCK when it creates the SPDT.
SPDT$L_AUXSTRUC	Address of auxiliary structure.
SPDT$L_SEL_SCDRP*	SCDRP used during selection interrupt.
SPDT$L_ENB_SEL_SCDRP*	SCDRP used to enable selection.
SPDT$L_MAP_BUFFER*	Address of the port driver routine that executes in response to a class driver's SPI$MAP_BUFFER macro call. The port driver initializes this field.
SPDT$L_UNMAP*	Address of the port driver routine that executes in response to a class driver's SPI$UNMAP_BUFFER macro call. The port driver initializes this field.
SPDT$L_SEND*	Address of the port driver routine that executes in response to a class driver's SPI$SEND_COMMAND macro call. The port driver initializes this field.
SPDT$L_SET_CONN_CHAR*	Address of the port driver routine that executes in response to a class driver's SPI$SET_CONNECTION_CHAR macro call. The port driver initializes this field.
SPDT$L_GET_CONN_CHAR*	Address of the port driver routine that executes in response to a class driver's SPI$GET_CONNECTION_CHAR macro call. The port driver initializes this field.
SPDT$L_RESET*	Address of the port driver routine that executes in response to a class driver's SPI$RESET macro call. The port driver initializes this field.
SPDT$L_CONNECT*	Address of the port driver routine that executes in response to a class driver's SPI$CONNECT macro call. The port driver initializes this field.
SPDT$L_DISCONNECT*	Address of the port driver routine that executes in response to a class driver's SPI$DISCONNECT macro call. The port driver initializes this field.
SPDT$L_ALLOC_COMMAND_BUFFER*	Address of the port driver routine that executes in response to a class driver's SPI$ALLOCATE_COMMAND_BUFFER macro call. The port driver initializes this field.

(continued on next page)

Table 1–19 (Cont.) Contents of SCSI Port Descriptor Table

Field Name	Contents
SPDT$L_DEALLOC_ COMMAND_BUFFER*	Address of the port driver routine that executes in response to a class driver's SPI$DEALLOCATE_COMMAND_BUFFER macro call. The port driver initializes this field.
SPDT$L_ABORT*	Address of the port driver routine that executes in response to a class driver's SPI$ABORT_COMMAND macro call. The port driver initializes this field.
SPDT$L_SET_PHASE*	Address of the port driver asynchronous event notification (AEN) routine that executes in response to a class driver's SPI$SET_PHASE macro call. The port driver initializes this field.
SPDT$L_SENSE_PHASE*	Address of the port driver AEN routine that executes in response to a class driver's SPI$SENSE_PHASE macro call. The port driver initializes this field.
SPDT$L_SEND_BYTES*	Address of the port driver AEN routine that executes in response to a class driver's SPI$SEND_BYTES macro call. The port driver initializes this field.
SPDT$L_RECEIVE_BYTES*	Address of the port driver AEN routine that executes in response to a class driver's SPI$RECEIVE_BYTES macro call. The port driver initializes this field.
SPDT$L_FINISH_CMD*	Address of the port driver AEN routine that executes in response to a class driver's SPI$FINISH_COMMAND macro call. The port driver initializes this field.
SPDT$L_RELEASE_BUS*	Address of the port driver routine that executes in response to a class driver's SPI$RELEASE_BUS macro call. The port driver initializes this field.
SPDT$L_QUEUE_CMD	Address of the port driver routine that executes in response to a class driver's SPI$QUEUE_COMMAND call. The port driver initializes this field.
SPDT$L_FREEZE_QUEUE	Address of the port driver routine that executes in response to a class driver's SPI$FREEZE_QUEUE call. The port driver initializes this field.
SPDT$L_RELEASE_QUEUE	Address of the port driver routine that executes in response to a class driver's SPI$RELEASE_QUEUE call. The port driver initializes this field.
SPDT$L_FLUSH_QUEUE	Address of the port driver routine that executes in response to a class driver's SPI$FLUSH_QUEUE call. The port driver initializes this field.
SPDT$B_BUS_HUNG_VEC*	Vector of suspected hung connections.
SPDT$B_TQE*	Timer queue element (52 bytes long), used by the port driver to time out pending disconnected I/O transfers. When this TQE expires, the timer thread times out expired pending I/O transfers.
SPDT$L_TQE_DELAY*	Delay time for next TQE delay.
SPDT$L_BUS_HUNG_CNT*	Count of detected bus hangs.
SPDT$L_TARRST_CNT*	Count of target-initiated bus resets.
SPDT$L_RETRY_CNT*	Total of retry attempts.
SPDT$L_STRAY_INT_CNT*	Count of interrupts occurring when channel is unowned.
SPDT$L_UNEXP_INT_CNT*	Count of unexpected interrupts occurring when channel is owned.
SPDT$L_NODISCON_CNT*	Count of reselections when port is not disconnected.
SPDT$W_DISCON_CNT*	Count of outstanding disconnects.

(continued on next page)

Table 1–19 (Cont.) Contents of SCSI Port Descriptor Table

Field Name	Contents	
SPDT$L_PORT_FLAGS*	Port-specific flags. The following bits are defined:	
	SPDT$V_SYNCH	Port supports synchronous mode data transfers.
	SPDT$V_ASYNCH	Port supports asynchronous mode data transfers.
	SPDT$V_MAPPING_REG	Port supports map registers.
	SPDT$V_BUF_DMA	Port supports buffered DMA transfers.
	SPDT$V_DIR_DMA	Port supports direct DMA transfers.
	SPDT$V_AEN	Port supports asynchronous event notification.
	SPDT$V_LUNS	Port supports logical unit numbers.
	SPDT$V_CMDQ	Port supports command queuing I/O.
	Bits <31:25>	Contain the recommended byte count divisor for the class driver to derive a proper DMA byte count for the port.
SPDT$L_VERSION_CHECK*	Value used to check driver versions.	
SPDT$B_CUR_STAT*	Copy of CUR_STAT register.	
SPDT$B_STATUS*	Copy of STATUS register.	
SPDT$B_MODE*	Copy of MODE register.	
SPDT$B_EVENT_CNT*	Count of events while servicing current interrupt.	
SPDT$L_TVDRV_ISR	Address of the TVDriver's ISR.	
SPDT$L_TVDRV_DMA_BASE	Address of the TVDriver's DMA buffer.	
SPDT$L_TVDRV_DMA_SIZE	Size of the TVDriver's DMA buffer.	
SPDT$L_TVDRV_UCB	Address of the TVDriver's UCB.	
SPDT$L_CUR_SCDT_VEC	Pointer into SCDT_VECTOR for CMDQ.	
SPDT$L_QMAN_RESUME	Address to resume QUEUE_MANAGER.	
SPDT$L_OWNER_TAG	Tag value of bus owner thread.	
SPDT$L_PORT_IO_COUNT	I/O number in port using SEND/QUEUE_CMD.	
SPDT$L_QUEUE_SPINS	Number of passes over port queues.	
SPDT$L_QUEUE_EXITS	Number of exits from the queue manager.	

1.18 Spinlock Data Structure (SPL)

The spinlock data structure (SPL) records all information necessary to properly grant, release, and record the ownership of a spinlock. Each static system spinlock (including the fork locks) and device lock uses an SPL to record the IPL required for spinlock acquisition, its rank, and its owner. The spinlock structure also maintains a history of spinlock use and a variety of counters used in accounting and debugging.

Static system spinlocks 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 spinlock data structure are illustrated in Figure 1–21 and described in Table 1–20.

Figure 1–21 Spinlock Data Structure (SPL)

SPL$B_VEC_INX*	SPL$B_RANK*	SPL$B_IPL*	SPL$B_SPINLOCK*	0
SPL$W_WAIT_CPUS*		SPL$W_OWN_CNT*		4
SPL$B_SUBTYPE*	SPL$B_TYPE*	SPL$W_SIZE*		8
SPL$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

*A read-only field

Table 1–20 Contents of the Spinlock Data Structure

Field	Contents
SPL$B_SPINLOCK*	The following fields are defined within SPL$B_SPINLOCK: SPL$V_INTERLOCK — Spinlock access interlock. When set, this bit signifies that the spinlock is owned. <7:1> — Reserved to Digital.
SPL$B_IPL*	IPL required for spinlock acquisition.
SPL$B_RANK*	Spinlock rank. Note that the internal value of a spinlock's rank, as stored in this field, is the inverse of the spin lock's logical rank, as displayed by the System Dump Analyzer. For instance, the spinlock structure with a logical rank of 0 contains the value 31 in this field.
SPL$B_VEC_INX*	Index of the next entry to be written in the spinlock PC vector index (SPL$L_OWN_PCVEC). SPL$B_VEC_INX is updated upon each successful acquisition or release of the spinlock.
SPL$W_OWN_CNT*	Ownership count. This field is –1 if the spinlock is unowned, zero or positive if owned. When a processor initially acquires a spinlock, this field goes from –1 to zero. A positive ownership count signifies concurrent acquisitions by a single processor.
SPL$W_WAIT_CPUS*	Number of processors waiting to obtain the spinlock.
SPL$W_SIZE*	Size of spinlock data structure (SPL$C_LENGTH).
SPL$B_TYPE*	Type of data structure. The operating system writes the value DYN$C_SPL in this field when it creates the SPL data structure.
SPL$B_SUBTYPE*	Spinlock subtype. This field can contain the following values: SPL$C_SPL_SPINLOCK — Static system spinlock SPL$C_SPL_FORKLOCK — Fork lock SPL$C_SPL_DEVICELOCK — Device lock (dynamic spinlock)
SPL$L_OWN_CPU*	Address of the per-CPU data structure of the processor that has acquired the spinlock. The field is cleared when a processor releases its last nested acquisition of the lock.
SPL$L_OWN_PC_VEC*	Last eight calling PCs of acquirers and releasers of the spinlock. 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 spinlock acquisition attempt fails.
SPL$L_RLS_PC*	PC of the last unconditional release of a set of nested acquisitions of the spinlock.

1.19 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 System Generation utility (SYSGEN) command CONNECT. The procedure creates UCBs of the length specified in the DPT. The driver uses UCB storage located beyond the standard UCB fields for device-specific data and temporary driver storage.

UCBs are variable in length depending on the type of device and whether the driver performs error logging for the device. The operating system defines a number of UCB extensions in the data structure definition macro $UCBDEF and defines a terminal device extension in $TTYUCBDEF. Table 1–21 lists those extensions that are most often used by device drivers, indicating where each extension is described in this chapter. Note that use of the dual-path extension is reserved to Digital; its contents should remain zero.

Table 1–21 UCB Extensions and Sizes Defined in $UCBDEF

Extension	Used by	Size	Figure	Table
Base UCB	All devices	UCB$K_SIZE	1–23	1–22
Error log extension	All disk and tape devices	UCB$K_ERL_LENGTH	1–24	1–23
Dual-path extension	Reserved to Digital	UCB$K_DP_LENGTH (UCB$K_2P_LENGTH)	—	—
Local tape extension	All tape devices	UCB$K_LCL_TAPE_LENGTH	1–25	1–24
Local disk extension	All disk devices	UCB$K_LCL_DISK_LENGTH	1–26	1–25
Terminal extension[1]	Terminal class and port drivers	UCB$K_TT_LENGTH	1–27[2]	1–26

[1] The terminal UCB extension is defined by the data structure definition macro, $TTYUCBDEF.

[2] Fields marked by asterisks may be written only by the terminal class driver (TTDRIVER.EXE); a port driver may only read these fields.

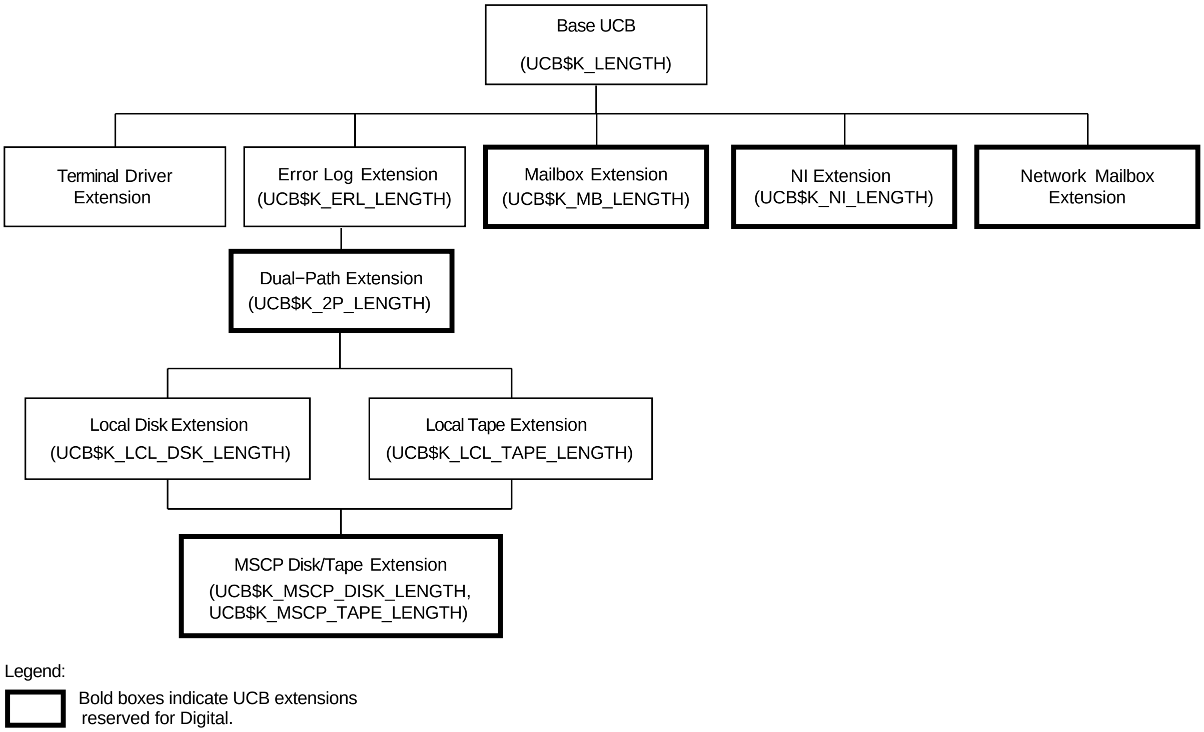
To use an extended UCB, a device driver must specify its length in the **ucbsize** argument to the DPTAB macro. For instance:

```
DPTAB   -,
   .
   .
   .
        UCBSIZE=UCB$K_LCL_TAPE_LENGTH,-
   .
   .
   .
```

As illustrated in Figure 1–22, each UCB extension used in a disk or tape driver builds upon the base UCB structure and any extension $UCBDEF defines earlier in the structure. (Note that UCB extensions shown in bold boxes are reserved to Digital.) For instance, if you specify a UCB size of UCB$K_LCL_TAPE_LENGTH, the size of the resulting UCB can accommodate the base UCB, the error-log extension, the dual-path extension, and the local tape extension.

The driver-loading procedure initializes some static UCB fields when it creates the block. The operating system and the 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 1–23 and described in Table 1–22. The length of the basic UCB is defined by the symbol UCB$K_LENGTH.

Figure 1–22 Composition of Extended Unit Control Blocks



ZK−6620−GE

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,-
        .
        .
        .
```

Figure 1–23 Unit Control Block (UCB)

UCB$L_FQFL*				0
UCB$L_FQBL*				4
UCB$B_FLCK	UCB$B_TYPE*	UCB$W_SIZE*		8
UCB$L_FPC				12
UCB$L_FR3				16
UCB$L_FR4				20
UCB$W_INIQUO*		UCB$W_BUFQUO*		24
UCB$L_ORB*				28
UCB$L_LOCKID*				32
UCB$L_CRB*				36
UCB$L_DLCK*				40
UCB$L_DDB*				44
UCB$L_PID*				48
UCB$L_LINK*				52
UCB$L_VCB*				56
UCB$L_DEVCHAR				60
UCB$L_DEVCHAR2				64
UCB$L_AFFINITY*				68
UCB$L_XTRA				72
UCB$W_DEVBUFSIZ		UCB$B_DEVTYPE	UCB$B_DEVCLASS	76

(continued on next page)

<table>
<tr><td colspan="4">UCB$Q_DEVDEPEND</td><td>80</td></tr>
<tr><td colspan="4">UCB$Q_DEVDEPEND2</td><td>88</td></tr>
<tr><td colspan="4">UCB$L_IOQFL*</td><td>96</td></tr>
<tr><td colspan="4">UCB$L_IOQBL*</td><td>100</td></tr>
<tr><td colspan="2">UCB$W_CHARGE*</td><td colspan="2">UCB$W_UNIT*</td><td>104</td></tr>
<tr><td colspan="4">UCB$L_IRP</td><td>108</td></tr>
<tr><td>UCB$B_AMOD*</td><td>UCB$B_DIPL</td><td colspan="2">UCB$W_REFC*</td><td>112</td></tr>
<tr><td colspan="4">UCB$L_AMB*</td><td>116</td></tr>
<tr><td colspan="4">UCB$L_STS</td><td>120</td></tr>
<tr><td colspan="2">UCB$W_QLEN*</td><td colspan="2">UCB$W_DEVSTS</td><td>124</td></tr>
<tr><td colspan="4">UCB$L_DUETIM*</td><td>128</td></tr>
<tr><td colspan="4">UCB$L_OPCNT*</td><td>132</td></tr>
<tr><td colspan="4">UCB$L_SVPN*</td><td>136</td></tr>
<tr><td colspan="4">UCB$L_SVAPTE*</td><td>140</td></tr>
<tr><td colspan="2">UCB$W_BCNT</td><td colspan="2">UCB$W_BOFF</td><td>144</td></tr>
<tr><td colspan="2">UCB$W_ERRCNT</td><td>UCB$B_ERTMAX</td><td>UCB$B_ERTCNT</td><td>148</td></tr>
<tr><td colspan="4">UCB$L_PDT*</td><td>152</td></tr>
<tr><td colspan="4">UCB$L_DDT*</td><td>156</td></tr>
<tr><td colspan="4">UCB$L_MEDIA_ID*</td><td>160</td></tr>
</table>

*A read-only field

Table 1–22 Contents of Unit Control Block

Field Name	Contents
UCB$L_FQFL*	Fork queue forward link. The link points to the next entry in the fork queue. EXE$IOFORK and system resource management routines write this field. The queue contains addresses of UCBs that contain driver fork process context of drivers waiting to continue I/O processing.
UCB$L_FQBL*	Fork queue backward link. The link points to the previous entry in the fork queue. EXE$IOFORK and system resource management routines write this field.
UCB$W_SIZE*	Size of UCB. The DPT of every driver must specify a value for this field. The driver-loading procedure uses the value to allocate space for a UCB and stores the value in each UCB created. Extra space beyond the standard bytes in a UCB (UCB$K_LENGTH) is for device-specific data and temporary storage.
UCB$B_TYPE*	Type of data structure. The driver-loading procedure writes the constant DYN$C_UCB into this field when the procedure creates the UCB.
UCB$B_FLCK	Index of the fork lock that synchronizes access to this UCB at fork level. The DPT of every driver must specify a value for this field. The driver-loading procedure writes the value in the UCB when the procedure creates the UCB. All devices that are attached to a single I/O adapter and actively compete for shared adapter resources and/or a controller data channel must specify the same value for this field. When the operating system creates a driver fork process to service an I/O request for a device, the fork process gains control at the IPL associated with the fork lock, holding the fork lock itself in a multiprocessing environment. When the driver creates a fork process after an interrupt, the operating system inserts the fork block into a processor-specific fork queue based on this fork IPL. A system fork dispatcher, executing at fork IPL, obtains the fork lock (if necessary), dequeues the fork block, and restores control to the suspended driver fork process. This field is also known as UCB$B_FIPL. Drivers designed to execute exclusively in a uniprocessing environment store the fork IPL associated with the UCB in this field.
UCB$L_FPC	Fork process driver PC address. When a system routine saves driver fork context in order to suspend driver execution, the routine stores the address of the next driver instruction to be executed in this field. A system routine that reactivates a suspended driver transfers control to the saved PC address. System routines that suspend driver processing include EXE$IOFORK, IOC$REQ*x*CHAN*y*, IOC$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. When a driver interrupt service routine determines that a device is expecting an interrupt, the routine restores control to the saved PC address in the device's UCB.
UCB$L_FR3	Value of R3 at the time that a system routine suspends a driver fork process. The value of R3 is restored just before a suspended driver regains control.
UCB$L_FR4	Value of R4 at the time that a system routine suspends a driver fork process. The value of R4 is restored just before a suspended driver regains control.
UCB$W_BUFQUO*	Buffered-I/O quota if the UCB represents a mailbox.
UCB$W_INIQUO*	Initial buffered-I/O quota if the UCB represents a mailbox.

(continued on next page)

Table 1–22 (Cont.) Contents of Unit Control Block

Field Name	Contents
UCB$L_ORB*	Address of ORB associated with the UCB. SYSGEN places the address in this field when you use SYSGEN's CONNECT command.
UCB$L_LOCKID*	Lock management lock ID of device allocation lock. A lock management lock is used for device allocation so that device allocation functions properly for cluster-accessible devices in a VAXcluster (DEV$V_CLU set within UCB$L_DEVCHAR2).
UCB$L_CRB*	Address of primary CRB associated with the device. The driver-loading procedure writes this field after it creates the associated CRB. Driver fork processes read this field to gain access to device registers. System routines use UCB$L_CRB to locate interrupt-dispatching code and the addresses of driver unit and controller initialization routines.
UCB$L_DLCK*	Address of device lock that—in a multiprocessing environment—synchronizes access to device registers and those fields in the UCB accessed at device IPL. The driver-loading routine copies the address of the device lock in the CRB (CRB$L_DLCK) to this field as it creates a UCB for each device on a controller.
UCB$L_DDB*	Address of DDB associated with device. The driver-loading procedure writes this field when the procedure creates the associated UCB. System routines generally read the DDB field in order to locate device driver entry points, the address of a driver FDT, or the ACP associated with a given device.
UCB$L_PID*	Process identification number of the process that has allocated the device. Written by the $ALLOC system service.
UCB$L_LINK*	Address of next UCB in the chain of UCBs attached to a single controller and associated with a DDB. The driver-loading procedure writes this field when the procedure adds the next UCB. Any system routine that examines the status of all devices on the system reads this field. Such routines include EXE$TIMEOUT, IOC$SEARCHDEV, and power failure recovery routines.
UCB$L_VCB*	Address of volume control block (VCB) that describes the volume mounted on the device. This field is written by the device's ACP and read by EXE$QIOACPPKT, ACPs, and the XQP.
UCB$L_DEVCHAR	First longword of device characteristics bits. The DPT of every driver should specify symbolic constant values (defined by the $DEVDEF macro in SYS$LIBRARY:STARLET.MLB) for this field. The driver-loading procedure writes the field when the procedure creates the UCB. The $QIO system service reads the field to determine whether a device is spooled, file structured, shared, has a volume mounted, and so on. The system defines the following device characteristics: DEV$V_REC Record-oriented device DEV$V_CCL Carriage control device DEV$V_TRM Terminal device DEV$V_DIR Directory-structured device DEV$V_SDI Single directory-structured device DEV$V_SQD Sequential block-oriented device (magnetic tape, for example) DEV$V_SPL Device spooled

(continued on next page)

Table 1–22 (Cont.) Contents of Unit Control Block

Field Name	Contents	
	DEV$V_OPR	Operator device
	DEV$V_RCT	Device contains RCT
	DEV$V_NET	Network device
	DEV$V_FOD	File-oriented device (disk and magnetic tape, for example)
	DEV$V_DUA	Dual-ported device
	DEV$V_SHR	Shareable device (used by more than one program simultaneously)
	DEV$V_GEN	Generic device
	DEV$V_AVL	Device available for use
	DEV$V_MNT	Device mounted
	DEV$V_MBX	Mailbox device
	DEV$V_DMT	Device marked for dismount
	DEV$V_ELG	Error logging enabled
	DEV$V_ALL	Device allocated
	DEV$V_FOR	Device mounted as foreign (not file structured)
	DEV$V_SWL	Device software write-locked
	DEV$V_IDV	Device capable of providing input
	DEV$V_ODV	Device capable of providing output
	DEV$V_RND	Device allowing random access
	DEV$V_RTM	Real-time device
	DEV$V_RCK	Read-checking enabled
	DEV$V_WCK	Write-checking enabled
UCB$L_DEVCHAR2	Second longword of device characteristics. The DPT of every driver should specify symbolic constant values (defined by the $DEVDEF macro in SYS$LIBRARY:STARLET.MLB) for this field. The driver-loading procedure writes the field when the procedure creates the UCB. The system defines the following device characteristics:	
	DEV$V_CLU	Device available clusterwide
	DEV$V_DET	Detached terminal
	DEV$V_RTT	Remote-terminal UCB extension
	DEV$V_CDP	Dual-pathed device with two UCBs
	DEV$V_2P	Two paths known to device
	DEV$V_MSCP	Disk or tape accessed using MSCP
	DEV$V_SSM	Shadow set member
	DEV$V_SRV	Served by MSCP server

(continued on next page)

Table 1–22 (Cont.) Contents of Unit Control Block

Field Name	Contents	
	DEV$V_RED	Redirected terminal
	DEV$V_NNM	Device name has a prefix of the format “*node*$”
	DEV$V_WBC	Device supports write-back caching
	DEV$V_WTC	Device supports write-through caching
	DEV$V_HOC	Device supports host caching
	DEV$V_LOC	Device is accessible via local (non-emulated) controller
	DEV$V_DFS	Device is DFS-served
	DEV$V_DAP	Device is DAP accessed
	DEV$V_NLT	Device is not-last-track; that is, it has no bad block information on its last track
	DEV$V_SEX	Device supports serious exception handling (tape)
	DEV$V_SHD	Device is a member of a host-based shadow set
	DEV$V_VRT	Device is a shadow set virtual unit
	DEV$V_LDR	Loader is present (tape)
	DEV$V_NOLB	Device ignores server load balancing requests
	DEV$V_NOCLU	Device will never be available clusterwide
	DEV$V_VMEM	Device is a virtual member of a constituent set
	DEV$V_SCSI	Device is a SCSI device
	DEV$V_WLG	Device has write logging capability
	DEV$V_NOFE	Device does not support forced error
	DEV$V_AIP	Allocation is in progress (MME)
	DEV$V_CRAMIO	Device supports CRAM mailbox I/O
UCB$L_AFFINITY*	Bit mask of the CPU-IDs of processors in a multiprocessing system that have physical connectivity to the device. Such processors can thereby access the device's registers and initiate I/O operations on the device.	
UCB$L_XTRA	SMP alternate STARTIO wait.	
UCB$B_DEVCLASS	Device class. The DPT of every driver should specify a symbolic constant (defined by the $DCDEF macro) for this field. The driver-loading procedure writes this field when it creates the UCB.	
	Drivers with set mode and device characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request.	
	The following device classes are defined:	
	DC$_DISK	Disk
	DC$_TAPE	Tape
	DC$_SCOM	Synchronous communications
	DC$_CARD	Card reader
	DC$_TERM	Terminal
	DC$_LP	Line printer

(continued on next page)

Table 1–22 (Cont.) Contents of Unit Control Block

Field Name	Contents	
	DC$_WORKSTATION	Workstation
	DC$_REALTIME	Real time
	DC$_BUS	Bus
	DC$_MAILBOX	Mailbox
	DC$_DECVOICE	DECVoice
	DC$_AUDIO	General audio
	DC$_REMCSL_ STORAGE	Remote console storage
	DC$_MISC	Miscellaneous
	Note that the definition of a device as a real-time device (DC$_REALTIME) is somewhat subjective; it implies no special treatment by the operating system.	
UCB$B_DEVTYPE	Device type. The DPT of every driver should specify a symbolic constant (defined by the $DCDEF macro) for this field. The driver-loading procedure writes the field when it creates the UCB. Drivers for devices with set mode and set characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request.	
UCB$W_DEVBUFSIZ	Default buffer size. The DPT can specify a value for this field if relevant. The driver-loading procedure writes the field when it creates the UCB. Drivers for devices with set mode and set characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request. This field is used by RMS for record I/O on nonfile devices.	
UCB$Q_DEVDEPEND	Device-descriptive data interpreted by the device driver itself. The DPT can specify a value for this field. The driver-loading procedure writes this field when it creates the UCB. Drivers for devices with set mode and set characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request.	
UCB$Q_DEVDEPEND2	Second longword for device-dependent status. This field is an extension of UCB$Q_DEVDEPEND.	
UCB$L_IOQFL*	Pending-I/O queue listhead forward link. The queue contains the addresses of IRPs waiting for processing on a device. EXE$INSERTIRP inserts IRPs into the pending-I/O queue when a device is busy. IOC$REQCOM dequeues IRPs when the device is idle. The queue is a priority queue that has the highest priority IRPs at the front of the queue. Priority is determined by the base priority of the requesting process. IRPs with the same priority are processed first-in/first-out.	
UCB$L_IOQBL*	Pending-I/O queue listhead backward link. EXE$INSERTIRP and IOC$REQCOM modify the pending-I/O queue.	
UCB$W_UNIT*	Number of the physical device unit; stored as a binary value. The driver-loading procedure writes a value into this field when it creates the UCB. Drivers for multiunit controllers read this field during unit initialization to identify a unit to the controller.	
UCB$W_CHARGE*	Mailbox byte count quota charge, if the device is a mailbox.	

(continued on next page)

Table 1–22 (Cont.) Contents of Unit Control Block

Field Name	Contents	
UCB$L_IRP	Address of IRP currently being processed on the device unit by the driver fork process. IOC$INITIATE writes the address of an IRP into this field before the routine creates a driver fork process to handle an I/O request. From this field, a driver fork process obtains the address of the IRP being processed. The value contained in this field is not valid if the UCB$V_BSY bit in UCB$L_STS is clear.	
UCB$W_REFC*	Reference count of processes that currently have process I/O channels assigned to the device. The $ASSIGN and $ALLOC system services increment this field. The $DASSGN and $DALLOC system services decrement this field.	
UCB$B_DIPL	Interrupt priority level (IPL) at which the device requests hardware interrupts. The DPT of every driver must specify a value for this field. The driver-loading procedure writes this field when the procedure creates the UCB. When the driver-loading procedure subsequently creates the device lock's spinlock structure (SPL), it moves the contents of this field into SPL$B_IPL. In a uniprocessing environment, device drivers raise IPL to device IPL before reading or writing device registers or accessing other fields in the UCB synchronized at device IPL. In a multiprocessing environment, drivers obtain the device lock at UCB$L_DLCK, thereby also raising IPL to device IPL in the process.	
UCB$B_AMOD*	Access mode at which allocation occurred, if the device is allocated. Written by the $ALLOC and $DALLOC system services.	
UCB$L_AMB*	Associated mailbox UCB pointer. A spooled device uses this field for the address of its associated device. Devices that are nonshareable and not file oriented can use this field for the address of an associated mailbox.	
UCB$L_STS	Device unit status (formerly UCB$W_STS). Written by drivers, IOC$REQCOM, IOC$CANCELIO, IOC$INITIATE, IOC$WFIKPCH, IOC$WFIRLCH, EXE$INSIOQ, and EXE$TIMEOUT. This field is read by drivers, the $QIO system service routines, IOC$REQCOM, IOC$INITIATE, and EXE$TIMEOUT. This longword includes the following bits:	
	UCB$V_TIM	Timeout enabled.
	UCB$V_INT	Interrupts expected.
	UCB$V_ERLOGIP	Error log in progress.
	UCB$V_CANCEL	Cancel I/O on unit.
	UCB$V_ONLINE	Device is on line.
	UCB$V_POWER	Power has failed while unit was busy.
	UCB$V_TIMOUT	Unit is timed out.
	UCB$V_INTTYPE	Receiver interrupt.
	UCB$V_BSY	Unit is busy.
	UCB$V_MOUNTING	Device is being mounted.
	UCB$V_DEADMO	Deallocate device at dismount.
	UCB$V_VALID	Volume appears valid to software.

(continued on next page)

Table 1–22 (Cont.) Contents of Unit Control Block

Field Name	Contents	
	UCB$V_UNLOAD	Unload volume at dismount.
	UCB$V_TEMPLATE	Template UCB from which other UCBs for this device are made. The $ASSIGN system service checks this bit in the requested UCB and, if the bit is set, creates a UCB from the template. The new UCB is assigned instead.
	UCB$V_MNTVERIP	Mount verification in progress.
	UCB$V_WRONGVOL	Volume name does not match name in the VCB.
	UCB$V_DELETEUCB	Delete this UCB when the value in UCB$W_REFC becomes zero.
	UCB$V_LCL_VALID	The volume on this device is valid on the local node.
	UCB$V_SUPMVMSG	Suppress mount-verification messages if they indicate success.
	UCB$V_MNTVERPND	Mount verification is pending on the device and the device is busy.
	UCB$V_DISMOUNT	Dismount in progress.
	UCB$V_CLUTRAN	VAXcluster state transition in progress.
	UCB$V_WRTLOCKMV	Write-locked mount verification in progress.
	UCB$V_SVPN_END	Last byte used from page is mapped by a system virtual page number.
	UCB$V_ALTBSY	Unit is busy via an alternate startup path.
	UCB$V_SNAPSHOT	Restart validation is in progress.
UCB$W_DEVSTS	Device-dependent status. Read and written by device drivers.	
	The system defines the following status bits:	
	UCB$V_JOB	Job controller has been notified.
	UCB$V_TEMPL_BSY	Template UCB is busy.
	UCB$V_PRMMBX	Device is a permanent mailbox.
	UCB$V_DELMBX	Mailbox is marked for deletion.
	UCB$V_SHMMBS	Device is shared-memory mailbox.
	Disk drivers use bits in UCB$W_DEVSTS as follows:	
	UCB$V_ECC	ECC correction made.
	UCB$V_DIAGBUF	Diagnostic buffer is specified.
	UCB$V_NOCNVRT	No logical block number to media address conversion.
	UCB$V_DX_WRITE	Console floppy write operation.
	UCB$V_DATACACHE	Data blocks are being cached.
UCB$W_QLEN*	Length of pending-I/O queue (pointed to by UCB$L_IOQFL).	

(continued on next page)

Table 1–22 (Cont.) Contents of Unit Control Block

Field Name	Contents
UCB$L_DUETIM*	Due time for I/O completion. Stored as the low-order 32-bit absolute time (time in seconds since the operating system was booted) at which the device will time out. IOC$WFIKPCH and IOC$WFIRLCH write this value when they suspend a driver to wait for an interrupt or timeout. EXE$TIMEOUT examines this field in each UCB in the I/O database once per second. If the timeout has occurred and timeouts are enabled for the device, EXE$TIMEOUT calls the device driver timeout handler.
UCB$L_OPCNT*	Count of operations completed on device unit since last bootstrap of the system. IOC$REQCOM writes this field every time the routine inserts an IRP into the I/O postprocessing queue.
UCB$L_SVPN*	Index to the virtual address of the system PTE that the driver loading procedure has permanently allocated to the device. The system virtual address of the page described by this index can be calculated by the following formula: $(\text{index} * 200_{16}) + 80000000_{16}$ If a DPT specifies DPT$M_SVP in the **flags** argument to the DPTAB macro, the driver-loading procedure allocates a page of nonpaged system memory to the device. The procedure writes the system PTE's index into UCB$L_SVPN when the procedure creates the UCB. Disk drivers use this field for ECC error correction.
UCB$L_SVAPTE	For a direct-I/O transfer, the virtual address of the system PTE for the first page to be used in the transfer; for a buffered-I/O transfer, the virtual address of the system buffer used in the transfer. IOC$INITIATE writes this field from IRP$L_SVAPTE before calling a driver start-I/O routine. Drivers read this value to compute the starting address of a transfer.
UCB$W_BOFF	For a direct-I/O transfer, the byte offset in the first page of the transfer buffer; for a buffered-I/O transfer, the number of bytes charged to the process for the transfer. IOC$INITIATE copies this field from the IRP. Drivers read the field in calculating the starting address of a DMA transfer. If only part of a DMA transfer succeeds, the driver adjusts the value in this field to be the byte offset in the first page of the data that was not transferred.
UCB$W_BCNT	Count of bytes in the I/O transfer. IOC$INITIATE copies this field from the IRP. Drivers read this field to determine how many bytes to transfer in an I/O operation.
UCB$B_ERTCNT	Error retry count of the current I/O transfer. The driver sets this field to the maximum retry count each time it begins I/O processing. Before each retry, the driver decreases the value in this field. During error logging, IOC$REQCOM copies the value into the error message buffer.
UCB$B_ERTMAX	Maximum error retry count allowed for single I/O transfer. The DPT of some drivers specifies a value for this field. The driver-loading procedure writes the field when the procedure creates the UCB. During error logging, IOC$REQCOM copies the value into the error message buffer.

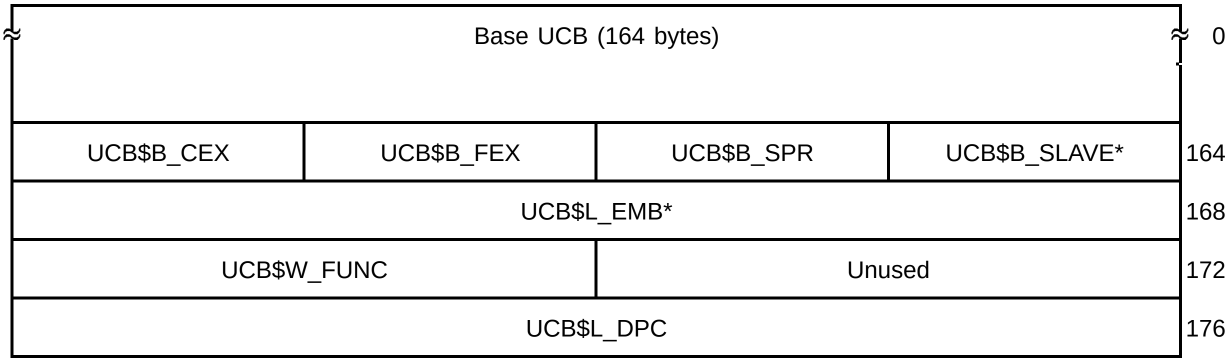
(continued on next page)

Table 1–22 (Cont.) Contents of Unit Control Block

Field Name	Contents
UCB$W_ERRCNT	Number of errors that have occurred on the device since the system was booted. The driver-loading procedure initializes the field to zero when the procedure creates the UCB. ERL$DEVICERR and ERL$DEVICTMO increment the value in the field and copy the value into an error message buffer. The DCL command SHOW DEVICE displays in its error count column the value contained in this field.
UCB$L_PDT*	Address of port descriptor table (PDT). This field is reserved for SCSI port drivers.
UCB$L_DDT*	Address of DDT for unit. The driver load procedure writes the contents of DDB$L_DDT for the device controller to this field when it creates the UCB.
UCB$L_MEDIA_ID*	Bit-encoded media name and type, used by MSCP devices.

The UCB error-log extension is illustrated in Figure 1–24 and described in Table 1–23.

Figure 1–24 UCB Error-Log Extension



*A read-only field

Table 1–23 UCB Error-Log Extension

Field Name	Contents
UCB$B_SLAVE*	Unit number of slave controller.
UCB$B_SPR	Spare byte. This field is reserved for driver use. MASSBUS adapter drivers use this field to store a fixed offset to the MASSBUS adapter registers for the unit.
UCB$B_FEX	Device-specific field. This field is reserved for driver use. Certain system disk drivers (such as DLDRIVER in one of the appendixes to the *OpenVMS VAX Device Support Manual*) use this field to store an index in a hardware function dispatch table.

(continued on next page)

Table 1–23 (Cont.) UCB Error-Log Extension

Field Name	Contents
UCB$B_CEX	Device-specific field. This field is reserved for driver use. Certain system disk drivers (such as DLDRIVER in one of the appendixes to the *OpenVMS VAX Device Support Manual*) use this field to store an index into a software function case table.
UCB$L_EMB*	Address of error message buffer. If error logging is enabled and a device /controller error or timeout occurs, the driver calls ERL$DEVICERR or ERL$DEVICTMO to allocate an error message buffer and copy the buffer address into this field. IOC$REQCOM writes final device status, error counters, and I/O request status into the buffer specified by this field.
UCB$W_FUNC	I/O function modifiers. This field is read and written by drivers that log errors.
UCB$L_DPC	Device-specific field. This field is reserved for driver use. Certain system disk drivers (such as DLDRIVER in one of the appendixes to the *OpenVMS VAX Device Support Manual*) use this field to store the driver's return PC across a dispatch to a hardware function routine.

The UCB local tape extension is illustrated in Figure 1–25 and described in Table 1–24.

Figure 1–25 UCB Local Tape Extension

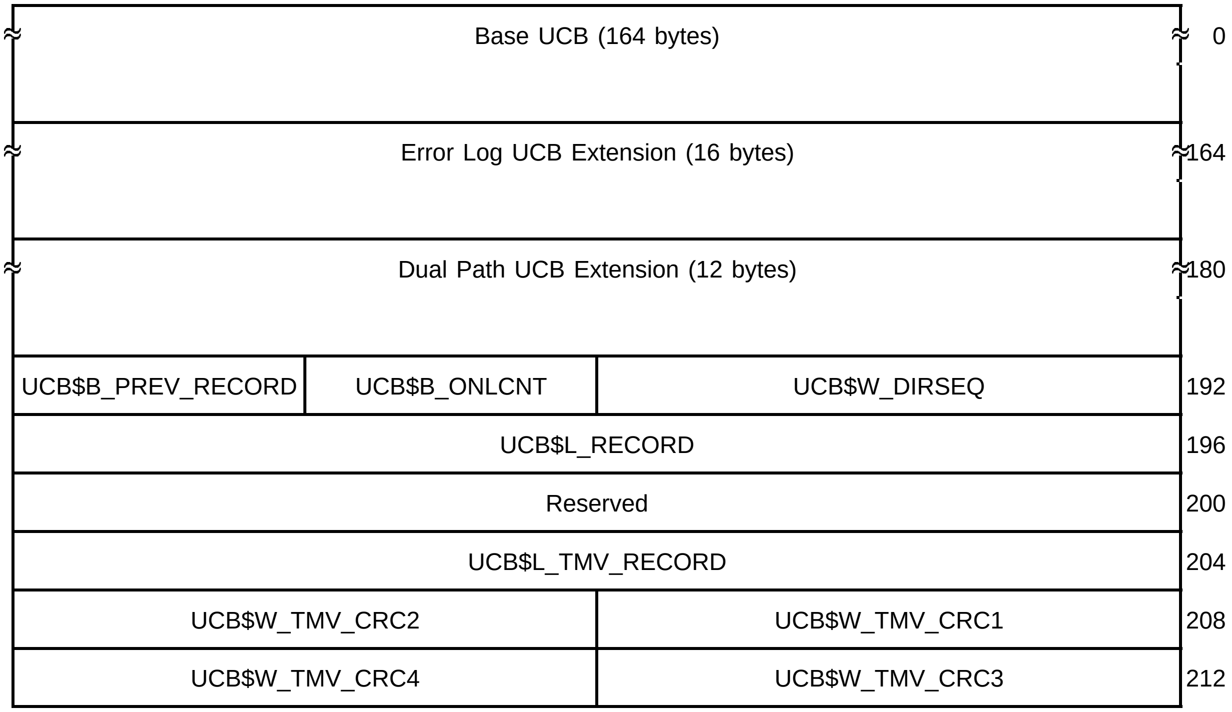


Table 1–24 UCB Local Tape Extension

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$B_ONLCNT	Number of times the device has been placed on line since the system was last bootstrapped.
UCB$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.

The UCB local disk extension is illustrated in Figure 1–26 and described in Table 1–25.

Figure 1–26 UCB Local Disk Extension

<table>
<tr><td colspan="3">Base UCB (164 bytes)</td><td>0</td></tr>
<tr><td colspan="3">Error Log UCB Extension (16 bytes)</td><td>164</td></tr>
<tr><td colspan="3">Dual Path UCB Extension (12 bytes)</td><td>180</td></tr>
<tr><td>Reserved</td><td>UCB$B_ONLCNT</td><td>UCB$W_DIRSEQ</td><td>192</td></tr>
<tr><td colspan="3">UCB$L_MAXBLOCK</td><td>196</td></tr>
<tr><td colspan="3">UCB$L_MAXBCNT</td><td>200</td></tr>
<tr><td colspan="3">UCB$L_DCCB</td><td>204</td></tr>
<tr><td colspan="3">UCB$L_QLENACC</td><td>208</td></tr>
<tr><td colspan="3">UCB$L_MEDIA</td><td>212</td></tr>
<tr><td colspan="3">UCB$L_BCR</td><td>216</td></tr>
<tr><td colspan="2">UCB$W_EC2</td><td>UCB$W_EC1</td><td>220</td></tr>
<tr><td>UCB$B_OFFRTC</td><td>UCB$B_OFFNDX</td><td>UCB$W_OFFSET</td><td>224</td></tr>
</table>

(continued on next page)

UCB$L_DX_BUF			228
UCB$L_DX_BFPNT			232
UCB$L_DX_RXDB			236
Unused	UCB$B_DX_SCTCNT	UCB$W_DX_BCR	240

Table 1–25 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$B_ONLCNT	Number of times device has been placed on line since the system was last bootstrapped.
UCB$L_MAXBLOCK	Maximum number of logical blocks on random-access device. This field is written by a disk driver during unit initialization and power recovery.
UCB$L_MAXBCNT	Maximum number of bytes that can be transferred. A disk driver writes this field during unit initialization and power recovery.
UCB$L_DCCB	Pointer to cache control block.
UCB$L_QLENACC	Queue length accumulator.
UCB$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$W_EC1	ECC position register. This field records the starting bit number of an error burst. Disk driver register dumping routines copy the contents of this field into an error message or diagnostic buffer. The system correction routine IOC$APPLYECC reads the contents of this field to locate the beginning of an error burst in a disk block.
UCB$W_EC2	ECC position register. Records the exclusive OR correction pattern. Disk driver register dumping routines copy the contents of this field into an error message or diagnostic buffer. The system ECC correction routine IOC$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).

The UCB terminal extension is illustrated in Figure 1–27 and described in Table 1–26.

Figure 1–27 UCB Terminal Extension

<table>
<tr><td colspan="4">Base UCB (164 bytes)</td><td>0</td></tr>
<tr><td colspan="4">UCB$L_TL_CTRLY*</td><td>164</td></tr>
<tr><td colspan="4">UCB$L_TL_CTRLC*</td><td>168</td></tr>
<tr><td colspan="4">UCB$L_TL_OUTBAND*</td><td>172</td></tr>
<tr><td colspan="4">UCB$L_TL_BANDQUE*</td><td>176</td></tr>
<tr><td colspan="4">UCB$L_TL_PHYUCB*</td><td>180</td></tr>
<tr><td colspan="4">UCB$L_TL_CTLPID*</td><td>184</td></tr>
<tr><td colspan="4">UCB$Q_TL_BRKTHRU*</td><td>188</td></tr>
<tr><td colspan="4">UCB$L_TT_RDUE*</td><td>196</td></tr>
<tr><td colspan="4">UCB$L_TT_RTIMOU*</td><td>200</td></tr>
<tr><td colspan="4">UCB$L_TT_STATE1*</td><td>204</td></tr>
<tr><td colspan="4">UCB$L_TT_STATE2*</td><td>208</td></tr>
<tr><td colspan="4">UCB$L_TT_LOGUCB*</td><td>212</td></tr>
<tr><td colspan="4">UCB$L_TT_DECHAR*</td><td>216</td></tr>
<tr><td colspan="4">UCB$L_TT_DECHA1*</td><td>220</td></tr>
<tr><td colspan="4">UCB$L_TT_DECHA2*</td><td>224</td></tr>
<tr><td colspan="4">UCB$L_TT_DECHA3*</td><td>228</td></tr>
<tr><td colspan="4">UCB$L_TT_WFLINK*</td><td>232</td></tr>
<tr><td colspan="4">UCB$L_TT_WBLINK*</td><td>236</td></tr>
<tr><td colspan="4">UCB$L_TT_WRTBUF*</td><td>240</td></tr>
<tr><td colspan="4">UCB$L_TT_MULTI*</td><td>244</td></tr>
<tr><td colspan="2">UCB$W_TT_SMLTLEN*</td><td colspan="2">UCB$W_TT_MULTILEN*</td><td>248</td></tr>
<tr><td colspan="4">UCB$L_TT_SMLT*</td><td>252</td></tr>
<tr><td>UCB$B_TT_DELFF*</td><td>UCB$B_TT_DECRF*</td><td colspan="2">UCB$W_TT_DESPEE*</td><td>256</td></tr>
<tr><td colspan="3">Unused</td><td>UCB$B_TT_DEPARI*</td><td>260</td></tr>
</table>

(continued on next page)

<table>
<tr><td>Reserved</td><td colspan="2">UCB$W_TT_DESIZE*</td><td>UCB$B_TT_DETYPE*</td><td>264</td></tr>
<tr><td>UCB$B_TT_LFFILL*</td><td>UCB$B_TT_CRFILL*</td><td>UCB$B_TT_RSPEED*</td><td>UCB$B_TT_TSPEED*</td><td>268</td></tr>
<tr><td colspan="3">Unused</td><td>UCB$B_TT_PARITY*</td><td>272</td></tr>
<tr><td colspan="4">UCB$L_TT_TYPAHD*</td><td>276</td></tr>
<tr><td>UCB$B_TT_LASTC*</td><td>UCB$B_TT_LINE*</td><td colspan="2">UCB$W_TT_CURSOR*</td><td>280</td></tr>
<tr><td>UCB$B_TT_ESC*</td><td>UCB$B_TT_FILL*</td><td colspan="2">UCB$W_TT_BSPLEN*</td><td>284</td></tr>
<tr><td colspan="2">UCB$W_TT_UNITBIT*</td><td>UCB$B_TT_INTCNT*</td><td>UCB$B_TT_ESC_O*</td><td>288</td></tr>
<tr><td>UCB$B_TT_OUTYPE*</td><td>UCB$B_TT_PREMPT</td><td colspan="2">UCB$W_TT_HOLD</td><td>292</td></tr>
<tr><td colspan="4">UCB$L_TT_GETNXT*</td><td>296</td></tr>
<tr><td colspan="4">UCB$L_TT_PUTNXT*</td><td>300</td></tr>
<tr><td colspan="4">UCB$L_TT_CLASS*</td><td>304</td></tr>
<tr><td colspan="4">UCB$L_TT_PORT</td><td>308</td></tr>
<tr><td colspan="4">UCB$L_TT_OUTADR</td><td>312</td></tr>
<tr><td colspan="2">UCB$W_TT_PRTCTL</td><td colspan="2">UCB$W_TT_OUTLEN*</td><td>316</td></tr>
<tr><td colspan="2">UCB$W_TT_DS_ST*</td><td>UCB$B_TT_DS_TX</td><td>UCB$B_TT_DS_RCV</td><td>320</td></tr>
<tr><td>UCB$B_TT_OLD*</td><td>UCB$B_TT_MAINT*</td><td colspan="2">UCB$W_TT_DS_TIM*</td><td>324</td></tr>
<tr><td colspan="4">UCB$L_TT_FBK*</td><td>328</td></tr>
<tr><td colspan="4">UCB$L_TT_RDVERIFY*</td><td>332</td></tr>
<tr><td colspan="4">UCB$L_TT_CLASS1*</td><td>336</td></tr>
<tr><td colspan="4">UCB$L_TT_CLASS2*</td><td>340</td></tr>
<tr><td colspan="4">UCB$L_TT_ACCPORNAM</td><td>344</td></tr>
<tr><td colspan="4">UCB$L_TP_MAP*</td><td>348</td></tr>
<tr><td colspan="3">Unused</td><td>UCB$B_TP_STAT*</td><td>352</td></tr>
</table>

Table 1–26 UCB Terminal Extension

Field Name	Contents	
UCB$L_TL_CTRLY*	Listhead of CTRL/Y AST control blocks (ACBs).	
UCB$L_TL_CTRLC*	Listhead of CTRL/C ACBs.	
UCB$L_TL_OUTBAND*	Out-of-band character mask.	
UCB$L_TL_BANDQUE*	Listhead of out-of-band ACBs.	
UCB$L_TL_PHYUCB*	Address of physical UCB.	
UCB$L_TL_CTLPID*	Process ID of controlling process (used with SPAWN).	
UCB$Q_TL_BRKTHRU*	Facility broadcast bit mask.	
UCB$L_TT_RDUE*	Absolute time at which a read timeout is due.	
UCB$L_TT_RTIMOU*	Address of read timeout routine.	
UCB$L_TT_STATE1*	First longword of terminal state information.	
	The following fields are defined within UCB$L_TT_STATE1:	
	TTY$V_ST_POWER	Power failure
	TTY$V_ST_CTRLS	Class output
	TTY$V_ST_FILL	Fill mode
	TTY$V_ST_CURSOR	Cursor
	TTY$V_ST_SENDLF	Forced line feed
	TTY$V_ST_BACKSPACE	Backspace
	TTY$V_ST_MULTI	Multi-echo
	TTY$V_ST_WRITE	Write in progress
	TTY$V_ST_EOL	End of line
	TTY$V_ST_EDITREAD	Editing read in progress
	TTY$V_ST_RDVERIFY	Read verify in progress
	TTY$V_ST_RECALL	Command recall
	TTY$V_ST_READ	Read in progress
UCB$L_TT_STATE2*	Second longword of terminal state information.	
	The following fields are defined within UCB$L_TT_STATE2:	
	TTY$V_ST_CTRLO	Output enable
	TTY$V_ST_DEL	Delete
	TTY$V_ST_PASALL	Pass-all mode
	TTY$V_ST_NOECHO	No echo
	TTY$V_ST_WRTALL	Write-all mode
	TTY$V_ST_PROMPT	Prompt
	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

(continued on next page)

Table 1–26 (Cont.) UCB Terminal Extension

Field Name	Contents	
	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
	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.	

(continued on next page)

Table 1–26 (Cont.) UCB Terminal Extension

<table>
<tr><th>Field Name</th><th>Contents</th></tr>
<tr><td>UCB$B_TT_DELFF*</td><td>Default line-feed fill.</td></tr>
<tr><td>UCB$B_TT_DEPARI*</td><td>Default parity/character size.</td></tr>
<tr><td>UCB$B_TT_DETYPE*</td><td>Default terminal type.</td></tr>
<tr><td>UCB$W_TT_DESIZE*</td><td>Default line size.</td></tr>
<tr><td>UCB$W_TT_SPEED*</td><td>Terminal line speed. This field is read and written by the class driver, and read by the port driver. It contains the following byte fields:
<table>
<tr><td>UCB$B_TT_TSPEED</td><td>Transmit speed</td></tr>
<tr><td>UCB$B_TT_RSPEED</td><td>Receive speed</td></tr>
</table></td></tr>
<tr><td>UCB$B_TT_CRFILL*</td><td>Number of fill characters to be output for carriage return.</td></tr>
<tr><td>UCB$B_TT_LFFILL*</td><td>Number of fill characters to be output for line feed.</td></tr>
<tr><td>UCB$B_TT_PARITY*</td><td>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:
<table>
<tr><td>UCB$V_TT_XXPARITY</td><td>Reserved to Digital.</td></tr>
<tr><td>UCB$V_TT_DISPARERR</td><td>Reserved to Digital.</td></tr>
<tr><td>UCB$V_TT_USERFRAME</td><td>Reserved to Digital.</td></tr>
<tr><td>UCB$V_TT_LEN</td><td>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.</td></tr>
<tr><td>UCB$V_TT_STOP</td><td>Number of stop bits: clear if one stop bit; set if two stop bits.</td></tr>
<tr><td>UCB$V_TT_PARTY</td><td>Parity checking. This bit is set if parity checking is enabled.</td></tr>
<tr><td>UCB$V_TT_ODD</td><td>Parity type: clear if even parity; set if odd parity.</td></tr>
</table></td></tr>
<tr><td>UCB$L_TT_TYPAHD*</td><td>Address of type-ahead buffer.</td></tr>
<tr><td>UCB$W_TT_CURSOR*</td><td>Current cursor position.</td></tr>
<tr><td>UCB$B_TT_LINE*</td><td>Current line position on page.</td></tr>
<tr><td>UCB$B_TT_LASTC*</td><td>Last formatted output character.</td></tr>
<tr><td>UCB$W_TT_BSPLEN*</td><td>Number of back spaces to output for non-ANSI terminals.</td></tr>
<tr><td>UCB$B_TT_FILL*</td><td>Current fill character count.</td></tr>
<tr><td>UCB$B_TT_ESC*</td><td>Current read escape syntax state.</td></tr>
<tr><td>UCB$B_TT_ESC_O*</td><td>Current write escape syntax state.</td></tr>
<tr><td>UCB$B_TT_INTCNT*</td><td>Number of characters in interrupt string.</td></tr>
<tr><td>UCB$W_TT_UNITBIT*</td><td>Enable and disable modem control.</td></tr>
</table>

(continued on next page)

Table 1–26 (Cont.) UCB Terminal Extension

Field Name	Contents
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$B_TT_OUTYPE contains –1. It is read and written by the port driver, and written by the class driver.
UCB$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.

(continued on next page)

Table 1–26 (Cont.) UCB Terminal Extension

Field Name	Contents	
	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.	
UCB$B_TT_MAINT*	Maintenance functions. This field is used as the argument to the port driver's PORT_MAINT routine. It is written by the class driver and read by the port driver.	
	It contains several bits that allow the following maintenance functions:	
	IO$M_LOOP	Set loopback mode.
	IO$M_UNLOOP	Reset loopback mode.
	IO$M_AUTXOF_ENA	Enable the use of auto XON/XOFF on this line. This is the default.
	IO$M_AUTXOF_DIS	Disable the use of auto XON/XOFF on this line.
	IO$M_LINE_OFF	Disable interrupts on this line.
	IO$M_LINE_ON	Reenable interrupts on this line.
	Reference these bits by using the mask, shifted as follows:	

```
BITB   #IO$M_LOOP@-
       7,UCB$B_TT_MAINT(R5)    ;Set loopback mode
```

Field Name	Contents
	UCB$B_TT_MAINT also defines the bit UCB$V_TT_DSBL that, when set, indicates that the line has been disabled.
UCB$B_OLD*	The full name of this field is UCB$B_TT_OLDCPZORG; it currently serves as a filler byte.

(continued on next page)

Table 1–26 (Cont.) UCB Terminal Extension

Field Name	Contents	
UCB$L_TT_FBK*	Address of fallback block.	
UCB$L_TT_RDVERIFY*	Address of read/verify table. Reserved for future use.	
UCB$L_TT_CLASS1*	First class driver longword.	
UCB$L_TT_CLASS2*	Second class driver longword.	
UCB$L_TT_ACCPORNAM	Address of counted string.	
UCB$L_TP_MAP*	UNIBUS/Q22–bus map registers.	
UCB$B_TP_STAT	DMA port-specific status.	
	The following fields are defined within UCB$B_TP_STAT.	
	TTY$V_TP_ABORT	DMA abort requested on this line.
	TTY$V_TP_ALLOC	Allocate map fork in progress.
	TTY$V_TP_DLLOC	Deallocate map fork in progress.

2

System Macros Invoked by Drivers

This chapter describes system macros that are the most frequently used by device drivers. When referring to these macro descriptions note the following conventions:

- If an argument is enclosed in brackets, you can choose to include that argument or omit it.
- The operating system 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 chapter, the format signifies such arguments with an equal sign (=) separating the argument from its keyword. For example:

  SETIPL [**ipl**=**31**]
- If an argument takes a keyword value, specify the keyword value using all uppercase letters. For example:

  preserve=**YES**
  condition=**RESTORE**

General information about the structure of macros and their arguments appears in the *VAX MACRO and Instruction Set Reference Manual*.

ADPDISP

Causes a branch to a specified address given the existence of a selected adapter characteristic.

Format

ADPDISP select ,addrlist [,adpaddr] [,crbaddr] [,ucbaddr] [,ecrbaddr] [,scratch=R0]

Parameters

select

Determines which ADP field or bit field is the basis for dispatching and, by implication, which adapter characteristic. See the Description section that follows for a list of legal values for **select**.

addrlist

A list containing one or more pairs of arguments in the following format:

<flag, destination>

The values the ADPDISP macro accepts for the **flag** argument depend on the adapter characteristic specified in **select** and are listed in the Description section that follows. The **destination** argument contains the address to which the code generated by the invocation of ADPDISP passes control if the specified **flag** is set.

[adpaddr]

Register containing the address of the adapter control block. If **adpaddr** is not specified, one of the following address fields must be specified.

[crbaddr]

Register containing the address of the channel request block.

[ucbaddr]

Register containing the address of the unit control block.

[ecrbaddr]

Register containing the address of the Ethernet controller data block (ECRB).

[scratch=R0]

Register, destroyed in macro invocation, used in computing the ADP address if **adpaddr** is not specified.

Description

ADPDISP dispatches upon the possible adapter characteristics listed in Table 2–1.

Table 2–1 Selectable Adapter Characteristics

Select Argument	Possible Value of *flag* in *addrlist*	Definition
ADAP_TYPE	UBA, MBA, GENBI, DR, or NULL. (See those symbols prefixed with AT$ defined by the $DCDEF macro in SYS$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 (8192) 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 you need an autopurging datapath that supports direct vectoring, 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 to determine the ADP address.

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.

BI_NODE_RESET

Initiates BIIC self-test on the specified VAXBI node.

Format

BI_NODE_RESET   csr

Parameters

csr
General purpose register that contains the address of the VAXBI node's control and status register (CSR).

Description

The BI_NODE_RESET macro uses the recommended instruction sequence to disable arbitration on the specified VAXBI node, and sets the node reset and self-test status bits in the BIIC CSR. The use of any instruction sequence other than that defined by the BI_NODE_RESET macro to perform these actions may cause an undefined condition on the VAXBI bus.

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$:
```

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).

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$L_TT_CLASS	Class driver vector table address
UCB$L_TT_PORT	Port driver vector table address
UCB$L_TT_GETNXT	Address of the class driver's get-next-character routine (CLASS_GETNXT)
UCB$L_TT_PUTNXT	Address of the class driver's put-next-character routine (CLASS_PUTNXT)
UCB$L_DDT	Address of the terminal class driver's driver dispatch table

Before invoking this macro, the unit initialization should place in R0 the address of the port driver vector table.

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 VAX system by type as listed in Table 2–2.

Table 2–2 VAX Systems and Their CPU Type

CPU Type	VAX System
1701	VAX 7000-6*xx*/10000-6*xx*
1303	VAX 4000-1*xx*/MicroVAX 3100-90
1302	VAX 6000-6*xx*
1202	VAX 6000-5*xx*
9AQ	VAX 9000-2*xx*/9000-4*xx*
9RR	VAX 6000-4*xx*
9CC	VAX 6000-2*xx*/6000-3*xx*/62*xx*/63*xx*
8PS	VAX 8810/8820/8830
8NN	VAX 8530/8550/8700/8800
790	VAX 8600/8650
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
690	VAX 4000-400/4000-500/4000-600
670	VAX 4000-300
660	VAX 4000-200
650	MicroVAX 3400/3600/3900-series system
520	VAX 3000FT
440	VAXstation 4000-VLC

---

[1] Because 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.

(continued on next page)

Table 2–2 (Cont.) VAX Systems and Their CPU Type

CPU Type	VAX System
420	VAXstation 3100/MicroVAX 3100
410	VAXstation 2000/MicroVAX 2000
60	VAXstation 3520/3540
46	VAXstation 4000-60
UV2	MicroVAX II

The CPUDISP macro identifies the VAX system by type and subtype as listed in Table 2–3.

Table 2–3 VAX Systems and Their CPU 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
	420	VAXstation 3100/MicroVAX 3100
	520	VAX 3000FT
	650	MicroVAX 3400/3600/3900-series system
	9CC	VAX 6200/6300-series system
	60	VAXstation 3520/3540
RV		CVAX-Rigel processor-based system
	9RR	VAX 6000-4*xx*
	670	VAX 4000-300
V12		Mariah processor-based systems
	1202	VAX 6000-5*xx*
	46	VAXstation 4000-60
V13		NVAX processor-based systems
	1302	VAX 6000-6*xx*
	690	VAX 4000-400/-500/-600

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_CPUTYPE does not correspond to any of the values specified as the **CPU-type** in the **addrlist** argument. A fatal bugcheck of UNSUPRTCPU occurs if the dispatching code does not find the executing processor identified in the **addrlist** and the value of **continue** is NO.

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$GB_CPUTYPE. These constants have the form PR$_SID_TYP*CPU-type*.

For each CPU subtype it encounters in the **addrlist** argument, CPUDISP also constructs symbolic constants of the form PR$_XSID_*xx_yyy*, where *xx* is the generic CPU type (for example CV) and *yyy* is the CPU subtype (420, 520, 650, 9CC, or 60 for CV). It compares the value of PR$_XSID_*xx_yyy* against the contents of EXE$GB_CPUDATA+15.

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$RETURN]
Address of alternate start-I/O routine. To initiate this routine, a driver FDT routine exits by means of system routine EXE$ALTQUEPKT instead of EXE$QIODRVPKT.

[mntver=+IOC$MNTVER]
Address of the system routine that is called at the beginning and end of a mount verification operation. The default, IOC$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 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 the operating system: 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 the operating system, its address is preceded by a plus sign (+).

$DEF

Defines a data-structure field within the context of a $DEFINI macro.

Format

$DEF sym [,alloc] [,siz]

Parameters

sym
Name of the symbol to access the field.

[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 $EQULST 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 $EQULST macros for additional information on defining symbols for data structure fields.

$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 $EQULST 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, 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
```

DEVICELOCK

Achieves synchronized access to a device's database as appropriate to the processing environment.

Format

DEVICELOCK [lockaddr] [,lockipl] [,savipl] [,condition] [,preserve=YES]

Parameters

[lockaddr]

Address of the device lock to be obtained. If **lockaddr** is not present, DEVICELOCK presumes that R5 contains the address of the UCB and uses the value at UCB$L_DLCK(R5) as the lock address.

[lockipl]

Location containing the IPL at which the device database is synchronized. In a uniprocessing environment, the DEVICELOCK macro sets IPL to the specified **lockipl**; if no **lockipl** is specified, it obtains the synchronization IPL from the device lock's data structure. In a multiprocessing environment, the system routine called by DEVICELOCK raises IPL to the IPL value contained in the device lock's data structure, regardless of whether the **lockipl** argument is present.

Digital recommends that you specify a **lockipl** value to facilitate debugging.

[savipl]

Location at which to save the current IPL.

[condition]

Indication of a special use of the macro. The only defined **condition** is **NOSETIPL**, which causes the macro to omit setting IPL. In some instances, setting IPL is undesirable or unnecessary when a driver obtains a device lock. For example, when an interrupt service routine issues the DEVICELOCK macro, the dispatching of the device interrupt has already raised IPL to device IPL.

[preserve=YES]

Indication that the macro should preserve R0 across the 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 DEVICELOCK macro raises IPL to **lockipl** (if **condition**=**NOSETIPL** is not specified).

In a multiprocessing environment, the DEVICELOCK macro performs the following actions:

- Preserves R0 through the macro call (if **preserve**=**YES** is specified).
- Stores the address of the device lock in R0.

- 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 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.

DEVICEUNLOCK

Relinquishes synchronized access to a device's database as appropriate to the processing environment.

Format

DEVICEUNLOCK [lockaddr] [,newipl] [,condition] [,preserve=YES]

Parameters

[lockaddr]

Address of the device lock to be released or restored. If **lockaddr** is not present, DEVICEUNLOCK presumes that R5 contains the address of the UCB and uses the value at UCB$L_DLCK(R5) as the lock address.

[newipl]

Location containing the IPL to which to lower. A prior invocation of the DEVICELOCK 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 multiprocessing environment—to call SMP$RESTOREL instead of SMP$RELEASEL. This releases a single acquisition of the spinlock 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 DEVICEUNLOCK 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 DEVICEUNLOCK macro performs the following tasks:

- Preserves R0 through the macro call (if **preserve**=**YES** is specified).
- Stores the address of the device lock in R0.
- Calls SMP$RELEASEL or, if **condition**=**RESTORE** is specified, SMP$RESTOREL.
- 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 DEVICELOCK macro sets the SMP-modified bit in the driver prologue table (DPT$V_SMPMOD in DPT$L_FLAGS).

Example

```
     DEVICELOCK -
             LOCKADDR=UCB$L_DLCK(R5),- ;Lock device access
             CONDITION=NOSETIPL,-      ;Do not set IPL
             PRESERVE=NO               ;Do not preserve R0
   .
   .
   .
20$: 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$MOVTOUSER. When this flag is set, the driver-loading procedure allocates a permanent system page-table entry (SPTE) for the device. It stores an index to the virtual address of the SPTE in UCB$L_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.

DPT$M_SMPMOD	Indicates that the driver has been designed to execute within a multiprocessing environment. Use of any of the multiprocessing synchronization macros (DEVICELOCK/DEVICEUNLOCK, FORKLOCK /FORKUNLOCK, or LOCK/UNLOCK) automatically sets this flag, as long as the code using the macro resides in the same module as the invocation of DPTAB.
DPT$M_XPAMOD	Indicates that the driver can operate on a system with extended physical addressing. When the system is operating with extended addressing, SYSGEN will not load the device driver unless this bit is set.
DPT$M_XVAMOD	Indicates that the driver can operate on a system with extended virtual addressing.

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 1–23 describes the system-defined extensions to the UCB and discusses how a driver defines 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 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, QBDRIVER. (Digital reserves to customers driver names beginning with the letters **J** and **Q**.)

[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 multiprocessing system. Note that use of any of the spinlock 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 identify 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,-
                 SPL$C_IOLOCK8      ;Fork lock index
DPT_STORE   UCB,UCB$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$W_DEVBUFSIZ,W,-
                XA_DEF_BUFSIZ      ;Default buffer size
DPT_STORE  REINIT                  ;Start of reload initialization table
DPT_STORE  DDB,DDB$L_DDT,D,XA$DDT  ;Address of DDT
DPT_STORE  CRB,CRB$L_INTD+VEC$L_ISR,D,-
                XA_INTERRUPT       ;Address of interrupt service routine
DPT_STORE  CRB,CRB$L_INTD+VEC$L_INITIAL,D,-
                XA_CONTROL_INIT    ;Address of controller init routine
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.

DPT_STORE

Instructs the system 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.

exp
Expression indicating the value with which the driver-loading procedure is to initialize the indicated field. If an at sign (@) 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$B_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$C_IOLOCK8
9	SPL$C_IOLOCK9
10	SPL$C_IOLOCK10
11	SPL$C_IOLOCK11

UCB$B_DIPL	Device interrupt priority level.

Other commonly initialized fields are as follows:

UCB$L_DEVCHAR	Device characteristics.
UCB$B_DEVCLASS	Device class.
UCB$B_DEVTYPE	Device type.
UCB$W_DEVBUFSIZ	Default buffer size.
UCB$Q_DEVDEPEND	Device-dependent parameters.

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$L_DDT	Driver dispatch table. Every driver must specify a value for this field.
CRB$L_INTD+ VEC$L_ISR	Interrupt service routine.
CRB$L_INTD2+ VEC$L_ISR	Interrupt service routine for second interrupt vector.
CRB$L_INTD+ VEC$L_INITIAL	Controller initialization routine.
CRB$L_INTD+ VEC$L_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, FORKLOCK /FORKUNLOCK, and LOCK/UNLOCK macros.

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, FORKLOCK /FORKUNLOCK, and LOCK/UNLOCK macros.

$EQULST

Defines a list of symbols and assigns values to the symbols.

Format

$EQULST prefix ,[gbl=LOCAL] ,init ,[incr=1] ,list

Parameters

prefix

Prefix to be used in forming the names of the symbols.

[gbl=LOCAL]

Scope of the definition of the symbol, either **LOCAL**, the default, or **GLOBAL**.

init

Value to be assigned to the first symbol in the list.

[incr=1]

Increment by which to increase the value of each succeeding symbol in the list. The default is 1.

list

List of symbols to be defined. Each element in the list can have one of the following forms:

> <**symbol**> — where **symbol** is the string appended to the prefix, forming the name of the symbol; the value of the symbol is assigned based on the values of **init** and **incr**.
>
> <**symbol**,**value**> — where **symbol** is the string that is appended to the prefix, forming the name of the symbol, and **value** specifies the value (in decimal) of the symbol.

Description

See the descriptions of the $DEFINI and _VIELD macros for additional information on defining symbols for data structure fields.

Example

```
$EQULST  XA_K_,,0,1,<-              ;Define CSR bit values
         <fnct1,2>-
         <fnct2,4>-
         <fnct3,8>-
         <statusa,2048>-
         <statusb,1024>-
         <statusc,512>-
         >
```

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_STATUSC                  = 00000200
```

FIND_CPU_DATA

Locates the start of the per-CPU database area (CPU) for the current process.

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 to calculate the base address of the per-CPU database structure. Use **istack**=**YES** 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.

FORK

Creates a fork process for the context of the code to execute that follows this macro invocation.

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.

Note

To avoid certain race conditions, this macro must be invoked at IPL3 or higher.

FORKLOCK

Achieves synchronized access to a device driver's fork database as appropriate to the processing environment.

Format

FORKLOCK [lock] [,lockipl] [,savipl] [,preserve=YES] [,fipl=NO]

Parameters

[lock]
Index of the fork lock to be obtained. If the **lock** argument is not present in the macro invocation, FORKLOCK presumes that R5 contains the address of the fork block and uses the value at FKB$B_FLCK(R5) as the lock index.

[lockipl]
Location containing the IPL at which the fork database is synchronized. Although the value of this argument is ignored by the macro, Digital recommends that you specify a **lockipl** value to facilitate debugging.

[savipl]
Location at which to save the current IPL.

[preserve=YES]
Indication that the macro should preserve R0 across the invocation. If you do not need to retain the contents of R0, specifying **preserve**=**NO** can enhance system performance.

[fipl=NO]
Indication that the macro does not need to determine whether the contents of the **lock** argument or FKB$B_FLCK(R5) is a fork lock index or a fork IPL. The FORKLOCK macro ignores the contents of this argument in a multiprocessing environment.

The system fork dispatcher uses **fipl**=**YES** to determine whether a fork block it is servicing contains a fork lock index or a fork IPL. Because a device driver initializes offset UCB$B_FLCK (also known as UCB$B_FIPL) in the fork block, it does not need to determine its contents when it issues a FORKLOCK macro.

Description

In a uniprocessing environment, the FORKLOCK macro raises IPL according to one of the following methods:

- It sets IPL to the IPL that corresponds to the fork lock index in the spinlock IPL vector (SMP$AR_IPLVEC).
- If you specify **fipl**=**YES**, the FORKLOCK macro takes the following actions:
  - If offset FKB$B_FLCK (FKB$B_FIPL) contains a fork lock index, it sets IPL to the IPL that corresponds to the fork lock index in the spinlock IPL vector (SMP$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 spinlock 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
     BBSS    #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 system 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 system routine tests a bit in the UCB to determine whether the device is busy. If the device is not busy, the operating system 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.

FORKUNLOCK

Relinquishes synchronized access to a device driver's fork database as appropriate to the processing environment.

Format

FORKUNLOCK [lock] [,newipl] [,condition] [,preserve=YES]

Parameters

[lock]

Index of the fork lock to be released or restored. If **lock** is not present, FORKUNLOCK assumes that R5 contains the address of the fork block and uses the value at FKB$B_FLCK(R5) as the fork lock index.

[newipl]

Location containing the IPL to which to lower. A prior invocation of the FORKLOCK 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 multiprocessing environment—to call SMP$RESTORE instead of SMP$RELEASE. This releases a single acquisition of the fork 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 FORKUNLOCK 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 FORKUNLOCK macro performs the following tasks:

- Preserves R0 through the macro call (if **preserve**=**YES** is specified).
- Stores the fork lock index in R0.
- Calls SMP$RELEASE or, if **condition**=**RESTORE** is specified, SMP$RESTORE.
- 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 FORKUNLOCK macro sets the SMP-modified bit in the driver prologue table (DPT$V_SMPMOD in DPT$L_FLAGS).

For an example of the use of the FORKUNLOCK macro, see the description of the FORKLOCK macro.

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 the operating system 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 the operating system. 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 system 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.

Example

```
XX_FUNCTABLE:                        ;Function decision table
     FUNCTAB   ,-                    ;Valid functions
               <READLBLK,-           ;Read logical block
                READPBLK,-           ;Read physical block
                READVBLK,-           ;Read virtual block
                SENSEMODE,-          ;Sense reader mode
                SENSECHAR,-          ;Sense reader characteristics
                SETMODE,-            ;Set reader mode
                SETCHAR,-            ;Set reader characteristics
               >
     FUNCTAB   ,-                    ;Buffered-I/O functions
               <READLBLK,-           ;Read logical block
                READPBLK,-           ;Read physical block
                READVBLK,-           ;Read virtual block
                SENSEMODE,-          ;Sense reader mode
                SENSECHAR,-          ;Sense reader characteristics
                SETMODE,-            ;Set reader mode
                SETCHAR,-            ;Set reader characteristics
               >
     FUNCTAB   XX_READ,-             ;Read 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. System 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 the operating system, a plus sign (+) precedes its name in the FUNCTAB macro argument.

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 to check memory access; 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.

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    $SS_ACCVIO,R0           ;Assume read access failure
    MOVL      ENTRY_LIST(AP),R11      ;Get address of entry point list
    IFRD      #4*4,(R11),50$          ;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.

INVALIDATE_TB

Allows a single page-table entry (PTE) to be modified while any translation buffer entry that maps it is invalidated, or invalidates the entire translation buffer.

Format

```
INVALIDATE_TB   [addr, inst1 [,inst2] [,inst3] [,inst4] [,inst5] [,inst6] [,save_r2=YES]
                [,checks=YES]]
```

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.

[inst1]

First instruction that modifies the PTE.

[inst2]

Second instruction that modifies the PTE.

[inst3]

Third instruction that modifies the PTE.

[inst4]

Fourth instruction that modifies the PTE.

[inst5]

Fifth instruction that modifies the PTE.

[inst6]

Sixth instruction that modifies the PTE.

[save_r2=YES]

Indication that the value in R2 at the invocation of this macro should be preserved across the macro call. By default, INVALIDATE_TB preserves the value in R2; any value but **YES** supplied in this argument overrides this behavior.

[checks=YES]

Argument enabling or disabling the generation of assembly-time warning messages that indicate misuse of the macro. When any value but **YES** is supplied in the **checks** argument, the INVALIDATE_TB macro does not generate these messages.

Description

When privileged code alters page mapping information, modifying a valid PTE in an active page table, it must notify the operating system. The operating system then takes suitable steps to invalidate all translation buffer entries that reference this PTE.

The INVALIDATE_TB macro allows you modify a single PTE and invalidate a single translation buffer cache entry by supplying the virtual address mapped by the PTE in the **addr** argument and at least one instruction argument. INVALIDATE_TB executes up to six instructions that modify the PTE while preventing all other processors in the system from referencing the page it maps. Because the INVALIDATE_TB macro calls system routines that rely on the stack contents and use R2, none of the specified instruction arguments should reference the stack or use R2.

To invalidate the entire translation buffer (without modifying PTEs), invoke the INVALIDATE_TB macro with no **addr** and instruction arguments. Note that, if the **addr** argument is not present and any instruction arguments are specified, the INVALIDATE_TB macro invalidates the entire translation buffer but does not execute any of the instructions. In this case, if **checks=YES** is not overridden, the macro generates an assembly-time warning message if any instruction arguments are present.

To invoke INVALIDATE_TB, code must be executing at or below IPL$_INVALIDATE, holding—in a multiprocessing environment—no spinlock 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.

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.

IOFORK

Disables timeouts from a target device and creates a fork process for the context of the code to execute that follows this macro invocation.

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

Loads 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.

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$L_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.

LOCK

Achieves synchronized access to a system resource as appropriate to the processing environment.

Format

LOCK lockname [,lockipl] [,savipl] [,condition] [,preserve=YES]

Parameters

lockname
Name of the resource to lock.

[lockipl]
Location containing the IPL at which the resource is synchronized. Although the value of this argument is ignored by the macro, Digital recommends that you specify a **lockipl** value to facilitate debugging.

[savipl]
Location at which to save the current IPL.

[condition]
Indication of a special use of the macro. The only defined **condition** is **NOSETIPL**, which causes the macro to omit setting IPL.

[preserve=YES]
Indication that the macro should preserve R0 across the 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 LOCK macro sets IPL to the IPL that corresponds to the constant IPL$_**lockname**.

In a multiprocessing environment, the LOCK macro performs the following actions:

- Preserves R0 through the macro call (if **preserve**=**YES** is specified).
- Generates a spinlock index of the form SPL$C_**lockname** and stores it in R0.
- Calls SMP$ACQUIRE to obtain the specified spinlock. SMP$ACQUIRE indexes into the system spinlock database (a pointer to this database is located at SMP$AR_SPNLKVEC) to obtain the spinlock. Prior to securing the spinlock, SMP$ACQUIRE raises IPL to the IPL associated with the spinlock, determining the appropriate IPL from the spinlock structure (SPL$B_IPL).

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)

LOCK_SYSTEM_PAGES

Locks a paged code segment in system memory.

Format

LOCK_SYSTEM_PAGES [startva] ,endva [,ipl]

Parameters

[startva]
System virtual address in the first page to be locked. If the **startva** argument is omitted, the starting virtual address defaults to the current PC.

endva
System virtual address in the last page to be locked.

[ipl]
IPL at which the locked code segment is to execute. If the **ipl** argument is omitted, the locked code segment executes at the current IPL.

Description

The LOCK_SYSTEM_PAGES macro calls a memory management routine to lock as many pages as necessary into the system working set. The macro accepts a virtual address that indicates the first page to be locked and a virtual address that indicates the last page to be locked. You can also supply the IPL at which the code in the locked pages is to execute.

The LOCK_SYSTEM_PAGES macro executes under the following conditions:

- The LOCK_SYSTEM_PAGES macro should be used only on system virtual addresses.
- All pages requested in a single LOCK_SYSTEM_PAGES macro call must be virtually contiguous. If you must lock discontiguous memory, you must invoke the LOCK_SYSTEM_PAGES macro once for each page or set of contiguous pages.
- You must invoke LOCK_SYSTEM_PAGES at IPL 2 or lower to allow page faulting to occur.
- When the locked code segment is finished, it must invoke the UNLOCK_SYSTEM_PAGES macro to release all previously locked pages. In other words, there must be exactly one UNLOCK_SYSTEM_PAGES macro call per LOCK_SYSTEM_PAGES macro call.
- When it invokes the UNLOCK_SYSTEM_PAGES macro, the code must ensure that the stack is exactly as it was when the LOCK_SYSTEM_PAGES macro was invoked. That is, if the code has pushed anything on the stack, it must remove it before invoking UNLOCK_SYSTEM_PAGES.
- If the **ipl** argument is supplied to the LOCK_SYSTEM_PAGES macro, the locked code segment must invoke the appropriate system synchronization macros (LOCK, FORKLOCK, or DEVICELOCK and UNLOCK, FORKUNLOCK or DEVICEUNLOCK) to obtain and release any spinlocks required to protect the resources accessed at the elevated IPL.

- If it specified the **ipl** argument to the LOCK_SYSTEM_PAGES macro, the code segment must restore the previous IPL, either explicitly, through the use of the **ipl** argument to the UNLOCK_SYSTEM_PAGES macro, or through the use of one of the system synchronization macros.

Example

```
                TSTB    (R0)                    ; Fault in page
30$:            LOCK_SYSTEM_PAGES,-
                        END=100$                ; Lock down pages
                LOCK    LOCKNAME=MMG,-          ; Synch with MMG
                        SAVIPL=-(SP)            ; Save current IPL
                MOVL    W^MMG$GL_SYSPHD,R3      ; Get system PHD
                .
                .
                .
                UNLOCK  LOCKNAME=MMG,-          ; Unlock MMG
                        NEWIPL=(SP)+            ; Restore IPL
                UNLOCK_SYSTEM_PAGES             ; Unlock pages
100$:
```

In this example, the LOCK_SYSTEM_PAGES macro locks all pages between labels 30$ and 100$ into the system working set. The UNLOCK_SYSTEM_PAGES macro does the coroutine return to unlock those pages locked by the LOCK_SYSTEM_PAGES macro call.

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_CSR

Reads the contents of a device control and status register.

Format

READ_CSR src, dest [,length=LONGWORD] [,error=BUGCHECK]
[,environ=GENERIC] [,vme=pio_reg]

Parameters

src
System virtual address or pseudo CSR address of the register in I/O space.

dest
Location to which the register data is to be returned.

[length=LONGWORD]
Size of the CSR access: BYTE, WORD, or LONGWORD. Default is LONGWORD.

[error=BUGCHECK]
Proper disposition on error. Default is BUGCHECK.

BUGCHECK	Register access failure should result in an UNEXPIOINT bugcheck.
CONTINUE	A status indication should be returned in the low bit of R0: set for success, clear for failure.

[environ=GENERIC]
Specifies how the environment is to be determined. Default is GENERIC.

DRIVER	Test for CRAM access to CSRs is based on bit DEV$M_CRAMIO in location UCB$L_DEVCHAR2. (UCB address must be stored in R5.) This bit is set when the driver is loaded.
GENERIC	Test for CRAM access to CSRs is based on bit ARC$M_CRAMIO in location EXE$GL_ARCHFLAGS. This bit is set during system initialization.
SPECIFIC	CRAM access to CSRs is assumed.

[vme=pio_reg]
Specifies the number of the programmed I/O (PIO) register. If the targeted device resides on a VMEbus, this argument is required.

Description

The READ_CSR macro determines what type of I/O is required for the access, either memory mapped or CRAM (mailbox) I/O, and reads the control register using the appropriate method.

Example

```
10$:    READ_CSR     XMI$L_XDEV(R2), R3
```

This invocation of the READ_CSR macro reads the XMI device type register and returns the value in R3. It assumes that R2 contains the system virtual address or the pseudo CSR address of the base register.

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.

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.

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.

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.

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.

REQCOM

Invokes system 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.

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.

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.

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.

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.

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 multiprocessing environment.

SPI$ABORT_COMMAND

Aborts execution of the outstanding SCSI command on a given connection.

Format

SPI$ABORT_COMMAND

Description

The SPI$ABORT_COMMAND macro aborts the outstanding SCSI command on the connection specified in SCDRP$L_CDT. The SCSI port driver's abort routine sends the SCSI ABORT command to the target device.

Note

VAXstation 3520/3540 systems do not implement the abort-SCSI-command function.

Inputs to the SPI$ABORT_COMMAND macro include the following:

Location	Contents
R4	Address of the SPDT
R5	Address of the SCDRP
SCDRP$L_CDT	Address of the SCDT

The port driver returns SS$_NORMAL status in R0, and preserves the contents of R3, R4, and R5. The original SPI$SEND_COMMAND call completes with SS$_ABORT status.

SPI$ALLOCATE_COMMAND_BUFFER

Allocates a port command buffer for a SCSI command descriptor block.

Format

SPI$ALLOCATE_COMMAND_BUFFER

Description

The SPI$ALLOCATE_COMMAND_BUFFER macro allocates a port command buffer for a SCSI command descriptor block.

Typically a SCSI class driver requests two additional longwords when specifying the size of the requested buffer, the first for the SCSI status byte and the second for the length of the SCSI command. The port command buffer allows the SCSI port driver to access both the SCSI command descriptor block and the SCSI status byte during the SCSI COMMAND and STATUS phases.

Inputs to the SPI$ALLOCATE_COMMAND_BUFFER macro include the following:

Location	Contents
R1	Size of requested buffer. This value should include the size of the SCSI command, plus 4 bytes reserved for the SCSI status byte and 4 bytes in which the SCSI class driver places the size of the SCSI command.
R4	Address of the SPDT.
R5	Address of the SCDRP.
SCDRP$L_CDT	Address of the SCDT.
SCDRP$W_CMD_ MAPREG	Page number of the first port DMA buffer page allocated for the port command buffer.
SCDRP$W_CMD_ NUMREG	Number of port DMA buffer pages allocated for the port DMA buffer.

The port driver returns the following values to the class driver, preserving the contents of R3, R4, and R5:

Location	Contents
R0	SS$_NORMAL
R1	Size of port command buffer
R2	Address of port command buffer

SPI$CONNECT

Creates a connection from a class driver to a SCSI device.

Format

SPI$CONNECT [select_callback [,select_context]]

Parameters

select_callback
Address of a routine in the class driver that executes in response to asynchronous event notification from the target device. The port driver invokes the selection callback routine at this address, holding the fork lock and no other locks at IPL 8; it passes to the routine the address of the SPDT in R4 and any optional selection context in R5.

If the SCSI class driver does not provide a callback address, no selections are allowed on the connection that is established.

select_context
Longword context value to be passed to selection callback routine. When the port driver invokes the selection callback routine, it passes this value to it in R5. For instance, some class drivers may specify the address of the UCB in this argument (**select_context**=**R5**) if the selection callback routine needs access to the device unit's UCB. The **select_context** value can help a class driver that supports multiple device units to identify which unit is generating the asynchronous event.

Description

The SPI$CONNECT macro establishes a connection between the class driver and a SCSI device. It also links a SCSI class driver to the port driver. Before a SCSI class driver can exchange commands and data with a SCSI device, it must invoke SPI$CONNECT.

In response to the call to SPI$CONNECT, the port driver allocates and links an SCDT for the connection. It marks the connection state open and initializes default connection information. If the connection already exists, it returns SS$_DEVALLOC status to the class driver.

Inputs to the SPI$CONNECT macro include the following:

Location	Contents
R1	SCSI device ID (bits <31:16>) and SCSI port ID (bits <15:0>). Valid SCSI device IDs are integers from 0 to 7; valid SCSI port IDs are integers 0 and 1, corresponding to controller IDs A and B.
R2	SCSI logical unit number (bits <31:16>). Bits <15:0> are reserved. Valid SCSI logical unit numbers are integers from 0 to 7.
R4	Address of the SPDT.

Table 2–4 lists the port driver return values to the class driver.

Table 2–4 Values Returned by the SPI$CONNECT Macro

Location	Contents	
R0	Port status. The port driver returns one of the following values:	
	SS$_DEVALLOC	Connection already open for this target.
	SS$_DEVOFFLINE	Port is off line and allows no connections.
	SS$_INSFMEM	Insufficient memory to allocate SCDT.
	SS$_NORMAL	Connection formed.
	SS$_NOSUCHDEV	Port not found.
R1	Maximum byte count allowed (SPDT$L_MAXBYTECNT) for a data transfer.	
R2	Address of the SCDT.	
R3	Port capability mask of SPDT$L_PORT_FLAGS. The following bits are defined by the $SPDTDEF macro (in SYS$LIBRARY:LIB.MLB):	
	SPDT$M_SYNCH	Supports synchronous mode.
	SPDT$M_ASYNCH	Supports asynchronous mode.
	SPDT$M_MAPPING_REG	Supports map registers.
	SPDT$M_BUF_DMA	Supports buffered DMA.
	SPDT$M_DIR_DMA	Supports direct DMA.
	SPDT$M_AEN	Supports asynchronous event notification.
	SPDT$M_LUNS	Supports LUNs (logical unit numbers).
	SPDT$M_CMDQ	Supports SCSI-2 command queuing I/O.
	Bits <25:31>	Contains the recommended byte count divisor for the class driver to derive a proper DMA byte count for the port.
R4	Address of the SPDT.	

The port driver returns the maximum allowed value (SPDT$L_MAXBYTECNT) in R1 to the class driver in response to the class driver's invocation of the SPI$CONNECT macro. Some devices, typically tape drives, need to utilize the full value of SPDT$L_MAXBYTECNT. Most devices, such as disk drives, can better utilize resources with a smaller (suggested) byte count per DMA transfer. The class driver can derive the suggested byte count by utilizing a divisor value in bits <31:25> of the port capability mask (SPDT$L_PORT_FLAGS longword) returned by SPI$CONNECT in R3. For example, if the maximum byte count is 64K and the divisor is 4, then the class driver calculates the suggested byte count as 16K. A sample code sequence (that follows the execution of SPI$CONNECT) for the class driver to calculate the suggested byte count is shown below:

```
;***********************************************************
; After SPI$CONNECT execution, R3 contains divisor value in
; <31:25> and R1 contains MAXBYTECNT

ASHL     #-24,R3,R3       ;Shift divisor value to low-order byte of R3
DIVL3    R3,R1,R0         ;Divide MAXBYTECNT (R1) by divisor (R3) and
                          ;place suggested byte count in R0
```

SPI$DEALLOCATE_COMMAND_BUFFER

Deallocates a port command buffer.

Format

SPI$DEALLOCATE_COMMAND_BUFFER

Description

The SPI$DEALLOCATE_COMMAND_BUFFER macro deallocates a port command buffer.

Inputs to the SPI$DEALLOCATE_COMMAND_BUFFER macro include the following:

Location	Contents
R4	Address of the SPDT.
R5	Address of the SCDRP.
SCDRP$L_CDT	Address of the SCDT.
SCDRP$W_CMD_MAPREG	Page number of the first port DMA buffer page allocated for the port command buffer.
SCDRP$W_CMD_NUMREG	Number of the port DMA buffer pages allocated for the port DMA buffer.

The port driver returns SS$_NORMAL status in R0, and preserves the contents of R3, R4, and R5.

SPI$DISCONNECT

Breaks a connection between a class driver and a SCSI port.

Format

SPI$DISCONNECT

Description

The SPI$DISCONNECT macro breaks a connection between a class driver and a SCSI device unit and deallocates the associated SCDT. The connection must not be busy when it is being disconnected.

Normally a connection between a class driver and a SCSI device unit lasts throughout the runtime life of a system. A SCSI class driver should never need to invoke this macro.

Inputs to the SPI$DISCONNECT macro include the following:

Location	Contents
R1	SCSI device ID (bits <31:16>) and SCSI port ID (bits <15:0>). Valid SCSI device IDs are integers from 0 to 7; valid SCSI port IDs are integers 0 and 1, corresponding to controller IDs A and B.
R2	SCSI logical unit number (bits <15:0>). Valid SCSI logical unit numbers are integers from 0 to 7.
R4	Address of the SPDT.
R5	Address of the SCDT.

The port driver returns SS$_NORMAL status in R0, and preserves the contents of R3, R4, and R5.

SPI$FINISH_COMMAND

Completes an I/O operation initiated with asynchronous event notification.

Format

SPI$FINISH_COMMAND

Description

The SPI$FINISH_COMMAND macro allows the host acting as a target to send a status byte, return the COMMAND COMPLETE message, and drive the SCSI bus to BUS FREE. The class driver's callback routine should invoke SPI$FINISH_COMMAND or SPI$RELEASE_BUS, but not both, before exiting.

The SPI$FINISH_COMMAND function is a higher-level function that class drivers can use to finish an I/O operation that is executing with asynchronous event notification.

Inputs to the SPI$FINISH_COMMAND macro include the following:

Location	Contents
R1	Address of the system buffer containing the SCSI status byte
R4	Address of the SPDT

The port driver returns SS$_NORMAL status in R0, destroys R2, and preserves all other registers.

SPI$GET_CONNECTION_CHAR

Returns characteristics of an existing connection to a specified buffer.

Format

SPI$GET_CONNECTION_CHAR

Description

The SPI$GET_CONNECTION_CHAR macro returns characteristics of an existing connection to a specified buffer.

The buffer format has the characteristics listed in Table 2–5.

Table 2–5 SPI$GET_CONNECTION_CHAR Macro Buffer Characteristics

<table>
<tr><th>Longword</th><th>Contents</th></tr>
<tr><td>1</td><td>Number of longwords in the buffer, not including this longword. The value of this field must be 10.</td></tr>
<tr><td>2</td><td>Connection flags. Bits in this longword are defined as follows:
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>0</td><td>ENA_DISCON. When set, this bit indicates that disconnect and reselection are enabled on this connection.</td></tr>
<tr><td>1</td><td>DIS_RETRY. When set, this bit indicates that command retry is disabled on this connection.</td></tr>
</table></td></tr>
<tr><td>3</td><td>Synchronous. When this longword contains 0, the connection supports asynchronous data transfers; when it contains a nonzero value, the connection supports synchronous data transfers.</td></tr>
<tr><td>4</td><td>Transfer period. If the synchronous parameter is nonzero, this field contains the number of 4-nanosecond ticks between a REQ and an ACK. The default is 64_{10}.</td></tr>
<tr><td>5</td><td>REQ-ACK offset. If the synchronous parameter is nonzero, this field contains the maximum number of REQs outstanding before there must be an ACK.</td></tr>
<tr><td>6</td><td>Busy retry count. Maximum number of retries allowed on this connection while waiting for the bus to become free.</td></tr>
<tr><td>7</td><td>Arbitration retry count. Maximum number of retries allowed on this connection while waiting for the port to win arbitration of the bus.</td></tr>
<tr><td>8</td><td>Select retry count. Maximum number of retries allowed on this connection while waiting for the port to be selected by the target device.</td></tr>
</table>

(continued on next page)

Table 2–5 (Cont.) SPI$GET_CONNECTION_CHAR Macro Buffer Characteristics

Longword	Contents
9	Command retry count. Maximum number of retries allowed on this connection to successfully send a command to the target device.
10	Phase change timeout. Default timeout value (in seconds) for a target to change the SCSI bus phase or complete a data transfer. This value is also known as the DMA timeout. Upon sending the last command byte, the port driver waits this many seconds for the target to change the bus phase lines and assert REQ (indicating a new phase). Or, if the target enters the DATA IN or DATA OUT phase, the transfer must be completed within this interval. If this value is not specified, the default value is 4 seconds.
11	Disconnect timeout. Default timeout value (in seconds) for a target to reselect the initiator to proceed with a disconnected I/O transfer. If this value is not specified, the default value is 4 seconds.
12	SCSI-2 device characteristic status bits. Bits of this longword are defined as follows:

Bit	Description
0	When set, (SCDT$V_SCSI_2) indicates the device connection is SCSI-2 conformant.
1	When set, (SCDT$V_CMDQ) indicates the device connection supports command queuing.

Inputs to the SPI$GET_CONNECTION_CHAR macro include the following:

Location	Contents
R2	Address of the connection characteristics buffer.
R4	Address of the SPDT.
R5	Address of the SCDRP.
SCDRP$L_CDT	Address of the SCDT.

The port driver returns the following values to the class driver, preserving R3, R4, and R5:

Location	Contents
R0	Port status. The port driver returns one of the following values:

SS$_NORMAL	Normal, successful completion
SS$_NOSUCHID	No connection for this SCSI connection ID

Location	Contents
R2	Address of the connection characteristics buffer in which device characteristics are returned.

SPI$MAP_BUFFER

Makes the process buffer involved in a data transfer available to the port driver.

Format

SPI$MAP_BUFFER [prio=HIGH]

Parameters

prio=HIGH
If **prio=HIGH** is specified, deadlocks (that can be incurred when several devices are mapping buffers) are avoided. If the argument is not specified, the macro defaults to LOW priority.

Description

The SPI$MAP_BUFFER macro makes the process buffer involved in a data transfer accessible to the port driver. Typically, the I/O buffer is specified in the $QIO call, is in process space (P0 space), and is mapped by process page-table entries. Because a port driver executes in system context, it cannot access a process's page table.

The means by which the SPI$MAP_BUFFER macro makes a process buffer available to the port driver depends upon the port hardware. For certain implementations, it allocates a segment of the port's DMA buffer and a set of system page-table entries that double-map the process buffer. In others, it obtains a set of port map registers and loads them with the page-frame numbers of the process buffer pages.

Table 2–6 lists the inputs to the SPI$MAP_BUFFER macro.

Table 2–6 Inputs to the SPI$MAP_BUFFER Macro

Location	Contents	
R4	Address of the SPDT.	
R5	Address of the SCDRP. The class driver must provide values in the following fields:	
	SCDRP$L_BCNT	Size in bytes of the buffer to be mapped. The largest single transfer that can be mapped is determined by the port driver in the call to SPI$CONNECT. The SPI$CONNECT macro returns this value to the class driver in R1. If the class driver must accomplish transfers larger than this value, it must segment them.
	SCDRP$W_BOFF	Byte offset into the first page of the buffer.

(continued on next page)

Table 2–6 (Cont.) Inputs to the SPI$MAP_BUFFER Macro

Location	Contents	
	SCDRP$L_SVA_USER	For direct DMA buffering, system virtual address of the process buffer to map in system space (S0 space)
	SCDRP$L_SVAPTE	System virtual address of the page-table entry that maps the first byte of the user buffer.
	SCDRP$L_SCSI_ FLAGS	SCSI mapping flags. If SCDRP$V_ S0BUF is set, SPI$MAP_BUFFER does not double-map the buffer into system space.
	SCDRP$W_STS	Transfer direction flags. IRP$V_ FUNC must be set for read I/O functions and clear for write I/O functions.

The port driver returns the values listed in Table 2–7 to the class driver, preserving R3, R4, and R5.

Table 2–7 SPI$MAP_BUFFER Macro Return Values to the Class Driver

Location	Contents	
R0	Port status. The port driver returns one of the following values:	
	SS$_NORMAL	Normal, successful completion
	SS$_BADPARAM	Bad parameter provided by class driver
R5	Address of the SCDRP. The port driver initializes the following fields:	
	SCDRP$L_SVA_USER	System virtual address of the process buffer as mapped in system space (S0 space)
	SCDRP$L_SVA_SPTE	System virtual address of the system page-table entry that maps the first page of the process buffer in S0 space
	SCDRP$W_NUMREG	Number of port DMA buffer pages allocated
	SCDRP$W_MAPREG	Page number of the first port DMA buffer page allocated

SPI$QUEUE_COMMAND

Initiates a new I/O to the port driver for queued SCSI-2 command tagged requests.

Format

SPI$QUEUE_COMMAND

Description

The SPI$QUEUE_COMMAND initiates a new I/O to the port driver for queued SCSI-2 command tagged requests. This macro is similar to the SPI$SEND_COMMAND, but SPI$QUEUE_COMMAND does not wait for command completion in the device before returning to the class driver.

A class driver first calls SPI$ALLOCATE_COMMAND_BUFFER to allocate a port command buffer and then formats a SCSI command descriptor block in the buffer before invoking this macro. The class driver may need to call SPI$MAP_BUFFER to allocate and map user buffer resources. To execute a burst of I/O requests, the class driver may call SPI$QUEUE_COMMAND for each request without waiting for any of these I/Os to complete. Each request must use a different SCDRP and separately allocate the needed resources. When the I/O request completes, the port driver then returns SCSI status to the class driver.

Table 2–8 lists the inputs to the SPI$QUEUE_COMMAND macro.

Table 2–8 Inputs to the SPI$QUEUE_COMMAND Macro

Location	Contents	
R0	Queue characteristics (constants, QCHAR$K_*xxx*).	
R3	Address of the UCB.	
R4	Address of the SPDT.	
R5	Address of the SCDRP. The class driver must provide values in the following fields:	
	SCDRP$L_CMD_PTR	Address of the port command buffer. The first longword of the port command buffer contains the number of bytes in the buffer, not including the count longword. Subsequent bytes contain the SCSI command descriptor block.
	SCDRP$L_BCNT	Size in bytes of the mapped process buffer.
	SCDRP$W_PAD_BCNT	Number of bytes to make the size of the buffer equal to the data length value required in the command.

(continued on next page)

Table 2–8 (Cont.) Inputs to the SPI$QUEUE_COMMAND Macro

Location	Contents	
	SCDRP$L_SVA_USER	System virtual address of the process buffer as mapped in system space (S0 space).
	SCDRP$L_STS_PTR	Address of the status longword. The port driver copies the SCSI status byte it receives in the bus STATUS phase into the low-order byte of this buffer.
	SCDRP$W_FUNC	Read or write operation.
	SCDRP$L_SCDT	Address of the SCDT.

The port driver returns the values listed in Table 2–9 to the class driver, preserving R3, R4, and R5.

Table 2–9 SPI$QUEUE_COMMAND Macro Return Values

Location	Contents	
R0	Port status. The port driver returns one of the following status values:	
	SS$_BADPARAM	Bad parameter specified by the class driver.
	SS$_CTRLERR	Controller error or port hardware failure.
	SS$_DEVACTIVE	Command outstanding on this connection.
	SS$_DEVREQERR	SCSI message reject.
	SS$_IVSTSFLG	All required phases are not entered.
	SS$_LINKABORT	Connection no longer exists.
	SS$_NORMAL	Normal, successful completion.
	SS$_TIMEOUT	Failed during selection or arbitration.
	SS$_PARITY	Nonrecoverable parity error detected.
R5	Address of the SCDRP. The port driver provides information in the following fields:	
	SCDRP$L_STS_PTR	Address of the status longword. The port driver copies the SCSI status byte it receives in the bus STATUS phase into the low-order byte of this buffer.
	SCDRP$L_TRANS_CNT	Actual number of bytes sent or received by the port driver during the data phase.

SPI$RECEIVE_BYTES

Receives command, message, and data bytes from a device acting as an initiator on the SCSI bus.

Format

SPI$RECEIVE_BYTES

Description

The SPI$RECEIVE_BYTES macro allows the host to receive information from the device acting as an initiator. A class driver uses SPI$RECEIVE_BYTES to receive command, message, and data bytes. This macro uses DMA operations for the transfer of large segments of data where appropriate.

Inputs to the SPI$RECEIVE_BYTES macro include the following:

Location	Contents
R0	Size of the system buffer into which the target returns the requested bytes
R1	Address of the system buffer into which the target device returns the requested bytes
R4	Address of the SPDT

The port driver returns the following values to the class driver, destroying R2, and preserving all other registers:

Location	Contents	
R0	Port status. The port driver returns one of the following values:	
	SS$_NORMAL	Normal, successful completion.
	SS$_CTRLERR	Timeout occurred during the operation.
R1	Actual number of bytes received.	

SPI$RELEASE_BUS

Releases the SCSI bus.

Format

SPI$RELEASE_BUS

Description

The SPI$RELEASE_BUS macro allows the host acting as a target to release the SCSI bus. The class driver's callback routine should invoke either SPI$RELEASE_BUS or SPI$FINISH_COMMAND, but not both, before exiting.

The class driver should use SPI$RELEASE_BUS instead of SPI$FINISH_COMMAND if it must explicitly send the SCSI status byte and COMMAND COMPLETE message using SPI$SEND_BYTES, or if it simply wants to drop off the bus and terminate the thread in certain error conditions.

Inputs to the SPI$RELEASE_BUS macro include the following:

Location	Contents
R4	Address of the SPDT

The port driver returns SS$_NORMAL status in R0, destroys R2, and preserves all other registers.

SPI$RELEASE_QUEUE

Clears the frozen state of the SCSI-2 port driver queues.

Format

SPI$RELEASE_QUEUE

Description

The SPI$RELEASE_QUEUE macro clears the frozen state of the port driver queues allowing queue processing to resume and new I/O to be processed.

The entry point routine in the port driver clears the SCDT$V_QUEUE_FROZEN bit in the SCDT$L_QUEUE_FLAGS longword mask. This bit is checked by the queue manager to signal when to resume queue processing.

Inputs to the SPI$RELEASE_QUEUE macro include the following:

Location	Contents
R3	Address of the UCB
R4	Address of the SPDT
R5	Address of the SCDRP

The port driver returns SS$_NORMAL status in R0, and preserves the contents of R3, R4, and R5.

SPI$RESET

Resets the SCSI bus and SCSI port hardware.

Format

SPI$RESET

Description

The SPI$RESET macro first resets the SCSI bus and then resets the port hardware. A SCSI class driver should rarely invoke this macro; those class drivers that do use it should be aware of the impact of a reset operation on other devices on the same bus. The SCSI port driver logs an error when a class driver invokes the SPI$RESET macro.

Inputs to the SPI$RESET macro include the following:

Location	Contents
R4	Address of the SPDT.
R5	Address of the SCDRP.
SCDRP$L_CDT	Address of the SCDT.

The port driver returns the following value to the class driver, preserving R3, R4, and R5:

Location	Contents	
R0	Port status. The port driver returns one of the following values:	
	SS$_NORMAL	Normal, successful completion.
	SS$_ABORT	Reset aborted before completion.

SPI$SEND_BYTES

Sends command, message, and data bytes to a device acting as an initiator on the SCSI bus.

Format

SPI$SEND_BYTES

Description

The SPI$SEND_BYTES macro allows the host to send information to the device acting as an initiator. A class driver uses SPI$SEND_BYTES to send command, message, and data bytes. This macro uses DMA operations for the transfer of large segments of data where appropriate.

Inputs to the SPI$SEND_BYTES macro include the following:

Location	Contents
R0	Size of the system buffer that contains the bytes to be sent
R1	Address of the system buffer that contains the bytes to be sent
R4	Address of the SPDT

The port driver returns the following values to the class driver, destroying R2, and preserving all other registers:

Location	Contents	
R0	Port status. The port driver returns one of the following values:	
	SS$_NORMAL	Normal, successful completion.
	SS$_CTRLERR	Timeout occurred during the operation.
R1	Actual number of bytes sent.	

SPI$SEND_COMMAND

Sends a command to a SCSI device.

Format

SPI$SEND_COMMAND

Description

The SPI$SEND_COMMAND macro sends a command to a SCSI device. A class driver invokes this macro, after calling SPI$ALLOCATE_COMMAND_BUFFER to allocate a port command buffer and formatting a SCSI command descriptor block in it.

The port driver responds to the SPI$SEND_COMMAND macro call by arbitrating for access to the SCSI bus, selecting the target device, sending the SCSI command descriptor block to the target, and waiting for a response. Before returning to the class driver, the port driver sends data to or receives data from the target device, obtains command status, processes SCSI message bytes, and transfers the data. When the SPI$SEND_COMMAND routine completes, the port driver then returns port status and SCSI status to the class driver.

Table 2–10 lists the inputs to the SPI$SEND_COMMAND macro.

Table 2–10 Inputs to the SPI$SEND_COMMAND Macro

Location	Contents	
R3	Address of the UCB.	
R4	Address of the SPDT.	
R5	Address of the SCDRP. The class driver must provide values in the following fields:	
	SCDRP$L_CMD_PTR	Address of the port command buffer. The first longword of the port command buffer contains the number of bytes in the buffer (not including the count longword). Subsequent bytes contain the SCSI command descriptor block.
	SCDRP$L_BCNT	Size in bytes of the mapped process buffer.
	SCDRP$W_PAD_BCNT	Number of bytes to make the size of the buffer equal to the data length value required in the command.

(continued on next page)

Table 2–10 (Cont.) Inputs to the SPI$SEND_COMMAND Macro

Location	Contents	
	SCDRP$L_SVA_USER	System virtual address of the process buffer as mapped in system space (S0 space).
	SCDRP$L_STS_PTR	Address of the status longword. The port driver copies the SCSI status byte it receives in the bus STATUS phase into the low-order byte of this buffer.
	SCDRP$W_FUNC	Read or write operation.
SCDRP$L_CDT	Address of the SCDT.	

The port driver returns the values listed in Table 2–11 to the class driver, preserving R3, R4, and R5.

Table 2–11 SPI$SEND_COMMAND Macro Return Values

Location	Contents	
R0	Port status. The port driver returns one of the following status values:	
	SS$_BADPARAM	Bad parameter specified by the class driver.
	SS$_CTRLERR	Controller error or port hardware failure.
	SS$_DEVACTIVE	Command outstanding on this connection.
	SS$_LINKABORT	Connection no longer exists.
	SS$_NORMAL	Normal, successful completion.
	SS$_TIMEOUT	Failed during selection or arbitration.
R5	Address of the SCDRP. The port driver provides information in the following fields:	
	SCDRP$L_STS_PTR	Address of the status longword. The port driver copies the SCSI status byte it receives in the bus STATUS phase into the low-order byte of this buffer.
	SCDRP$L_TRANS_ CNT	Actual number of bytes sent or received by the port driver during the Data phase.

SPI$SENSE_PHASE

Returns the current phase of the SCSI bus.

Format

SPI$SENSE_PHASE

Description

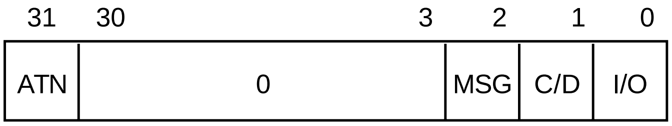
The SPI$SENSE_PHASE macro allows the host to read the current SCSI bus phase, and the state of the ATN signal, while using the asynchronous event notification feature.

A class driver must supply the address of the SPDT in R4 as input to the SPI$SENSE_PHASE macro.

The port driver returns the following values to the class driver, destroying R2, and preserving all other registers:

Location	**Contents**
R0	SS$_NORMAL.
R1	SCSI bus phase (and ATN signal). This SCSI-defined longword has the format illustrated in Figure 2–1.

Figure 2–1 SCSI Bus Phase Longword Returned to SPI$SENSE_PHASE



ZK−1377A−GE

SPI$SET_CONNECTION_CHAR

Sets characteristics of an existing connection.

Format

SPI$SET_CONNECTION_CHAR

Description

The SPI$SET_CONNECTION_CHAR macro sets characteristics of an existing SCSI connection. Prior to altering the characteristics of a connection, a SCSI class driver should read and examine the current connection characteristics using the SPI$GET_CONNECTION_CHAR macro.

The class driver specifies the characteristics to be set for the connection in a connection characteristics buffer. The buffer has the format listed in Table 2–12.

Table 2–12 SPI$SET_CONNECTION_CHAR Macro Settable Characteristics

<table>
<tr><th>Longword</th><th>Contents</th></tr>
<tr><td>1</td><td>Number of longwords in the buffer, not including this longword. The value of this field must be 10.</td></tr>
<tr><td>2</td><td>Connection flags. Bits in this longword are defined as follows:
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>0</td><td>ENA_DISCON. When set, this bit enables disconnect and reselection on the connection.</td></tr>
<tr><td>1</td><td>DIS_RETRY. When set, this bit disables command retry on the connection.</td></tr>
</table></td></tr>
<tr><td>3</td><td>Synchronous. When this longword contains 0, the connection uses asynchronous data transfer mode; when it contains a nonzero value, the connection uses synchronous data transfer mode.</td></tr>
<tr><td>4</td><td>Transfer period. If the synchronous parameter is nonzero, this field controls the number of 4-nanosecond ticks between a REQ and an ACK. The default is 64_{10}.</td></tr>
<tr><td>5</td><td>REQ-ACK offset. If the synchronous parameter is nonzero, this field controls the maximum number of REQs outstanding before there must be an ACK.</td></tr>
<tr><td>6</td><td>Busy retry count. Maximum number of retries allowed on this connection while waiting for the port to become free.</td></tr>
<tr><td>7</td><td>Arbitration retry count. Maximum number of retries allowed on this connection while waiting for the port to win arbitration of the bus.</td></tr>
</table>

(continued on next page)

Table 2–12 (Cont.) SPI$SET_CONNECTION_CHAR Macro Settable Characteristics

Longword	Contents
8	Select retry count. Maximum number of retries allowed on this connection while waiting for the port to be selected by the target device.
9	Command retry count. Maximum number of retries allowed on this connection to successfully send a command to the target device.
10	Phase change timeout. Default timeout value (in seconds) for a target to change the SCSI bus phase or complete a data transfer. This value is also known as the DMA timeout. Upon sending the last command byte, the port driver waits this many seconds for the target to change the bus phase lines and assert REQ (indicating a new phase). Or, if the target enters the DATA IN or DATA OUT phase, the transfer must be completed within this interval. If this value is not specified, the default value is 4 seconds.
11	Disconnect timeout. Default timeout value (in seconds) for a target to reselect the initiator to proceed with a disconnected I/O transfer. If this value is not specified, the default value is 4 seconds.
12	SCSI-2 device characteristic status bits. Bits of this longword are defined as follows: **Bit** — **Description** 0 — When set, (SCDT$V_SCSI_2) indicates the device connection is SCSI-2 conformant. 1 — When set, (SCDT$V_CMDQ) indicates the device connection supports command queuing.

Inputs to the SPI$SET_CONNECTION_CHAR macro include the following:

Location	Contents
R2	Address of the connection characteristics buffer.
R4	Address of the SPDT.
R5	Address of the SCDRP.
SCDRP$L_CDT	Address of the SCDT.

The port driver returns the following values to the class driver, preserving R3, R4, and R5:

Location	Contents
R0	Port status. The port driver returns one of the following values: SS$_NORMAL Normal, successful completion SS$_NOSUCHID No connection for this SCSI connection ID

SPI$SET_PHASE

Sets the bus to a new phase.

Format

SPI$SET_PHASE

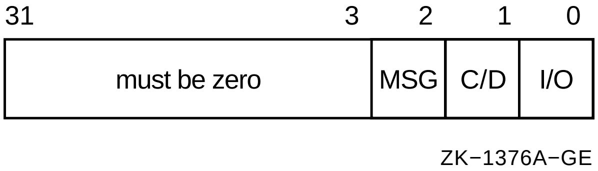
Description

The SPI$SET_PHASE macro allows the host to set the SCSI bus to a new phase. A class driver uses this macro to drive the phase transitions of the SCSI bus while using the asynchronous event notification feature.

Inputs to the SPI$SET_PHASE macro include the following:

Location	Contents
R0	New SCSI bus phase. This SCSI-defined longword has the format shown in Figure 2–2.
R4	Address of the SPDT.

Figure 2–2 SCSI Bus Phase Longword Supplied to SPI$SET_PHASE



The port driver returns SS$_NORMAL status in R0, destroys R2, and preserves all other registers.

SPI$UNMAP_BUFFER

Releases port mapping resources and deallocates port DMA buffer space, as required to unmap a process buffer.

Format

SPI$UNMAP_BUFFER

Description

The SPI$UNMAP_BUFFER macro releases mapping resources and deallocates port DMA buffer space, as required to unmap a process buffer.

Inputs to the SPI$UNMAP_BUFFER macro include the following:

Location	Contents	
R4	Address of the SPDT.	
R5	Address of the SCDRP. The class driver must provide values in the following fields:	
	SCDRP$W_NUMREG	Number of port DMA buffer pages allocated
	SCDRP$W_MAPREG	Page number of the first port DMA buffer page

The port driver returns the following values to the class driver, preserving R3, R4, and R5:

Location	Contents
R0	SS$_NORMAL.
R5	Address of the SCDRP. The port driver clears SCDRP$W_NUMREG and SCDRP$W_MAPREG.

SWAPLONG

Swaps the bytes within each longword supplied.

Format

SWAPLONG longword

Parameters

longword
The address of the longword data that requires the bytes to be swapped.

Description

When a data word is passed between a host CPU and a device with a differing byte-order pattern (big-endian and little-endian devices), the byte positions must be swapped. The SWAPLONG macro reads the location of the 4-byte data supplied in the longword argument and modifies the byte positions to a mirrored order.

SWAPWORD

Swaps the bytes within each word supplied.

Format

SWAPWORD word

Parameters

word
The address of the data (2 bytes) that requires the bytes to be swapped.

Description

When a data word is passed between a host CPU and a device with a differing byte-order pattern (big-endian and little-endian devices), the byte positions must be swapped. The SWAPWORD macro reads the location of the 2-byte data supplied in the word argument and swaps the byte positions.

TIMEDWAIT

Waits a specified interval of time for an event or condition to occur.

Format

TIMEDWAIT time [,ins1] [,ins2] [,ins3] [,ins4] [,ins5] [,ins6] [,donelbl] [,imbedlbl] [,ublbl]

Parameters

time
Number of 10-microsecond intervals to wait. The operating system 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.

[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 of time it waits is approximate and depends on 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 ready?
          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.

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. The operating system 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.

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 multiprocessing environment—to call SMP$RESTORE instead of SMP$RELEASE, thus releasing a single acquisition of the spinlock 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 spinlock index of the format SPL$C_**lockname** and stores it in R0.
- Calls SMP$RELEASE or, if **condition**=**RESTORE** is specified, SMP$RESTORE. These routines index into the system spinlock database (a pointer to which is located at SMP$AR_SPNLKVEC) to release the appropriate spinlock.
- 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$V_SMPMOD in DPT$L_FLAGS).

UNLOCK_SYSTEM_PAGES

Terminates a request to lock down a series of system pages.

Format

UNLOCK_SYSTEM_PAGES [ipl]

Parameters

[ipl]
IPL at which to continue execution.

Description

The UNLOCK_SYSTEM_PAGES macro terminates a request to lock down a series of contiguous system pages. In a code segment that uses this locking technique, there must be exactly one UNLOCK_SYSTEM_PAGES macro call per LOCK_SYSTEM_PAGES macro call. When the locked code segment completes, it must invoke the UNLOCK_SYSTEM_PAGES macro to release all previously locked pages.

The UNLOCK_SYSTEM_PAGES macro executes under the following conditions:

- When it invokes the UNLOCK_SYSTEM_PAGES macro, the code must ensure that the stack is exactly as it was when the LOCK_SYSTEM_PAGES macro was invoked. That is, if the code has pushed anything on the stack, it must remove it before invoking UNLOCK_SYSTEM_PAGES.
- If it specified the **ipl** argument to the LOCK_SYSTEM_PAGES macro, the code segment must restore the previous IPL, either explicity, through the use of the **ipl** argument to the UNLOCK_SYSTEM_PAGES macro, or through the use of one of the system synchronization macros (UNLOCK, FORKUNLOCK or DEVICEUNLOCK). If it lowers IPL, the locked code segment must invoke the appropriate system synchronization macro to release any spinlocks that were required to protect the resources accessed at the elevated IPL.

$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.

$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_] [,size=_LENGTH]

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.

[,size]
Number of bytes allocated for the vector table.

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.

$VIELD, _VIELD

Defines symbolic offsets and masks for bit fields.

Format

$$\left\{ \begin{array}{l} \text{\$VIELD} \\ \text{\_VIELD} \end{array} \right\} \text{ mod ,inibit ,fields}$$

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 system-defined symbols, use of the _VIELD macro is recommended for non-Digital-supplied device drivers.

See the descriptions of the $DEFINI and $EQULST macros for additional information on defining symbols for data structure fields.

Example

```
$EQULST  XA_K_,,0,1,<-              ;Define CSR bit values
         <fnct1,2>-
         <fnct2,4>-
         <fnct3,8>-
_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
```

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

$\left\{\begin{array}{l}\text{WFIKPCH}\\\text{WFIRLCH}\end{array}\right\}$ excpt [,time=65536]

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

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 **excpt** 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.

WRITE_CSR

Writes data to a device control and status register.

Format

WRITE_CSR src, dest [,length=LONGWORD] [,error=BUGCHECK]
[,environ=GENERIC] [,vme=pio_reg]

Parameters

src
Location containing the data to be written to the register.

dest
System virtual address or pseudo CSR address of the register in I/O space.

[length=LONGWORD]
Size of the CSR access: BYTE, WORD, or LONGWORD. Default is LONGWORD.

[error=BUGCHECK]
Proper disposition on error. Default is BUGCHECK.

BUGCHECK	Register access failure should result in an UNEXPIOINT bug check.
CONTINUE	A status indication should be returned in the low bit of R0: set for success, clear for failure.

[environ=GENERIC]
Specifies how the environment is to be determined. Default is GENERIC.

DRIVER	Test for CRAM access to CSRs is based on bit DEV$M_CRAMIO in location UCB$L_DEVCHAR2. (UCB address must be stored in R5.) This bit is set when the driver is loaded.
GENERIC	Test for CRAM access to CSRs is based on bit ARC$M_CRAMIO in location EXE$GL_ARCHFLAGS. This bit is set during system initialization.
SPECIFIC	CRAM access to CSRs is assumed.

[vme=pio_reg]
Specifies the number of the programmed I/O (PIO) register. If the targeted device resides on a VMEbus, this argument is required.

Description

The WRITE_CSR macro determines what type of I/O is required for the access, either memory mapped or CRAM (mailbox) I/O, and writes the control register using the appropriate method.

Example

```
10$:    WRITE_CSR     #XMI$M_NRESET, XMI$L_XBE(R5)
```

This invocation of the WRITE_CSR macro writes the reset bit to the XBE register of the XMI.

3

Operating System Routines

This chapter describes the operating system routines that are used by device drivers and employs the following conventions:

- Most routines reside in modules within the [SYS] facility of the operating system. 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, the operating system 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. Chapter 2 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 *OpenVMS System Services Reference Manual* and the *OpenVMS System Messages and Recovery Procedures Reference Manual*.
- 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.

BYTE_SWAP_LONG

Swaps the bytes within each longword in a given data transfer buffer.

Module

[DRIVER]VME_SUPPORT

Input

Location	**Contents**
R0	Length of the data transfer buffer in bytes. This number should fall on a longword boundary.
R1	Address of the data transfer buffer.

Output

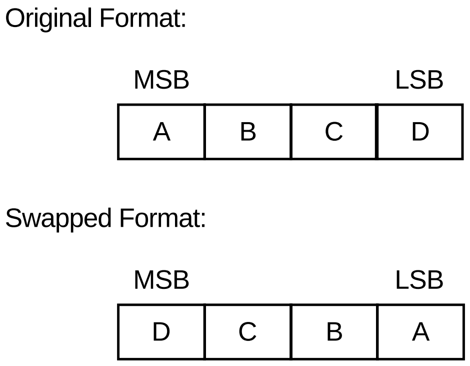
Location	**Contents**
R0, R1	Destroyed
	(All other registers preserved)

Synchronization

A driver calls BYTE_SWAP_LONG in kernel mode at or above IPL$_ASTDEL.

Description

BYTE_SWAP_LONG swaps the bytes within each longword of a given data transfer. The data is read from an input system buffer, then the byte positions of each longword are modified to a mirrored order, swapping the least significant bytes (LSB) with the most significant bytes (MSB), as shown in the following figure.



ZK−3733A−GE

Note that if the buffer byte-length is not an exact number of longwords, the bytes in the last incomplete longword are unaffected.

BYTE_SWAP_WORD

Swaps the bytes within each word in a given data transfer buffer.

Module

[DRIVER]VME_SUPPORT

Input

Location	**Contents**
R0	Length of the data transfer buffer in bytes. This number should fall on a word boundary.
R1	Address of the data transfer buffer.

Output

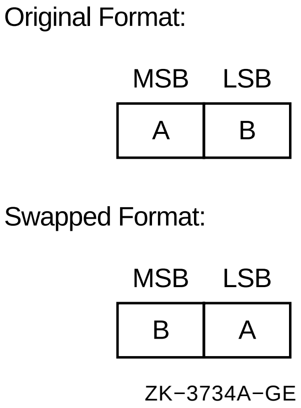
Location	**Contents**
R0, R1	Destroyed
	(All other registers preserved)

Synchronization

A driver calls BYTE_SWAP_WORD in kernel mode at or above IPL$_ASTDEL.

Description

BYTE_SWAP_WORD swaps the bytes within each word of a given data transfer. The data is read from an input system buffer, then the byte positions of each word are modified to a mirrored order, swapping the least significant byte (LSB) with the most significant byte (MSB), as shown in the following figure.



Note that if the buffer contains an odd number of bytes, the last byte in the incomplete word at the end of the buffer is unaffected.

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 spinlocks. However, the caller must be executing at IPL3 or higher to avoid certain race conditions.

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 spinlocks it held at the time of the call. If called at IPL$_SYNCH or higher, the routine executes the fork process.

Description

COM$DRVDEALMEM calls EXE$DEANONPAGED to deallocate the buffer specified by R0. If COM$DRVDEALMEM cannot deallocate memory at the caller's IPL, it transforms the block being deallocated into a fork block and queues the block in the fork queue. The code that executes in the fork process then jumps to EXE$DEANONPAGED.

If the buffer to be deallocated is less than FKB$C_LENGTH in size, or its address is not aligned on a 16-byte boundary, COM$DRVDEALMEM issues a BADDALRQSZ bugcheck.

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$L_DLCK	Address of device lock
PCB$L_PID	Process ID
PCB$W_ASTCNT	ASTs remaining in quota

Output

Location	**Contents**
R0	SS$_NORMAL
R1, R2, R7	Destroyed
PCB$W_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 spinlocks 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$POST, COM$POST_NOCNT

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 (COM$POST only)
IRP$L_MEDIA	Data to be copied to the I/O status block
IRP$L_MEDIA+4	Data to be copied to the I/O status block

Output

Location	**Contents**
R0	Destroyed
UCB$L_OPCNT	Incremented (COM$POST only)

Synchronization

Drivers call COM$POST and COM$POST_NOCNT at or above fork IPL. These routines execute at their callers' IPL and return control at that IPL. The caller retains any spinlocks it held at the time of the call.

Description

A driver fork process calls COM$POST or COM$POST_NOCNT after it has completed device-dependent I/O processing for an I/O request initiated by EXE$ALTQUEPKT. Because COM$POST_NOCNT, unlike COM$POST, does not increment the unit's operations count (UCB$L_OPCNT), a driver uses COM$POST_NOCNT to initiate completion processing for an I/O request when the associated UCB is not available.

COM$POST and COM$POST_NOCNT insert the IRP into the systemwide I/O postprocessing queue, request an IPL$_IOPOST software interrupt, and return control to the caller. Unlike IOC$REQCOM, these routines do not attempt to dequeue any IRP waiting for the device or change the busy status of the device.

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$_INSFMEM
R1 and R2	Destroyed
R3	Address of IRP
R5	Address of UCB
R6, R7, R8	Destroyed
PCB$W_ASTCNT	Decremented
Specified listhead	Updated

Synchronization

COM$SETATTNAST must be called from code executing at IPL$_ASTDEL. COM$SETATTNASTIPL acquires the corresponding device lock inserting the AST into the AST queue. It returns control to the caller at IPL$_ASTDEL.

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.

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:

- Spinlock 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$SETATTNAST links the ACB to the start of the specified linked list of ACBs located in a UCB extension area. (See Section 1.19 for information on defining an extension to a UCB.) COM$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.

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 multiprocessing environment. These routines return control to the caller at the caller's IPL. The caller retains any spinlocks 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 spinlock.

- 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$DEVICERR	Device error (EMB$C_DE)
ERL$DEVICTMO	Device timeout (EMB$C_DT)
ERL$DEVICEATTN	Device attention (EMB$C_DA)

- Places the address of the error message buffer in UCB$L_EMB.
- Sets UCB$V_ERLOGIP in UCB$L_STS.
- Loads fields from the UCB, the IRP, and the DDB into the buffer, including the following:

UCB$B_DEVCLASS	Device class
UCB$B_DEVTYPE	Device type
IRP$L_PID	Process ID of the process originating the I/O request (ERL$_DEVICERR and ERL$_DEVICTMO)
IRP$W_BOFF	Transfer parameter (ERL$DEVICERR and ERL$DEVICTMO)
IRP$W_BCNT	Transfer parameter (ERL$DEVICERR and ERL$DEVICTMO)
UCB$L_MEDIA	Disk size
UCB$W_UNIT	Unit number
UCB$W_ERRCNT	Count of device errors
UCB$L_OPCNT	Count of completed operations
ORB$L_OWNER	UIC of volume owner
UCB$L_DEVCHAR	Device characteristics
UCB$B_SLAVE	Slave unit number
IRP$W_FUNC	I/O function value (ERL$DEVICERR and ERL$DEVICTMO)
DDB$T_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 1.19, to use these error-logging routines.

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 ACB$V_QUOTA was set

Synchronization

EXE$ABORTIO executes at its caller's IPL and raises to fork IPL, acquiring the associated fork lock in a multiprocessing environment. As a result, its caller cannot be executing above fork IPL. A driver usually transfers control to EXE$ABORTIO at IPL$_ASTDEL.

EXE$ABORTIO exits at normal process IPL (IPL 0).

Description

EXE$ABORTIO performs the following actions:

1. Clears IRP$L_IOSB so that no status is returned by I/O postprocessing
2. Clears ACB$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
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.

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.

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$C_LENGTH for EXE$ALLOCIRP).
R2	Address of allocated buffer.
R4	See the following discussion.
IRP$W_SIZE (in allocated buffer)	Size of requested buffer in bytes (for EXE$ALLOCBUF), IRP$C_LENGTH (for EXE$ALLOCIRP).
IRP$B_TYPE (in allocated buffer)	DYN$C_BUFIO (for EXE$ALLOCBUF), DYN$C_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 the caller at IPL$_ASTDEL.

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$L_BYTCNT or JIB$L_BYTLM, or both) 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.)

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.

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	Size of the allocated block, which may be larger than the requested size
R2	Address of allocated block

Synchronization

EXE$ALONONPAGED executes at its caller's IPL and at IPL$_POOL, obtaining the POOL spinlock in a multiprocessing environment. Thus, if a packet cannot be obtained from one of the lookaside lists, callers at IPL greater than IPL$_POOL are required to fork to a lower IPL and retry in order to successfully obtain a packet.

EXE$ALONONPAGED returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

Depending on the size of the requested block, EXE$ALONONPAGED allocates nonpaged pool from either a lookaside list or from the variable region of nonpaged dynamic memory. This entry point is also known as EXE$ALONPAGVAR. (Since OpenVMS VAX Version 6.0, EXE$ALONPAGVAR is an obsolete routine.)

EXE$ALONONPAGED does not initialize the header of the allocated block of memory.

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 spinlock. As a result, its caller cannot be executing above IPL$_SYNCH or hold any spinlock ranked higher than MMG. (For instance, a driver fork process executing at IPL$_SYNCH holding the IOLOCK8 fork lock can call EXE$ALOPHYCNTG.)

EXE$ALOPHYCNTG returns control to its caller at IPL$_SYNCH. The caller retains any spinlock 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$L_ALTSTART	Address of alternate start-I/O routine
UCB$B_FLCK	Fork lock index
UCB$L_DDB	Address of unit's DDB
DDB$L_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.

EXE$CRAM_CMD

Processes a CSR read or write to a device connected to a remote bus.

Module

[SYSLOA]CRAM_ROUTINES_LSB

Input

Location	**Contents**
04(SP)	Pseudo CSR address (PCA)
08(SP)	Flags longword, containing data length, environment flag, and disposition flags, as supplied by the READ_CSR or WRITE_CSR macro
0A(SP)	Operation, either read or write
0C(SP)	Data to be written; reserved for data to be read
10(SP)	Address of user-supplied CRAM (optional)
R5	Address of CRB

Output

Location	**Contents**
R0	Status indicating success or failure of the operation

Synchronization

EXE$CRAM_CMD executes at the IPL necessary to read and write CSRs.

Description

EXE$CRAM_CMD is called from the READ_CSR and WRITE_CSR macros with all necessary parameters pushed on the stack. It processes the entire I/O transaction.

If no CRAM address has been supplied, the routine first allocates a CRAM by calling routine IOC$ALLOCATE_CRAM. Then, according to the requirements of the specified I/O interconnect (as determined from the PCA) and the input parameters, the routine calculates and fills the following fields of the hardware I/O mailbox within the CRAM:

- HW_CRAM$L_COMMAND
- HW_CRAM$B_BYTE_MASK
- HW_CRAM$Q_RBADR
- HW_CRAM$Q_WDATA (if the operation is a write)

EXE$CRAM_CMD then calls routine IOC$CRAM_IO to perform the actual hardware mailbox I/O transaction.

When the operation completes, EXE$CRAM_CMD checks the status return and processes any errors in accordance with the **error** argument of the READ_CSR or WRITE_CSR macro. If the operation was a successful read, the returned data is stored on the stack.

If a CRAM was allocated at the start of the routine, the CRAM is deallocated via a call to routine IOC$DEALLOCATE_CRAM. EXE$CRAM_CMD then returns to the caller.

Digital does not recommend calling this routine directly.

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$_SYNCH and obtain the JIB spinlock and the SCHED spinlock (if JIB$V_BYTCNT_WAITERS is set) in a multiprocessing environment. As a result, their callers cannot be executing above IPL$_SYNCH or hold any spinlock ranked higher than JIB. (For instance, a driver fork process executing at IPL$_SYNCH holding the IOLOCK8 fork lock can call these routines. It cannot, however, hold the SCHED spinlock.)

EXE$CREDIT_BYTCNT and EXE$CREDIT_BYTCNT_BYTLM return control to their callers at the caller's IPL. Their caller retains any spinlocks 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$DEANONPAGED, EXE$DEANONPGDSIZ

Deallocates a block of memory and returns it to nonpaged pool.

Module

MEMORYALC

Input

Location	**Contents**
R0	Address of block to be deallocated
R1	Size of block in bytes, if from variable pool (EXE$DEANONPGDSIZ only)
IRP$W_SIZE	Size of block in bytes (EXE$DEANONPAGED only)
IRP$B_TYPE	Type of block to be deallocated (EXE$DEANONPAGED only)

Note

The MSB of field IRP$B_TYPE must be zero, unless it is definning a shared memory structure.

Output

Location	**Contents**
R1 and R2	Destroyed

Synchronization

EXE$DEANONPAGED and EXE$DEANONPGDSIZ execute at the caller's IPL, at IPL$_SYNCH holding the SCHED spinlock, and at IPL$_POOL holding the POOL spinlock. As a result, the caller cannot be executing above IPL$_SYNCH. EXE$DEANONPAGED and EXE$DEANONPGDSIZ return control to the caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

EXE$DEANONPAGED and EXE$DEANONPGDSIZ deallocate the specified block of memory to nonpaged dynamic memory, returning it to a lookaside list or the variable region of nonpaged pool as appropriate. These routines also report 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.

If enabled by the SYSGEN parameter POOLCHECK, these routines overwrite portions of the deallocated pool with a checksum and a one-byte pattern. This action is helpful when tracking pool corruption problems.

Do not expect R0 to give good status upon returning, if it fails system bugchecked.

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$_SYNCH and obtain the JIB spinlock in a multiprocessing environment. As a result, their callers cannot be executing above IPL$_SYNCH or hold any spinlock 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 spinlock.)

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 spinlocks it held at the time of the call.

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.

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$_NORMAL, SS$_EXQUOTA, or SS$_INSFMEM
R1	Number of bytes deducted; bit 31 cleared
R2	Address of requested buffer
R3	Destroyed
JIB$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$C_BUFIO

Synchronization

EXE$DEBIT_BYTCNT_ALO and EXE$DEBIT_BYTCNT_BYTLM_ALO raise IPL to IPL$_SYNCH and obtain the JIB spinlock in a multiprocessing environment. Their callers cannot be executing above IPL$_SYNCH or hold any spinlock.

EXE$DEBIT_BYTCNT_ALO and EXE$DEBIT_BYTCNT_BYTLM_ALO return control to their callers at IPL$_ASTDEL.

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 there is insufficient nonpaged pool available for the buffer, these routines return SS$_INSFMEM status to the caller.

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.

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$_NORMAL
IRP$L_IOST1	First longword of I/O status
IRP$L_IOST2	Second longword of I/O status (cleared by EXE$FINISHIOC)
UCB$L_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 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$L_OPCNT). This task is performed at fork IPL, holding the associated fork lock in a multiprocessing environment.
- Stores the contents of R0 and R1 in the IRP.

- Inserts the IRP in the local processor's I/O postprocessing queue.
- If the queue is empty, requests a software interrupt from the local processor at IPL$_IOPOST.

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.

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 spinlocks 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 spinlock 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$INSERTIRP must be called at fork IPL or higher. In a multiprocessing environment, the caller must also hold the associated fork lock. EXE$INSERTIRP does not alter IPL or acquire any spinlocks. 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.

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$B_FLCK	Fork lock index
UCB$L_STS	UCB$V_BSY set indicates device is busy, clear indicates device is idle
UCB$L_IOQFL	Address of pending-I/O queue listhead
UCB$W_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$L_STS	UCB$V_BSY set.
UCB$W_QLEN	Incremented.

Synchronization

EXE$INSIOQ and EXE$INSIOQC immediately raise to fork IPL and, in a multiprocessing environment, obtain the corresponding fork lock. As a result, their callers must not be executing at an IPL higher than fork IPL or hold a spinlock 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$W_QLEN and proceed according to the status of the device (as indicated by UCB$V_BSY in UCB$W_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
TQE$Q_TIME	Quadword expiration time for TQE
EXE$GQ_1ST_TIME	Updated if TQE is inserted at the head of the timer queue

Synchronization

EXE$INSTIMQ immediately raises to IPL$_TIMER (IPL$_SYNCH), obtaining the TIMER spinlock in a multiprocessing environment. As a result, its caller must not be executing above IPL$_SYNCH or hold any spinlocks 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 spinlocks 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 spinlock in a multiprocessing environment), inserts it on the head of the queue, and updates EXE$GQ_1ST_TIME.

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 spinlocks 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 spinlock 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.

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.

---

[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.

- If MMG$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE$MODIFYLOCKR.

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.

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.

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.

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.

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 spinlocks 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.

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.

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.

---

[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.

- If MMG$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE$READLOCKR.

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.

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   R0,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$READLOCKR.
- Determines whether the specified buffer is write accessible for a read I/O function, with one of the following results:
  - If the buffer allows write access, EXE$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.

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
;
```

EXE$RMVTIMQ

Removes timer queue elements (TQEs) from the timer queue.

Module

EXSUBROUT

Input

Location	**Contents**
R2	Access mode (unused by system subroutine)
R3	Request identification (unused by system subroutine)
R4	Type of TQE entry (TQE$B_RQTYPE) to remove from queue (TQE$C_ SSNGL) if bit 31 is zero. If bit 31 is set, then R4 contains the address of the TQE.
R5	Process ID (TQE$L_PID)

Output

Location	**Contents**
R0	If R0=1, then at least one TQE was removed. If R0=0, then no TQE was removed.
R1	Destroyed

Synchronization

EXE$RMVTIMQ immediately raises to IPL$_TIMER (IPL$_SYNCH), obtaining the TIMER spinlock in a multiprocessing environment. As a result, its caller must not be executing above IPL$_SYNCH or hold any spinlocks of a higher rank. (For instance, a driver fork process executing at IPL$_SYNCH holding the IOLOCK8 fork lock can call EXE$RMVTIMQ and might need the SCHED and HWCLK spinlocks, but these impose no additional restrictions on the caller.)

EXE$RMVTIMQ returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

EXE$RMVTIMQ removes the specified TQEs from the timer queue. Entries are removed by address, type, access mode, request identification, and process ID. Any entries which meet matching criteria are removed from queue.

If a system subroutine or a wake request TQE is being removed, access mode and request identification need not be supplied. If the TQE address is supplied in R4, no other input need be supplied.

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 systemwide 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.

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_DEVBUFSIZ) 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$FINISHIO to insert the IRP in the systemwide I/O postprocessing queue.
- EXE$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_DEVBUFSIZ, 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. (SYS$AR_JOBCTLMB contains the address of the job controller's mailbox; SYS$AR_OPRMBX contains the address of OPCOM's mailbox.)
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 spinlock in a 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 spinlocks 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$W_UNIT)
4 through 31	Counted string of device controller name, formatted as *node$controller* for clusterwide devices

EXE$SNDEVMSG then calls EXE$WRTMAILBOX to send the message to a mailbox.

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.

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$WRITECHKR), or error status
R1	Size of transfer in bytes
R2	0, indicating a write function
IRP$W_STS	IRP$V_FUNC clear, indicating a write function
IRP$L_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$L_BCNT. If the byte count is negative, it returns SS$_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.

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   R0,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$_NORMAL
R1	System virtual address of the process page-table entry (PTE) that maps the first page of the buffer
R2	0, indicating a write function
IRP$W_STS	IRP$V_FUNC clear, indicating a write function
IRP$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$WRITELOCK and EXE$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$L_BCNT. If the byte count is negative, it returns SS$_BADPARAM status to 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 spinlock in a 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 spinlocks it held at the time of the call.

Description

EXE$WRTMAILBOX checks fields in the mailbox UCB (UCBW_BUFQUO, UCBW_DEVBUFSIZ) to determine whether it can deliver a message of the specified size to the mailbox. It also checks fields in the associated ORB to determine whether the caller is sufficiently privileged to write to the mailbox. Finally, it calls EXE$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.

EXE$ZEROPARM

Processes an I/O function code that requires no parameters.

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

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.

IOC$ALLOCATE_CRAM

Allocates a control register access mailbox (CRAM).

Module

[SYSLOA]CRAM_ROUTINES_LSB

Input

Location	**Contents**
04(SP)	Reserved for returned address of CRAM

Output

Location	**Contents**
04(SP)	Address of CRAM
R0	Status of operation

Synchronization

During normal processing, IOC$ALLOCATE_CRAM uses no spinlocks. However, if there are no free CRAMs available, the routine acquires and releases the MMG spinlock while allocating additional CRAMs from nonpaged memory. Thus, the caller should be running at or below IPL$_MMG. If the caller is running above IPL$_MMG, no attempt is made to allocate additional CRAMs, and IOC$ALLOCATE_CRAM returns an error status.

Description

IOC$ALLOCATE_CRAM first checks to see if memory has already been allocated for use as mailboxes. If memory has been allocated, the routine checks to see if a CRAM is available. If no CRAMs have been configured or none are available, the routine allocates four pages of system nonpaged memory and divides this memory into one CRAM header (CRAMH) and 15 CRAMs.

When an available CRAM is found, the routine marks it as unavailable and returns its address to the caller.

IOC$ALOALTMAP, IOC$ALOALTMAPN, IOC$ALOALTMAPSP

Allocate a set of Q22–bus alternate map registers.

Module

[SYSLOA]MAPSUB*xxx*

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$_SSFAIL
R1	Destroyed
R2	Address of ADP
CRB$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

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 multiprocessing environment. Each routine returns control to its caller at the caller's IPL. The caller retains any spinlocks 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 system does not support alternate map registers, the routine exits with SS$_SSFAIL status.

Device drivers generally obtain Q22–bus alternate map registers by calling IOC$REQALTMA which calls IOC$ALOALTMAP to do the actual allocating. If registers are not available, IOC$REQALTMA places the process on the map register wait queue and does not return to the caller until sufficient registers have been allocated.

IOC$ALOTCMAP_DMA, IOC$ALOTCMAP_DMAN

Allocate a set of TURBOchannel DMA map registers.

Module

[DRIVER]TCDMA_PTA

Input

Inputs for both routines follow:

Location	Contents
CRB$L_INTD+ VEC$L_ADP	Address of ADP
ADP$W_MRNREGARY ADP$W_MRFREGARY ADP$L_MRACTMDRS	Map register descriptor arrays
For IOC$ALOTCMAP_DMA only	
R5	Address of UCB
UCB$W_CRB	Address of CRB
UCB$W_BCNT	Transfer byte count
UCB$W_BOFF	Byte offset to start of transfer in first page
For IOC$ALOTCMAP_DMAN only	
R1	Address of the map register descriptor (TC_MD)
R2	Address of ADP
R3	Number of map registers to be allocated

Output

Outputs for both routines follow:

Location	Contents
R0	SS$_NORMAL or SS$_INSFMAPREG
R2	Address of ADP
ADP$W_MRNREGARY ADP$W_MRFREGARY ADP$L_MRACTMDRS	Updated
For IOC$ALOTCMAP_DMA only	
R1	Destroyed
CRB$L_INTD+ VEC$B_NUMREG	Number of map registers allocated
CRB$L_INTD+ VEC$W_MAPREG	Starting map register number
For IOC$ALOTCMAP_DMAN only	
R1	Address of the map register descriptor (TC_MD)

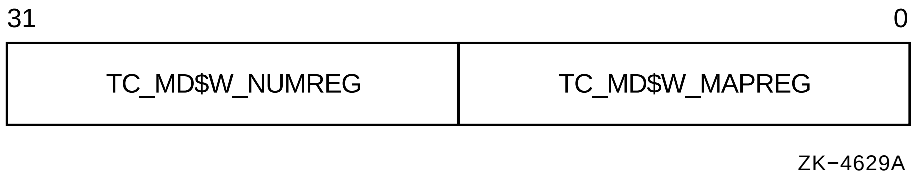
Synchronization

The caller of IOC$ALOTCMAP_DMA or IOC$ALOTCMAP_DMAN must be executing at fork IPL or above and must hold the corresponding fork lock (typically IOLOCK8) in a multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

IOC$ALOTCMAP_DMA and IOC$ALOTCMAP_DMAN allocate a contiguous set of TURBOchannel DMA map registers. IOC$ALOTCMAP_DMA records the allocation in the ADP and CRB while IOC$ALOTCMAP_DMAN records the same information in a map register descriptor. Figure 3–1 shows the structure of the map register descriptor used by IOC$ALOTCMAP_DMAN.

Figure 3–1 TURBOchannel Map Register Descriptor (TC_MD)



TC_MD$W_MAPREG contains the number of the first (starting) map register and TC_MD$W_NUMREG contains the number of map registers allocated.

These routines differ in the way in which they determine the number of map registers they allocate:

- IOC$ALOTCMAP_DMA calculates the number of map registers required using the values contained in UCB$W_BCNT and UCB$W_BOFF.
- IOC$ALOTCMAP_DMAN uses the value in R3 as the number of required registers.

If there are not enough contiguous map registers available, the routine returns an error status of SS$_INSFMAPREG to its caller.

Because the map registers eventually must be released, the caller of IOC$ALOTCMAP_DMAN must keep track of the map registers allocated. Care should be exercised in the consumption and management of map register resources.

When using the IOC$ALOTCMAP_DMA routine, note that if there are not enough map registers available, your driver can put a fork block onto the map register allocation wait queue in the ADP (ADP$L_MRQFL). When registers are released, the release routine checks for waiting fork threads. If any threads are waiting, the routine attempts to complete the allocation at that time.

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$ALOUBAMAPN only). The value should account for one extra register needed to prevent a transfer overrun.
R5	Address of UCB
UCB$W_BCNT	Transfer byte count (IOC$ALOUBAMAP only)
UCB$W_BOFF	Byte offset in page (IOC$ALOUBAMAP only)
UCB$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

Output

Location	**Contents**
R0	SS$_NORMAL or 0
R1	Destroyed
R2	Address of ADP
CRB$L_INTD+ VEC$B_NUMREG	Number of map registers allocated
CRB$L_INTD+ VEC$W_MAPREG	Starting map register number
ADP$W_MRNREGARY, ADP$W_MRFREGARY, ADP$L_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 multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

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.

Device drivers generally obtain UNIBUS map registers or a set of the first 496 Q22–bus map registers by calling IOC$REQMAPREG which calls IOC$ALOUBAMAP to do the actual allocating. If registers are not available, IOC$REQMAPREG places the process on the map register wait queue and does not return to the caller until sufficient registers have been allocated.

IOC$ALOVMEMAP_DMA, IOC$ALOVMEMAP_DMAN

Allocate a set of VME DMA map registers.

Module

[DRIVER]VMEDMA_XMI

Input

Location	**Contents**
CRB$L_INTD+ VEC$L_ADP	Address of ADP
ADP$W_MRNREGARY ADP$W_MRFREGARY ADP$L_MRACTMDRS	Map register descriptor arrays
For IOC$ALOVMEMAP_DMA only	
R5	Address of UCB
UCBW$_CRB	Address of CRB
UCB$W_BCNT	The transfer byte count
UCB$W_BOFF	Byte offset to start of transfer in first page
For IOC$ALOVMEMAP_DMAN only	
R1	Address of the map register descriptor (VME_MD)
R2	Address of ADP
R3	Number of map registers to be allocated

Output

Location	**Contents**
R0	SS$_NORMAL or SS$_INSFMAPREG
R2	Address of ADP
ADP$W_MRNREGARY, ADP$W_MRFREGARY, ADP$L_MRACTMDRS	Updated
For IOC$ALOVMEMAP_DMA only	
R1	Destroyed
CRB$L_INTD+ VEC$B_NUMREG	Number of map registers allocated
CRB$L_INTD+ VEC$W_MAPREG	Starting map register number
For IOC$ALOVMEMAP_DMAN only	
R1	Address of the map register descriptor (VME_MD)

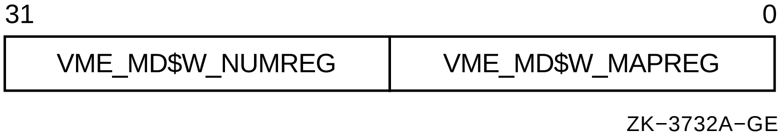
Synchronization

The caller of IOC$ALOVMEMAP_DMA or IOC$ALOVMEMAP_DMAN must be executing at fork IPL or above and must hold the corresponding fork lock (typically IOLOCK8) in a multiprocessing environment. Either routine returns control to its caller and the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

IOC$ALOVMEMAP_DMA and IOC$ALOVMEMAP_DMAN allocate a contiguous set of VME DMA map registers. IOC$ALOVMEMAP_DMA records the allocation in the ADP and CRB while IOC$ALOVMEMAP_DMAN records the same information in a map register descriptor. Figure 3–2 shows the structure of the map register descriptor used by IOC$ALOVMEMAP_DMAN.

Figure 3–2 VME Map Register Descriptor (VME_MD)



VME_MD$W_MAPREG contains the number of the first (starting) map register and VME_MD$W_NUMREG contains the number of map registers allocated.

These routines differ in the way in which they determine the number of map registers they allocate:

- IOC$ALOVMEMAP_DMA calculates the number of needed map registers using the values contained in UCB$W_BCNT and UCB$W_BOFF.
- IOC$ALOVMEMAP_DMAN uses the value in R3 as the number of required registers.

If there are not enough contiguous map registers available, the routine returns an error status of SS$_INSFMAPREG to its caller.

Because the map registers eventually must be released, the caller of IOC$ALOVMEMAP_DMAN must keep track of the map registers allocated. Care should be exercised in the consumption and management of map register resources.

When using the IOC$ALOVMEMAP_DMA routine, note that if there are not enough map registers available, your driver can put a fork block onto the map register allocation wait queue in the ADP (ADP$L_MRQFL). When registers are released, the release routine checks for waiting fork threads. If any threads are waiting, the routine attempts to complete the allocation at that time.

IOC$ALOVMEMAP_PIO

Alocates a set of VME PIO map registers.

Module

[DRIVER]VMEPIO_XMI, VMEPIO_TC

Input

Location	**Contents**
R3	Number of map registers to allocate
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of ADP
ADP$W_MR2NREGAR, ADP$W_MR2FREGAR, ADP$L_MR2ACTMDR	Map register descriptor arrays

Output

Location	**Contents**
R0	SS$_NORMAL or SS$_INSFMAPREG
R1	Destroyed
R2	Address of ADP
CRB$L_INTD+ VEC$B_NUMALT	Number of map registers allocated.
ADP$W_MR2NREGAR, ADP$W_MR2FREGAR, ADP$L_MR2ACTMDR	Updated

Synchronization

The caller of IOC$ALOVMEMAP_PIO must be executing at fork IPL or above and must hold the corresponding fork lock in a multiprocessing environment. IOC$ALOVMEMAP_PIO returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

IOC$ALOVMEMAP_PIO allocates a contiguous set of VME PIO map registers and records the allocation in the VMEbus adapter ADP and CRB.

IOC$ALOVMEMAP_PIO searches the map register descriptor arrays for the required number of map registers. If there are not enough contiguous map registers available, the routine returns an error status of SS$_INSFMAPREG to its caller.

Note that if there are not enough map registers available, your driver can put a fork block onto the map register allocation wait queue in the ADP (ADP$L_MRQFL). When registers are released, the release routine checks for waiting fork threads. If any threads are waiting, the routine attempts to complete the allocation at that time.

IOC$ALOXBIMAP, IOC$ALOXBIMAPN

Allocate a set of XBI+ map registers.

Module

[IO_ROUTINES]IOSUBNPAG

Input

Location	**Contents**
R3	Number of XBI+ map registers to allocate (IOC$ALOXBIMAPN only)
R5	Address of UCB
UCB$W_BCNT	Transfer byte count (IOC$ALOXBIMAP only)
UCB$W_BOFF	Byte offset in page (IOC$ALOXBIMAP only)
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of device ADP
ADP$L_BIMASTER	Address of XBI+ adapter ADP

Output

Location	**Contents**
R0	Status of operation
R1	Unpredictable
R2	Address of ADP
CRB$L_INTD+ VEC$W_XBINUMREG	Number of XBI+ map registers allocated
CRB$L_INTD+ VEC$W_MAPREG	Starting XBI+ map register number

Synchronization

Callers of IOC$ALOXBIMAP or IOC$ALOXBIMAPN must be executing at fork IPL or above. No specific spinlock is required. Each routine returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

IOC$ALOXBIMAP and IOC$ALOXBIMAPN allocate a contiguous set of XBI+ map registers 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 map registers they allocate:

- IOC$ALOXBIMAP calculates the number of map registers required using the values contained in UCB$W_BCNT and UCB$W_BOFF.
- IOC$ALOXBIMAPN uses the value in R3 as the number of required registers.

If an odd number of map registers is required, these routines round up this value to an even multiple of 64.

If XBI+ map registers have been permanently allocated to the controller, IOC$ALOXBIMAP and IOC$ALOXBIMAPN return a success status indicator (SS$_NORMAL) without allocating the requested map registers. Otherwise, they search for the required number of map registers, returning SS$_NORMAL when they are found. If there are not enough contiguous XBI+ map registers available, the routines return an error status indicator (SS$_INSFMAPREG).

Device drivers generally obtain XBI+ map registers by calling routine IOC$REQXBI which calls IOC$ALOXBIMAP to do the actual allocating. If registers are not available, IOC$REQXBI places the process on the map register wait queue and does not return to the caller until sufficient registers have been allocated.

IOC$ALOXBIMAPRM, IOC$ALOXBIMAPRMN

Permanently allocate a set of XBI+ map registers.

Module

[IO_ROUTINES]IOSUBNPAG

Input

Location	Contents
R3	Number of XBI+ map registers to allocate (IOC$ALOXBIMAPRMN only)
R5	Address of UCB
UCB$W_BCNT	Transfer byte count (IOC$ALOXBIMAPRM only)
UCB$W_BOFF	Byte offset in page (IOC$ALOXBIMAPRM only)
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of device ADP
ADP$L_BIMASTER	Address of XBI+ adapter ADP

Output

Location	Contents
R0	Status of operation
R1	Unpredictable
R2	Address of ADP
CRB$L_INTD+ VEC$W_XBINUMREG	Number of XBI+ map registers allocated
CRB$L_INTD+ VEC$W_MAPREG	Starting XBI+ map register number; bit 15 (VEC$M_MAPLOCK) set to indicate allocation is permanent

Synchronization

Callers of IOC$ALOXBIMAPRM or IOC$ALOXBIMAPRMN must be executing at fork IPL or above. No specific spinlock is required. Each routine returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

IOC$ALOXBIMAPRM and IOC$ALOXBIMAPRMN permanently allocate a contiguous set of XBI+ map registers 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 map registers they allocate:

- IOC$ALOXBIMAPRM calculates the number of map registers required using the values contained in UCB$W_BCNT and UCB$W_BOFF.
- IOC$ALOXBIMAPRMN uses the value in R3 as the number of required registers.

If an odd number of map registers is required, these routines round up this value to an even multiple of 64. The routines then search for the required number of map registers, returning SS$_NORMAL when they are found. If there are not enough contiguous XBI+ map registers available, the routines return an error status indicator (SS$_INSFMAPREG).

Once XBI+ map registers have been permanently allocated to the controller, subsequent calls to IOC$ALOXBIMAPRM and IOC$ALOXBIMAPRMN will return a success status indicator (SS$_NORMAL) without allocating additional map registers.

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 spinlocks, 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 1.19.

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$CANCELIO executes at its caller’s IPL, obtains no spinlocks, and returns control to its caller at the caller’s IPL. It is usually called by EXE$CANCEL (if specified in the DDT as the driver’s cancel-I/O routine) at fork IPL, holding the corresponding fork lock in a multiprocessing environment.

Description

IOC$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$CRAM_IO

Initiates an I/O operation to a device on a remote bus and waits for the operation to complete.

Module

[SYSLOA]CRAM_ROUTINES_LSB

Input

Location	**Contents**
04(SP)	Address of CRAM

Output

Location	**Contents**
R0	Status indicating success or failure of the operation

Synchronization

IOC$CRAM_IO executes at the IPL necessary to read and write CSRs.

Description

IOC$CRAM_IO is called by routine EXE$CRAM_CMD. It performs the entire hardware I/O mailbox transaction by queuing the hardware I/O mailbox to the mailbox pointer register (MBPR) and waiting for the transaction to complete.

IOC$CRAM_IO initiates the I/O operation by writing the physical address of the hardware I/O mailbox portion of the specified CRAM to the MBPR. If the routine is unable to post the mailbox to the MBPR within the queuing timeout interval (found at location CRAM$Q_QUEUE_TIME), it returns a status of SS$_INTERLOCK.

If the routine does successfully queue the mailbox, it sets the CRAM$V_IN_USE bit in location CRAM$B_CRAM_FLAGS and then repeatedly checks bit HW_CRAM$V_MBX_DONE in location HW_CRAM$W_MBX_FLAGS of the hardware I/O mailbox to determine when the operation has completed:

- If the operation completes successfully, the routine clears the CRAM$V_IN_USE bit and returns a status of SS$_NORMAL.
- If the operation completes with an error (bit HW_CRAM$V_MBX_ERROR set in location HW_CRAM$W_MBX_FLAGS), the routine clears the CRAM$V_IN_USE bit and returns a status of SS$_CTRLERR.

- If the operation does not complete within the timeout interval (found at location CRAM$Q_WAIT_TIME), the routine returns a status of SS$_TIMEOUT. Bit CRAM$V_CRAM_IN_USE is left set.

If IOC$CRAM_IO returns a status of SS$_NORMAL for a read operation, the requested data is stored in HW_CRAM$Q_RDATA of the hardware I/O mailbox. Note, however, that a normal return for a write operation does not necessarily guarantee that the write data stored in HW_CRAM$Q_WDATA has been successfully written to the device register.

IOC$DEALLOCATE_CRAM

Deallocates a control register access mailbox (CRAM).

Module

[SYSLOA]CRAM_ROUTINES_LSB

Input

Location	Contents
04(SP)	Address of CRAM

Output

Location	Contents
R0	Status of operation

Synchronization

IOC$DEALLOCATE_CRAM uses no spinlocks.

Description

IOC$DEALLOCATE_CRAM deallocates the CRAM by marking it available for use.

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
DDB$L_DDT	Address of DDT
DDT$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$DIAGBUFILL may be executing at or above fork IPL and must hold the corresponding fork lock in a multiprocessing environment. IOC$DIAGBUFILL returns control to its caller at the caller's IPL. The caller retains any spinlocks 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.

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$L_DIAGBUF	Address of diagnostic buffer, if one is present
EXE$GQ_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**
R0, 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_TIMOUT cleared
Diagnostic buffer	Current system time (first quadword)

Synchronization

IOC$INITIATE is called at fork IPL with the corresponding fork lock held in a multiprocessing system. Within this context, it transfers control to the driver's start-I/O routine.

Description

IOC$INITIATE creates the context in which a driver fork process services an I/O request. IOC$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$INITIATE takes steps to initiate the I/O function on another processor in a multiprocessing system. It then returns to its caller.
- Stores the address of the current IRP in UCB$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_TIMOUT 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.

IOC$IOPOST

Performs device-independent I/O postprocessing and delivers the results of an I/O request to a process.

Module

IOCIOPOST

Input

Location	**Contents**
IRP$L_PID	Process identification of the process that initiated the I/O request
IRP$L_UCB	Address of UCB
IRP$W_STS	IRP$V_BUFIO set if buffered-I/O request, clear if direct-I/O request; IRP$V_PHYSIO set if physical-I/O function; IRP$V_EXTEND set if an IRPE is linked to this IRP; IRP$V_KEY set if IRP$L_KEYDESC contains the address of an encryption key buffer; IRP$V_FUNC set if read function, clear if write function; IRP$V_DIAGBUF set if diagnostic buffer exists; IRP$V_MBXIO set if mailbox read function
IRP$L_DIAGBUF	Address of diagnostic buffer, if one is present
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$L_BCNT	Size in bytes of transfer
IRP$L_OBCNT	Original byte count for virtual I/O transfer
IRP$L_IOST1	First I/O status longword
IRP$W_CHAN	I/O request channel index number
IRP$L_IOSB	Address of I/O status block, if specified
IRP$B_RMOD	Access mode of I/O request; ACB$V_QUOTA set if request specified AST
IRP$B_EFN	Event flag number
UCB$W_QLEN	Length of pending-I/O queue
UCB$L_DEVCHAR	DEV$V_FOD set if file-oriented device
PCB$W_DIOCNT	Process's direct-I/O count
PCB$W_BIOCNT	Process's buffered-I/O count
JIB$L_BYTCNT	Job byte count quota
CCB$W_IOC	Number of outstanding I/O requests on channel
CCB$L_DIRP	Address of IRP for requested deaccess

Output

Location	**Contents**
UCB$W_QLEN	Decremented
PCB$W_DIOCNT	Incremented for a direct-I/O request
PCB$W_BIOCNT	Incremented for a buffered I/O request
JIB$L_BYTCNT	Updated for buffered I/O request
CCB$W_IOC	Decremented
CCB$L_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 spinlocks required to deallocate nonpaged pool and adjust process quotas.

Description

This interrupt service routine processes IRPs in the systemwide and local CPU I/O postprocessing queues, gaining control when the processor grants a software interrupt at IPL$_IOPOST. When the I/O postprocessing queues are empty, IOC$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 SS$_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.

IOC$LOADALTMAP

Loads a set of Q22–bus alternate map registers.

Module

[SYSLOA]MAPSUB*xxx*

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$LOADALTMAP at fork IPL, holding the corresponding fork lock in a multiprocessing environment. IOC$LOADALTMAP returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

A driver fork process calls IOC$LOADALTMAP to load a previously allocated set of alternate map registers with page-frame numbers (PFNs). This enables a device DMA transfer to or from the buffer indicated by the contents of UCBL_SVAPTE, UCBW_BCNT, and UCB$W_BOFF.

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 system does not support alternate map registers, the routine exits with SS$_SSFAIL status.

IOC$LOADMBAMAP

Loads MASSBUS map registers.

Module

LOADMREG

Macro

LOADMBA

Input

Location	Contents
R4	Address of MBA configuration register (MBA$L_CSR)
R5	Address of UCB
UCB$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
MBA$L_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$LOADMBAMAP uses the contents of UCBL_SVAPTE, UCBW_BCNT, and UCB$W_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$LOADMBAMAP also loads the negated transfer size into the MASSBUS adapter’s byte count register (MBA$L_BCR) and the byte offset of the transfer into the MASSBUS adapter’s virtual address register (MBA$L_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.

IOC$LOADTCMAP_DMA, IOC$LOADTCMAP_DMAN

Load a set of TURBOchannel map registers for DMA.

Module

[DRIVER]TCDMA_PTA

Input

Inputs for both routines follow:

Location	Contents
For IOC$LOADTCMAP_DMA only	
R5	Address of UCB
CRB$L_INTD+ VEC$L_ADP	Address of ADP
UCB$W_BCNT	Number of bytes in transfer
UCB$W_BOFF	Byte offset to start of transfer in first page
UCB$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$W_MAPREG	Number of first map register allocated
For IOC$LOADTCMAP_DMAN only	
R1	Address of the map register descriptor
R2	Address of ADP
R3	System virtual address (SVAPTE) of first page to transfer
R4	Byte count of the transfer
R5	Byte offset to start of transfer in first page

Output

Outputs for both routines follow:

Location	Contents
R1, R2	Destroyed

Synchronization

A driver fork process calls IOC$LOADTCMAP_DMA or IOC$LOADTCMAP_DMAN at fork IPL, holding the corresponding fork lock (typically IOLOCK8) in a multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

A driver fork process calls IOC$LOADTCMAP_DMA or IOC$LOADTCMAP_DMAN to load a previously allocated set of DMA map registers with page-frame numbers (PFNs). This enables a device to perform DMA transfers to or from the buffer indicated by the contents of UCBL_SVAPTE, UCBW_BCNT, and UCB$W_BOFF (or the contents of R3, R4, and R5 when using IOC$LOADTCMAP_DMAN).

IOC$LOADTCMAP_DMA or IOC$LOADTCMAP_DMAN checks whether sufficient map registers were allocated. If there are insufficient map registers, the routine issues a UBMAPEXCED bugcheck. Otherwise, the routine loads the appropriate PFN into each map register.

IOC$LOADTCMAP_DMA and IOC$LOADTCMAP_DMAN load and set the mapping register valid bit for the number of mapping registers needed for the length of the DMA request. Both routines clear the last map register. This last invalid register prevents a transfer overrun.

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 multiprocessing environment. Either routine returns control to its caller at the caller’s IPL. The caller retains any spinlocks 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 UCBL_SVAPTE, UCBW_BCNT, and UCB$W_BOFF.

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.

IOC$LOADVMEMAP_DMA, IOC$LOADVMEMAP_DMAN

Load a set of VME map registers for DMA.

Module

[DRIVER]VMEDMA_XMI

Input

Location	**Contents**
R0	VMEbus control flags: VME$M_RMWMODE—Translate VME read-modify-write into XMI interlocked accesses VME$K_WORDSWAP—Enables hardware-assisted byte swapping within words. VME$K_LONGSWAP—Enables hardware-assisted byte swapping within longwords.
CRB$L_INTD+ VEC$L_ADP	Address of ADP
For IOC$LOADVMEMAP_DMA only	
R5	Address of the UCB
UCB$W_BCNT	Number of bytes in transfer
UCB$W_BOFF	Byte offset to start of transfer in first page
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
UCB$L_SVAPTE	System virtual address of PTE for the first page of the transfer
For IOC$LOADVMEMAP_DMAN only	
R1	Address of the VME map register descriptor (VME_MD shown in Figure 3–2)
R2	Address of ADP
R3	System virtual address (SVAPTE) of first page to transfer
R4	Byte count of the transfer
R5	Byte offset to start of transfer in first page

Output

Location	**Contents**
R0, R1, R2	Destroyed

Synchronization

A driver fork process calls IOC$LOADVMEMAP_DMA or IOC$LOADVMEMAP_DMAN at fork IPL, holding the corresponding fork lock (typically IOLOCK8) in a multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

A driver fork process calls IOC$LOADVMEMAP_DMA or IOC$LOADVMEMAP_DMAN to load a previously allocated set of DMA map registers with page-frame numbers (PFNs). This enables a device to perform DMA transfer to or from the buffer indicated by the contents of UCBL_SVAPTE, UCBW_BCNT, and UCB$W_BOFF (or the contents of R3, R4, and R5 when using IOC$LOADVMEMAP_DMAN).

IOC$LOADVMEMAP_DMA or IOC$LOADVMEMAP_DMAN checks whether sufficient map registers were allocated. If there are insufficient map registers, the routine issues a UBMAPEXCED bugcheck. Otherwise, the routine loads the appropriate PFN into each map register.

IOC$LOADVMEMAP_DMA and IOC$LOADVMEMAP_DMAN check the VMEbus control-flags register and set the appropriate bits in each map register.

The IOC$ALOVMEMAP routines load and set the mapping register valid for the number of mapping registers needed for the length of the DMA request. Both routines also clear the last map register. This last invalid register prevents a transfer overrun.

The routines also set the byte swapping requested and the type of access for the VME bus. Access type is whether VME read-modify-writes are translated into XMI interlocked accesses or not.

IOC$LOADVMEMAP_PIO

Loads a set of VME PIO map registers.

Module

[DRIVER]VMEPIO_XMI, VMEPIO_TC

Input

Location	**Contents**
R0	VME address
R1	VMEbus access flags: <VME$K_SHORT@PIOMAP$V_ADRLEN>—VME access in short address-space mode <VME$K_STAND@PIOMAP$V_ADRLEN>—VME access in standard address-space mode <VME$K_EXTEND@PIOMAP$V_ADRLEN>—VME access in extended address-space mode <VME$K_BYTE@PIOMAP$V_DATALEN>—VME byte accesses <VME$K_WORD@PIOMAP$V_DATALEN>—VME word accesses <VME$K_LONG@PIOMAP$V_DATALEN>—VME longword accesses
R3	Number of registers to load
R5	Address of UCB
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$W_NUMALT	Number of PIO map registers allocated
CRB$L_INTD+ VEC$W_MAPALT	Number of first PIO map register allocated
CRB$L_INTD+ VEC$L_ADP	Address of ADP
ADP$L_MR2ADDR	Address of first VME PIO map register

Output

Location	**Contents**
R0	SS$_NORMAL, SS$_INSFMAPREG, or SS$_FAIL
R1, R2	Destroyed

Synchronization

A driver fork process calls IOC$LOADVMEMAP_PIO at fork IPL, holding the corresponding fork lock in a multiprocessing environment. IOC$LOADVMEMAP_PIO returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

A driver fork process calls IOC$LOADVMEMAP_PIO to load a previously allocated set of map registers with VME PFNs. For the DWMVA adapter, a VME PFN for programmed I/O access contains bits A<31:20>. The low order bits A<19:0> are taken from the XMI I/O address offset that corresponds to the map register in question. For more detail, see the adapter technical manual.

The VME address type, access length, and access mode are all controlled by setting or clearing the appropriate flags in the access flags register.

IOC$LOADVMEMAP_PIO confirms that sufficient VME PIO 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.

IOC$LOADXBIMAP

Loads a set of XBI+ map registers.

Module

[IO_ROUTINES]IOSUBNPAG

Input

Location	**Contents**
R5	Address of UCB
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of device ADP
ADP$L_BIMASTER	Address of XBI+ adapter ADP
CRB$L_INTD+ VEC$W_XBINUMREG	Number of XBI+ map registers to load
CRB$L_INTD+ VEC$W_MAPREG	Starting XBI+ map register number
UCB$L_SVAPTE	System virtual address of first page table entry (PTE) from which to extract the page frame numbers (PFNs)

Output

Location	**Contents**
R0	Status of operation
R1, R2	Unpredictable

Synchronization

Callers of IOC$LOADXBIMAP must be executing at fork IPL or above. No specific spinlock is required. The routine returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

IOC$LOADXBIMAP loads a contiguous set of XBI+ map registers with page frame numbers, as specified in the UCB, and marks the map registers as containing valid data. The routine also clears one extra map register, rendering it invalid. This invalid registers prevents transfer overrun.

The routine assumes that a prior call to IOC$REQXBIMAP, IOC$ALOXBIMAP /N, or IOC$ALOXBIMAPRM/N has allocated the registers and entered valid information into the CRB about the number of registers and starting register number.

IOC$MOVFRUSER, IOC$MOVFRUSER2

Move data from a user buffer to a device.

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

R0	Next address of user's buffer

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 multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

A driver calls IOC$MOVFRUSER and IOC$MOVFRUSER2 to move data from a user buffer to a device that cannot itself map the user buffer to system virtual addresses (for instance, a non-DMA device).

In order to accomplish the move, IOC$MOVFRUSER and IOC$MOVFRUSER2 first map the user buffer using the system page-table entry (SPTE) the driver allocated in a DPTAB macro invocation. If an SPTE has not been allocated to the driver, these routines cause an access violation when they attempt to refer to the location addressed by the contents of the field UCB$L_SVAPTE. (See the description of the DPTAB macro in Chapter 2 for information on how to allocate this SPTE.)

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.

IOC$MOVTOUSER, IOC$MOVTOUSER2

Move data from a device to a user buffer.

Module

BUFFERCTL

Input

Location	Contents
R0	User buffer address to which to move the byte (IOC$MOVTOUSER2 only)
R1	Address of driver's buffer
R2	Number of bytes to move
R5	Address of UCB
DPT$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

Location	Contents
R0	Next starting address of user's buffer

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 multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

A driver calls IOC$MOVTOUSER and IOC$MOVTOUSER2 to move data from a device to a user buffer when the device itself (for instance, a non-DMA device) cannot map the user buffer to system virtual addresses.

In order to accomplish the move, IOC$MOVTOUSER and IOC$MOVTOUSER2 first map the user buffer using the system page-table entry (SPTE) the driver allocated in a DPTAB macro invocation. If an SPTE has not been allocated to the driver, these routines cause an access violation when they attempt to refer to the location addressed by the contents of the field UCB$L_SVAPTE. (See the description of the DPTAB macro in Chapter 2 for information on how to allocate this SPTE.)

IOC$MOVTOUSER2 is useful for moving blocks of data in several pieces, each piece beginning within a page rather than on a page boundary. It handles as many pages as you need. To begin, the driver calls IOC$MOVTOUSER. For each subsequent buffer to move, the driver calls IOC$MOVTOUSER2.

IOC$PURGDATAP

Purges the buffered data path and logs memory errors that may have occurred during an I/O transfer.

Module

[SYSLOA]LIOSUB*xxx*

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 multiprocessing environment. It returns control to its caller at the caller's IPL. The caller retains any spinlocks 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 systems), 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$L_CRB → CRB$L_INTD+VEC$L_ADP → ADP$L_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.
- 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]MAPSUB*xxx*

Macro

RELALT

Input

Location	**Contents**
R5	Address of UCB
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of ADP
CRB$L_INTD+ VEC$W_MAPALT	Starting alternate map register number; VEC$V_ALTLOCK set indicates that alternate map registers have been permanently allocated to this controller
CRB$L_INTD+ VEC$W_NUMALT	Number of allocated alternate map registers
ADP$L_MR2QFL	Head of queue of UCBs waiting for alternate map registers
ADP$W_MR2NREGAR, ADP$W_MR2FREGAR, ADP$L_MR2ACTMDR	Alternate map register descriptor arrays

Output

Location	**Contents**
R0	SS$_NORMAL or SS$_SSFAIL
R1, R2	Destroyed
ADP$W_MR2NREGAR, ADP$W_MR2FREGAR, ADP$L_MR2ACTMDR	Updated

Synchronization

A driver fork process calls IOC$RELALTMAP at fork IPL, holding the corresponding fork lock in a multiprocessing environment.

Description

A driver fork process calls IOC$RELALTMAP to release a previously allocated set of Q22–bus alternate map registers (registers 496 to 8191) and update the alternate map register descriptor arrays in the ADP. IOC$RELMAPREG assumes that its caller is the current owner of the controller data channel.

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$L_CRB	Address of CRB
CRB$L_LINK	Address of secondary CRB
CRB$B_MASK	CRB$V_BSY set if the channel is busy
CRB$L_INTD+ VEC$L_IDB	Address of IDB
IDB$L_OWNER	Address of UCB of channel owner
CRB$L_WQFL	Head of queue of UCBs waiting for the controller channel

Output

Location	**Contents**
R0, R1, R2	Destroyed
IDB$L_OWNER	Cleared if no driver is waiting for the channel
CRB$B_MASK	CRB$V_BSY cleared if no driver is waiting for the channel

Synchronization

A driver fork process calls IOC$RELCHAN at fork IPL, holding the corresponding fork lock in a multiprocessing environment. IOC$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$RELCHAN dequeues a process, assigns the channel to that process, restores R3 and R5, moves the address of the CSR (IDB$L_CSR) into R4, and reactivates the suspended fork process.

IOC$RELDATAP

Releases a UNIBUS adapter’s buffered data path.

Module

IOSUBNPAG

Macro

RELDPR

Input

Location	**Contents**
R5	Address of UCB
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of ADP
CRB$L_INTD+ VEC$B_DATAPATH	Data path specifier; VEC$V_PATHLOCK set if the data path has been permanently allocated to the controller
ADP$L_DPQFL	Head of queue of UCBs waiting for a UNIBUS adapter buffered data path
ADP$W_DPBITMAP	Data path bit map

Output

Location	**Contents**
R0, R1, R2	Destroyed
ADP$W_DPBITMAP	Bit representing data path set if the path is not allocated to another driver fork process
CRB$L_INTD+ VEC$B_DATAPATH	Bits 0 through 4 cleared if the path is not permanently allocated

Synchronization

A driver fork process calls IOC$RELDATAP at fork IPL, holding the corresponding fork lock in a multiprocessing environment. IOC$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$RELDATAP obtains the number of the allocated data path from bits 0 through 4 of the data path specifier. If VEC$V_PATHLOCK is set in the specifier, the data path has been permanently allocated to the controller and IOC$RELDATAP returns to its caller.

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.

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$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of ADP
CRB$L_INTD+ VEC$W_MAPREG	Starting map register number; VEC$V_MAPLOCK set indicates that map registers have been permanently allocated to this controller
CRB$L_INTD+ VEC$B_NUMREG	Number of allocated map registers
ADP$L_MRQFL	Head of queue of UCBs waiting for map registers
ADP$W_MRNREGARY, ADP$W_MRFREGARY, ADP$L_MRACTMDRS	Map register descriptor arrays

Output

Location	**Contents**
R0	SS$_NORMAL or SS$_SSFAIL
R1, R2	Destroyed
ADP$W_MRNREGARY, ADP$W_MRFREGARY, ADP$L_MRACTMDRS	Updated

Synchronization

A driver fork process calls IOC$RELMAPREG at fork IPL, holding the corresponding fork lock in a 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.

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.

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$L_CRB	Address of CRB
CRB$L_LINK	Address of secondary CRB
CRB$B_MASK	CRB$V_BSY set if the channel is busy
CRB$L_INTD+ VEC$L_IDB	Address of IDB
IDB$L_OWNER	Address of UCB of channel owner
CRB$L_WQFL	Head of queue of UCBs waiting for the controller channel

Output

Location	**Contents**
R0, R1, R2	Destroyed
IDB$L_OWNER	Cleared if no driver is waiting for the channel
CRB$B_MASK	CRB$V_BSY cleared if no driver is waiting for the channel

Synchronization

A driver fork process calls IOC$RELSCHAN at fork IPL, holding the corresponding fork lock in a multiprocessing environment. IOC$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$RELTCMAP_DMA, IOC$RELTCMAP_DMAN

Release a set of TURBOchannel DMA map registers.

Module

[DRIVER]TCDMA_PTA

Input

Inputs for both routines follow:

Location	**Contents**
ADP$L_MRQFL	Head of queue of UCBs waiting for map registers
ADP$W_MRNREGARY ADP$W_MRFREGARY ADP$L_MRACTMDRS	Map register descriptor arrays
For IOC$RELTCMAP_DMA only	
R5	Address of UCB
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of ADP
CRB$L_INTD+ VEC$W_MAPREG	Starting map register number
CRB$L_INTD+ VEC$B_NUMREG	Number of allocated map registers
For IOC$RELTCMAP_DMAN only	
R1	Address of map register descriptor
R2	Address of ADP

Output

Outputs for both routines follow:

Location	**Contents**
R0	SS$_NORMAL or SS$_SSFAIL
R1	Destroyed
ADP$W_MRNREGARY ADP$W_MRFREGARY ADP$L_MRACTMDRS	Updated

Synchronization

A driver fork process calls IOC$RELTCMAP_DMA or IOC$RELTCMAP_DMAN at fork IPL, holding the corresponding fork lock in a multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

A driver fork process calls IOC$RELTCMAP_DMA or IOC$RELTCMAP_DMAN to release a previously allocated set of the TURBOchannel DMA map registers.

IOC$RELTCMAP_DMA obtains the location and number of the allocated map registers from CRB$L_INTED+VEC$W_MAPREG and CRB$L_INTED+VEC$B_NUMREG, respectively, while IOC$RELTCMAP_DMAN obtains this same information from the map register descriptor (TC_MD).

After adjusting the map register descriptor arrays, IOC$RELTCMAP routines examine the TURBOchannel DMA map register wait queue, located in the ADP at ADP$L_MRQFL. If the queue is empty, IOC$RELTCMAP returns successfully to its caller. If the queue contains waiting fork processes, IOC$RELTCMAP dequeues the first process and calls IOC$ALOTCMAP_DMA to attempt to allocate the set of map registers it requires.

If IOC$ALOTCMAP is called with sufficient map registers available, IOC$RELTCMAP restores R3 through R5 to the process and reactivates it. When this fork process returns control to IOC$RELTCMAP, IOC$RELTCMAP attempts to allocate map registers to the next waiting fork process. IOC$RELTCMAP continues to allocate map registers in this manner until the map-register wait queue is empty or it cannot satisfy the requirements of the process at the head of the queue. In the latter event, IOC$RELTCMAP reinserts the fork process UCB in the queue and returns successfully to its caller.

IOC$RELVMEMAP_DMA, IOC$RELVMEMAP_DMAN

Release a set of VME DMA map registers.

Module

[DRIVER]VMEDMA_XMI

Input

Location	**Contents**
ADP$L_MRQFL	Head of queue of UCBs waiting for map registers
ADP$W_MRNREGARY ADP$W_MRFREGARY ADP$L_MRACTMDRS	Map register descriptor arrays
For IOC$RELVMEMAP_DMA only	
R5	Address of UCB
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of ADP
CRB$L_INTD+ VEC$B_NUMREG	Number of allocated map registers
For IOC$RELVMEMAP_DMAN only	
R1	Address of map register descriptor (VME_MD shown in Figure 3–2)
R2	Address of ADP

Output

Location	**Contents**
R0	SS$_NORMAL or SS$_SSFAIL
R1, R2	Destroyed
ADP$W_MRNREGARY ADP$W_MRFREGARY ADP$L_MRACTMDRS	Updated

Synchronization

A driver fork process calls IOC$RELVMEMAP_DMA or IOC$RELVMEMAP_DMAN at fork IPL, holding the corresponding fork lock in a multiprocessing environment. Either routine returns control to its caller at the caller’s IPL. The caller retains any spinlocks it held at the time of the call.

Description

A driver fork process calls IOC$RELVMEMAP_DMA or IOC$RELVMEMAP_DMAN to release a previously allocated set of VME DMA map registers.

IOC$RELVMEMAP_DMA obtains the location and number of the allocated map registers from CRB$L_INTED+VEC$W_MAPREG and CRB$L_INTED+VEC$B_NUMREG, respectively, while IOC$RELVMEMAP_DMAN obtains this same information from the map register descriptor (VME_MD).

After adjusting the map register descriptor arrays, IOC$RELVMEMAP_DMA examines the VME DMA map register wait queue, located in the ADP at ADP$L_MRQFL. If the queue is empty, IOC$RELVMEMAP_DMA returns successfully to its caller. If the queue contains waiting fork processes, IOC$RELVMEMAP_DMA dequeues the first process and calls IOC$ALOVMEMAP_DMA to attempt to allocate the set of map registers it requires.

If IOC$ALOVMEMAP is called with sufficient map registers available, IOC$RELVMEMAP restores R3 through R5 to the process and reactivates it. When this fork process returns control to IOC$RELVMEMAP, IOC$RELVMEMAP attempts to allocate map registers to the next waiting fork process. IOC$RELVMEMAP continues to allocate map registers in this manner until the map-register wait queue is empty or it cannot satisfy the requirements of the process at the head of the queue. In the latter event, IOC$RELVMEMAP reinserts the fork process UCB in the queue and returns successfully to its caller.

IOC$RELVMEMAP_PIO

Releases a set of VME PIO map registers.

Module

[DRIVER]VMEPIO_XMI, VMEPIO_TC

Input

Location	**Contents**
R5	Address of UCB
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of ADP
CRB$L_INTD+ VEC$B_NUMALT	Number of allocated PIO map registers
ADP$L_MR2QFL	Head of queue of UCBs waiting for PIO map registers
ADP$W_MR2NREGAR, ADP$W_MR2FREGAR, ADP$L_MR2ACTMDR	PIO Map register descriptor arrays

Output

Location	**Contents**
R0	SS$_NORMAL or SS$_SSFAIL
R1, R2	Destroyed
ADP$W_MR2NREGAR, ADP$W_MR2FREGAR, ADP$L_MR2ACTMDR	Updated

Synchronization

A driver fork process calls IOC$RELVMEMAP_PIO at fork IPL, holding the corresponding fork lock in a multiprocessing environment.

Description

A driver fork process calls IOC$RELVMEMAP_PIO to release a previously allocated set of VME PIO map registers in the ADP.

IOC$RELVMEMAP_PIO obtains the location and number of the allocated map registers from CRB$L_INTED+VEC$W_MAPALT and CRB$L_INTED+VEC$W_NUMALT, respectively.

After adjusting the PIO map register descriptor arrays, IOC$RELVMEMAP_PIO examines the VME PIO map register wait queue. If the queue is empty, IOC$RELVMEMAP_PIO returns successfully to its caller. If the queue contains waiting fork processes, IOC$RELVMEMAP_PIO dequeues the first process and calls IOC$ALOVMEMAP_PIO to attempt to allocate the set of map registers it requires.

If there are sufficient alternate map registers, IOC$RELVMEMAP_PIO restores R3 through R5 to the process and reactivates it. When this fork process returns control to IOC$RELVMEMAP_PIO, IOC$RELVMEMAP_PIO attempts to allocate map registers to the next waiting fork process. IOC$RELVMEMAP_PIO continues to allocate map registers in this manner until the VMEPIO-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$RELVMEMAP_PIO reinserts the fork process UCB in the queue and returns successfully to its caller.

IOC$RELXBIMAP

Releases a set of XBI+ map registers.

Module

[IO_ROUTINES]IOSUBNPAG

Input

Location	**Contents**
R5	Address of UCB
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of device ADP
ADP$L_BIMASTER	Address of XBI+ adapter ADP
CRB$L_INTD+ VEC$W_XBINUMREG	Number of XBI+ map registers to release
CRB$L_INTD+ VEC$W_MAPREG	Starting XBI+ map register number

Output

Location	**Contents**
R0	Status of operation

Synchronization

Callers of IOC$RELXBIMAP must be executing at IOLOCK8 IPL or above. No specific spinlock is required. The routine returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

IOC$RELXBIMAP deallocates a contiguous set of XBI+ map registers, as specified in the VEC structure of the CRB.

If the map registers have been permanently allocated to the controller, IOC$RELXBIMAP returns a success status indicator (SS$_NORMAL) without deallocating the specified map registers. Otherwise, the routine deallocates the registers by calling IOC$DALOXBIMAP. The routine then checks the XBI map register wait queue (ADP$L_MPRQFL) to determine if other processes are waiting for XBI+ map registers. If so, the routine honors those allocation requests (if possible) before returning to the caller.

IOC$REQALTMA

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

Output

Location	**Contents**
R0	SS$_NORMAL or SS$_SSFAIL
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$L_MR2ACTMDR	Updated
ADP$L_MR2QBL	Updated
UCB$L_FR3	R3 of caller
UCB$L_FR4	R4 of caller
UCB$L_FPC	00(SP)

Synchronization

A driver fork process calls IOC$REQALTMAP at fork IPL, holding the corresponding fork lock in a multiprocessing environment.

Description

A driver fork process calls IOC$REQALTMAP to allocate a contiguous set of Q22–bus alternate map registers (registers 496 to 8191) to service the DMA transfer described by UCB$W_BCNT and UCB$W_BOFF. IOC$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$ALOALTMAP determines the required number of alternate map registers from the contents of UCB$W_BOFF and UCB$W_BCNT. It allocates one extra map register; this register is marked invalid when the driver fork process subsequently calls IOC$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$REQALTMAP cannot allocate a sufficient number of contiguous map registers, it saves process context by placing the contents of R3, R4, and the PC into the UCB fork block and the UCB into the alternate-map-register wait queue (ADP$L_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.

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.

Synchronization

A driver fork process calls IOC$REQCOM at fork IPL, holding the corresponding fork lock in a multiprocessing environment. IOC$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 in systemwide I/O postprocessing queue.
- 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$L_CRB	Address of CRB
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of ADP
CRB$L_INTD+ VEC$B_DATAPATH	Data path specifier; VEC$V_PATHLOCK set if the data path is permanently allocated to the controller
ADP$W_DPBITMAP	Data path bit map

Output

Location	Contents
R0	SS$_NORMAL or bit 0 set (indicating error status)
CRB$L_INTD+ VEC$B_DATAPATH	Data path specifier
ADP$W_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 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.

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

Output

Location	**Contents**
R0	SS$_NORMAL
R1	Destroyed
R2	Address of ADP
CRB$L_INTD+ VEC$B_NUMREG	Number of map registers allocated
CRB$L_INTD+ VEC$W_MAPREG	Starting map register number
ADP$W_MRNREGARY, ADP$W_MRFREGARY, ADP$L_MRACTMDRS	Updated

ADP$L_MRQBL	Updated
UCB$L_FR3	R3 of caller
UCB$L_FR4	R4 of caller
UCB$L_FPC	00(SP)

Synchronization

A driver fork process calls IOC$REQMAPREG at fork IPL, holding the corresponding fork lock in a multiprocessing environment.

Description

A driver fork process calls IOC$REQMAPREG to allocate a contiguous set of UNIBUS map registers or a set of the first 496 Q22–bus map registers to service the DMA transfer described by UCB$W_BCNT and UCB$W_BOFF. IOC$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$ALOUBAMAP determines the required number of map registers from the contents of UCB$W_BOFF and UCB$W_BCNT. It allocates one extra map register; this register is marked invalid when the driver fork process subsequently calls IOC$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$REQMAPREG cannot allocate a sufficient number of contiguous map registers, it saves process context by placing the contents of R3, R4, and the PC into the UCB fork block and R5 into the standard-map-register wait queue (ADP$L_MRQBL). It then returns to its caller's caller.

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, REQSCHAN

Input

Location	**Contents**
R5	Address of UCB
00(SP)	Return PC of caller
04(SP)	Return PC of caller's caller
UCB$L_CRB	Address of CRB
CRB$L_LINK	Address of secondary CRB (IOC$REQSCHANH and IOC$REQSCHANL only)
CRB$B_MASK	CRB$V_BSY set if the channel is busy
CRB$L_INTD+ VEC$L_IDB	Address of IDB
CRB$L_WQFL	Head of queue of UCBs waiting for the controller channel
CRB$L_WQBL	Tail of queue of UCBs waiting for the controller channel
IDB$L_CSR	Address of device CSR

Output

Location	**Contents**
R0, R1, R2	Destroyed
R4	Address of device CSR
IDB$L_OWNER	Address of UCB
CRB$L_WQFL	Updated
CRB$L_WQBL	Updated

Synchronization

A driver fork process calls IOC$REQPCHANH, IOC$REQPCHANL, IOC$REQSCHANH, or IOC$REQSCHANL holding the corresponding fork lock in a multiprocessing environment.

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$REQXBIMAP

Requests a set of XBI+ map registers, and if unavailable, places the requesting fork process in the XBI+ map register wait queue.

Module

[IO_ROUTINES]IOSUBNPAG

Input

Location	**Contents**
R5	Address of UCB
UCB$W_BCNT	Transfer byte count (IOC$ALOXBIMAP only)
UCB$W_BOFF	Byte offset in page (IOC$ALOXBIMAP only)
UCB$L_CRB	Address of CRB
CRB$L_INTD+ VEC$L_ADP	Address of device ADP
ADP$L_BIMASTER	Address of XBI+ adapter ADP

Output

Location	**Contents**
R0	Status of operation
R1	Unpredictable
R2	Address of ADP
CRB$L_INTD+ VEC$W_XBINUMREG	Number of XBI+ map registers allocated
CRB$L_INTD+ VEC$W_MAPREG	Starting XBI+ map register number

Synchronization

Callers of IOC$REQXBIMAP must be executing at fork IPL or above. No specific spinlock is required. The routine returns control to its caller at the caller's IPL. The caller retains any spinlocks it held at the time of the call.

Description

IOC$REQXBIMAP allocates a contiguous set of XBI+ map registers and records the allocation in the ADP and CRB. It calculates the number of needed map registers using the values contained in UCB$W_BCNT and UCB$W_BOFF. If an odd number of map registers is required, the value is rounded up to an even multiple of 64.

If XBI+ map registers have been permanently allocated to the controller, IOC$REQXBIMAP returns a success status indicator (SS$_NORMAL) without allocating the requested map registers. Otherwise, the routine searches for the required number of map registers, returning SS$_NORMAL when they are found.

If there are not enough contiguous XBI+ map registers available, the routine places the process fork block onto the XBI+ map register wait queue (ADP$L_MPRQFL) to wait until enough map registers are available.

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$RETURN is a universal executive routine vector in the fixed portion of the system executive. It contains a single RSB instruction. When a driver invokes the DDTAB macro, the macro writes the address of IOC$RETURN into routine address fields of the DDT that are not supplied in the macro invocation.

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$GL_CCBBASE	Base address of process CCB table
CCB$B_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 CCBB_AMOD, IOCVERIFYCHAN returns SS$_NORMAL!SS$_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**
R3, R4	(Preserved)
R5	Address of UCB
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.

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$L_FPC) so that the driver's interrupt service routine can resume execution of the driver fork thread with a JSB instruction.
- Moves contents of R3 and R4 into the UCB fork block.
- Sets UCB$V_INT to indicate an expected interrupt from the device unit.
- Sets UCB$V_TIM to indicate that the operating system should check for timeouts from the device unit.
- Determines the timeout due time from the timeout value, now at the top of the stack, and EXE$GL_ABSTIM, and stores the result in UCB$L_DUETIM.
- Clears UCB$V_TIMOUT 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 spinlock in a multiprocessing environment, its caller cannot be executing above IPL$_SYNCH or hold any higher ranked spinlocks. (For instance, a driver fork process executing at IPL$_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 spinlocks 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.

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 spinlock in a multiprocessing environment, its caller cannot be executing above IPL$_SYNCH or hold any higher ranked spinlocks. (For instance, a driver fork process executing at IPL$_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 spinlocks 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 (set to zero).[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 Chapter 2.

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 spinlock in a multiprocessing environment, its caller cannot be executing above IPL$_SYNCH or hold any higher ranked spinlocks. MMG$UNLOCK returns control to its caller at the caller's IPL. The caller retains any spinlocks 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.

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 spinlock and enforces the appropriate IPL synchronization on the local processor.

Module

SPINLOCKS

Macro

FORKLOCK, LOCK

Input

Location	**Contents**
R0	Fork lock or spinlock index

Output

Location	**Contents**
R0	Fork lock or spinlock 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 spinlock 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.

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 spinlock 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 spinlock index

Output

Location	**Contents**
R0	Fork lock or spinlock 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.

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**
R0	Address of device lock

Output

Location	**Contents**
R0	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 spinlock held by the local processor.

Module

SPINLOCKS

Macro

FORKUNLOCK, UNLOCK

Input

Location	Contents
R0	Fork lock or spinlock index

Output

Location	Contents
R0	Fork lock or spinlock 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.

SMP$RESTOREL

Releases a single acquisition of a device lock held by the local processor.

Module

SPINLOCKS

Macro

DEVICEUNLOCK

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 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.

4

Device Driver Entry Points

This chapter describes the standard driver routines and their environment that the operating system uses as entry points in a device driver program. The standard entry routines are:

- Alternate start-I/O
- Cancel-I/O
- Cloned UCB
- Controller initialization
- Driver unloading
- FDT
- Interrupt service
- Register-dumping
- Start-I/O
- Timeout handling
- Unit delivery
- Unit initialization
- Unsolicited interrupt service

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 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

System routines call a driver's cancel-I/O routine under the following circumstances:

- When a process issues a Cancel-I/O-on-Channel system service ($CANCEL)
- When a process deallocates a device, causing the device's reference count (UCB$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 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 R4 and R5.

Input

Location	Contents
R2	Channel index number
R3	Contents of UCB$L_IRP (address of current IRP, if any, for device)
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:	
	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 and R4.

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$W_REFC(R2)	0
UCB$L_STS(R2)	UCB$V_DELETEUCB set, UCB$V_ONLINE set
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$W_FLAGS of template ORB	ORB$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

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.

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

The System Generation utility (SYSGEN) calls a driver's controller initialization routine when processing a CONNECT command. Also, the operating system calls this routine if the device, controller, processor, or adapter to which the device is connected experiences a power failure.

Synchronization

The operating system 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.

The portion of the controller initialization that services power failure cannot acquire any spinlocks. As a result, the routine cannot fork to perform power failure servicing.

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

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.

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

The System Generation utility (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$_POWER or obtain spinlocks, it is of limited usefulness. It cannot safely modify I/O database fields, but can use COM$DRVDEALMEM to return system buffers allocated by the driver to nonpaged pool.

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 spinlocks they may acquire in a 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

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 routine must exit using one of the mechanisms listed in Table 4–1.

Table 4–1 Last FDT Routine Exit 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 system routine that delivers it to the driver.

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 the operating system and do not need to provide their own interrupt service routine.

Called by

The interrupt service routine is called either by the system interrupt dispatcher (for direct-vectored 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 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.

Input

Location	Contents
00(SP)	Address of longword that contains the address of the IDB
04(SP) to 24(SP)	For UNIBUS, Q22–bus, and generic VAXBI devices, the contents of R0 through R5 at the time of the interrupt
28(SP)	For UNIBUS, Q22–bus, and generic VAXBI devices, PC at the time of the interrupt
32(SP)	For UNIBUS, Q22–bus, and generic VAXBI devices, PSL at the time of the interrupt
04(SP) to 16(SP)	For MASSBUS devices, the contents of R2 through R5 at the time of the interrupt
20(SP)	For MASSBUS devices, PC at the time of the interrupt
24(SP)	For MASSBUS devices, PSL at the time of the interrupt

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 the operating system 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 system interrupt service routine performs these functions.

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 system error-logging routines (ERL$DEVICERR, ERL$DEVICTMO, and ERL$DEVICEATTN) and diagnostic buffer filling routine (IOC$DIAGBUFILL) call the register-dumping routine.

Synchronization

The operating system calls a register-dumping routine at the same IPL at which the driver called the system 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 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 multiprocessing environment. An invocation of the WFIKPCH 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 direct-I/O transfer, virtual address of first page-table entry (PTE) of I/O-transfer buffer; for buffered-I/O transfer, address of buffer in system address space

Exit

The start-I/O routine suspends itself whenever it must wait for a required resource, such as a controller data channel or UNIBUS or Q22–bus map registers. To do so, it invokes a system macro (such as REQPCHAN 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 system 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 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 **excpt** 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 **excpt** argument. These macros are used in the driver’s start-I/O routine; thus, strictly speaking, the driver itself is the only entity that uses this entry point.

Routines in the system 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 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 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.

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, the operating system 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

The System Generation utility (SYSGEN) command AUTOCONFIGURE 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 or Q22–bus devices); address of MBA configuration register (MASSBUS devices)
R7	Address of AUTOCONFIGURE command's configuration control block (ACF)
R8	Address of ADP

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.

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

The System Generation utility (SYSGEN) calls a driver's unit initialization routine when processing a CONNECT command. The operating system 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

The operating system 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 spinlocks. As a result, the routine cannot fork to perform power failure servicing.

Context

Because the operating system 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.

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.
8. For multiunit VAXBI controllers, tests for the existence of the unit for which it was called and returns success or failure status to SYSGEN.

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_UNSOLINT.

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.

Index