RSX-11M-PLUS and Micro/RSX Guide to Writing an I/O Driver

Order No. AA-H267C-TC

RSX-11M-PLUS Version 4.0
Micro/RSX Version 4.0

Digital Equipment Corporation Maynard, Massachusetts

First printing, October 1979
Revised, March 1982
Revised, September 1987

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

The following are trademarks of Digital Equipment Corporation:

DEC	EduSystem	UNIBUS
DEC/CMS	IAS	VAX
DEC/MMS	MASSBUS	VAXcluster
DECnet	MicroPDP-11	VMS
DECsystem-10	Micro/RSX	VT
DECSYSTEM-20	PDP	
DECUS	PDT	
DECwriter	RSTS	digital
DIBOL	RSX	

ZK3088

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*	CANADA	INTERNATIONAL
Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario K2K 2A6 Attn: Direct Order Desk	Digital Equipment Corporation PSG Business Manager c/o Digital's local subsidiary or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.
In New Hampshire, Alaska, and Hawaii call 603-884-6660.
In Canada call 800-267-6215.
*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).
Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

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

Contents

Chapter 2 Device Driver I/O Structures

Chapter 3 Executive Services and Driver Processing

Chapter 4 Programming Specifics for Writing an I/O Driver

Chapter 5 Incorporating a User-Supplied Driver Into RSX-11M-PLUS

Chapter 6 Debugging a User-Supplied Driver

Chapter 7 Executive Services Available to an I/O Driver

Chapter 8 Sample Driver Code

Appendix A Converting A User-Supplied RSX-11M Driver

Figures

Tables

Preface

Manual Objectives

The primary goals of this manual are to introduce RSX-11M-PLUS physical I/O concepts, define Executive and I/O service routine protocol, describe system I/O data structures, and prescribe I/O service routine coding procedures. This information is intended to allow you to perform the following kinds of operations:

- Prepare software that interacts with the Executive and supports a conventional I/O device
- Incorporate user-written software into an RSX-11M-PLUS or Micro/RSX system
- Detect typical errors that cause the system to fail
- Use Executive service routines that an I/O service routine frequently employs

Secondary objectives are to introduce advanced hardware and software features and sophisticated Executive facilities, and to describe both the conventional and advanced features of I/O data structures and mechanisms. This discussion of advanced features can enhance your understanding of conventional I/O processing and data structures.

This manual does not describe how to write software that incorporates advanced driver features. The only complete package of such information is DIGITAL-supplied software, such as DVINT.MAC and DBDRV.MAC (for overlapped seek, dual access, and common interrupt handling); IOSUB.MAC and TTDRV.MAC (for full duplex I/O); and MMDRV.MAC (for subcontroller device operation). The manual also does not describe how to attach hardware to the PDP-11, how to perform diagnostic functions to uncover hardware faults, nor how to incorporate DIGITAL-standard error-reporting functions in user-written software.

Intended Audience

This manual is written for the senior-level system programmer who is familiar with the hardware characteristics of both the PDP-11 and the device that the user-written software supports. The programmer should also be knowledgeable about DIGITAL peripheral devices and experienced in using the software supplied with an RSX-11M-PLUS system. The manual does not describe general Executive concepts nor does it define general system structures. The manual does describe I/O concepts, the Executive role in processing I/O requests, and some pertinent aspects of I/O processing done by DIGITAL-supplied software. With a firm understanding of hardware characteristics and real-time system software, a senior-level system programmer can appreciate how the interaction of user-written software with the Executive also affects overall system performance.

Structure of This Document

This document is designed to provide the conceptual, procedural, and reference information you need to build and incorporate a user-written driver into your system.

Chapter 1 introduces terms and concepts fundamental to understanding physical I/O in RSX-11M-PLUS and describes the protocol that a driver must follow to preserve system integrity. It summarizes advanced driver features and RSX-11M-PLUS capabilities to help you become acquainted with overall Executive and driver interaction.

Chapter 2 continues the conceptual discussion begun in Chapter 1. It introduces on a general level the software data structures involved in handling I/O operations at the device level, examines typical arrangements of data structures that are necessary for controlling hardware functions, and presents a macroscopic software configuration that summarizes the logical relationships of the I/O data structures.

Chapter 3 ends the conceptual presentation. It summarizes how an I/O request originates, how the Executive processes the request, and how a driver uses Executive services to satisfy an I/O request.

Chapter 4 provides the following reference information necessary to code a conventional I/O driver:

- A summary of programming standards and protocol
- An introduction to the programming facilities and requirements for both the driver data base itself and the executable code that constitutes the driver
- An extensive elaboration of the driver data base and of the driver code

Chapter 5 supplies the procedural information you need to assemble and build a loadable driver image, load it into memory, and make accessible the devices that the driver supports. Also included are a summary of the system generation dialogue concerning including user-supplied drivers and a description of the loading mechanism and the diagnostic operation performed during loading.

Chapter 6 summarizes software features provided to help you uncover faults in drivers and gives procedures to follow that might prove successful in isolating faults in drivers.

Chapter 7 gives general coding information relating to the PDP-11 and RSX-11M-PLUS Executive service routines.

Chapter 8 shows the source code for the data base and driver of a conventional device and an excerpt of source code from a driver that handles special user buffers.

Appendix A describes the modifications you must make to enable an RSX-11M user-supplied driver to run on an RSX-11M-PLUS or Micro/RSX system.

Associated Documents

Your RSX-11M-PLUS system comes with documents that describe both the software and hardware on the system. The software documents are listed and described in the *RSX-11M-PLUS Information Directory and Master Index*. Consult the directory for concise summaries of software-related publications. Processor and peripherals handbooks summarize hardware information published in various maintenance, installation, and operator manuals that are provided with your system.

Consult the *RSX-11M-PLUS and Micro/RSX Crash Dump Analyzer Reference Manual* for the "System Data Structures and Symbol Definitions". The *RSX-11M-PLUS and Micro/RSX Crash Dump Analyzer Reference Manual* lists the source code of system macro calls that define system device structures and driver-related structures, as well as the system-wide symbolic offsets needed to access those structures.

Conventions Used in This Document

The following conventions are used in this manual:

Convention	Meaning
>	A right angle bracket is the default prompt for the Monitor Console Routine (MCR), which is one of the command interfaces used on RSX-11M-PLUS systems. All systems include MCR.
$	A dollar sign followed by a space is the default prompt of the DIGITAL Command Language (DCL), which is one of the command interfaces used on RSX-11M-PLUS and Micro/RSX systems. Many systems include DCL.
xxx>	Three characters followed by a right angle bracket indicate the explicit prompt for a task, utility, or program on the system.
UPPERCASE	Uppercase letters in a command line indicate letters that must be entered as they are shown. For example, utility switches must always be entered as they are shown in format specifications.
command abbreviations	Where short forms of commands are allowed, the shortest form acceptable is represented by uppercase letters. The following example shows the minimum abbreviation allowed for the DCL command DIRECTORY: `$ DIR`

Convention	Meaning
lowercase	Any command in lowercase must be substituted for. Usually the lowercase word identifies the kind of substitution expected, such as a filespec, which indicates that you should fill in a file specification. For example: `filename.filetype;version` This command indicates the values that comprise a file specification; values are substituted for each of these variables as appropriate.
/keyword, /qualifier, or /switch	A command element preceded by a slash (/) is an MCR keyword; a DCL qualifier; or a task, utility, or program switch. Keywords, qualifiers, and switches alter the action of the command they follow.
parameter	Required command fields are generally called parameters. The most common parameters are file specifications. Commas are used as separators for command line parameters and to indicate positional entries on a command line. Positional entries are those elements that must be in a certain place in the command line. Although you might omit elements that come before the desired element, the commas that separate them must still be included.
[g,m] [directory]	The convention [g,m] signifies a User Identification Code (UIC). The g is a group number and the m is a member number. The UIC identifies a user and is used mainly for controlling access to files and privileged system functions. This may also signify a User File Directory (UFD), commonly called a directory. A directory is the location of files. Other notations for directories are: [ggg,mmm], [gggmmm], [ufd], [name], and [directory]. The convention [directory] signifies a directory. Most directories have 1- to 9-character names, but some are in the same [g,m] form as the UIC. Where a UIC, UFD, or directory is required, only one set of brackets is shown (for example, [g,m]). Where the UIC, UFD, or directory is optional, two sets of brackets are shown (for example, [[g,m]]).
. . .	A vertical ellipsis shows where elements of command input or statements in an example or figure have been omitted because they are irrelevant to the point being discussed.
boldface	Boldface type indicates that the term is being defined; for example, command parameters and glossary entries.
italics	Italic type indicates emphasis, a book title, or that a new term is being used in the text.

Convention	Meaning
"print" and "type"	As these words are used in the text, the system prints and the user types.
black ink	In examples, what the system prints or displays is printed in black.
red ink	In interactive examples, what the user types is printed in red. System responses appear in black.

Summary of Technical Changes

The following sections list features, qualifiers, error messages, and restrictions that are new to user-written drivers for the RSX-11M-PLUS and Micro/RSX Version 4.0 operating systems. These new features are documented in this revision of the *RSX-11M-PLUS and Micro/RSX Guide to Writing an I/O Driver.*

New Features

The *RSX-11M-PLUS and Micro/RSX Guide to Writing an I/O Driver* has the following new features:

- Vectoring for privileged tasks and the Executive
- Converting to vectored access in privileged tasks
- Converting to vectored access in drivers
- Overlapped I/O completion

  Vectoring allows access to Executive code by privileged tasks.

  Conversion procedures allow existing privileged tasks vectored access to the Executive.

  A conversion routine allows you to convert existing drivers for vectored access to the Executive.

  Overlapped I/O completion means that the Executive will process the next I/O request while a physical device services the current I/O request; thus speeding up I/O operations.

Changes to the Document

- "System Data Structures and Symbolic Definitions," which was Appendix A, has been deleted from this manual and now appears in the *RSX-11M-PLUS and Micro/RSX Crash Dump Analyzer Reference Manual.* Appendix A is now "Converting a User-Supplied RSX-11M-PLUS Driver".

  A discussion of ASTs has been added.

  The description of the $DEUMR routine has been modified.
- A routine description has been added for $IODSA.

- Additional information about the suc argument has been provided.
- Additional information about the K.OWN function has been provided.
- The description of US.FOR in Figure 4-7 has been modified.
- The description of US.PUB in the UCB field U.ST2 has been modified.
- The address description of U.BUF (for a MASSBUS or 22-bit Q BUS NPR device) has been modified.
- The S.RCNT field in Figure 4-9 has been modified.
- The following additional input for the $ASUMR routine has been documented:

```
M.BFVL(R0)=LOW ADDRESS OF TRANSFER (FOR ODD/EVEN BYTE DETERMINATION)
```

- The reference to the file specification for the sample driver code on page 8-1 has been changed from "UFD [200,1]" to "directory [USER]".
- The description of SCBDF$ has been modified in the sample driver data base.
- Differences between incorporating user-supplied drivers on RSX-11M-PLUS and Micro/RSX are discussed.

Chapter 1

RSX-11M-PLUS I/O Drivers

Device drivers on RSX-11M-PLUS are the primary method for creating an interface between the Executive software and the hardware attached to the computer.[1] Most DIGITAL-supplied hardware is supported by drivers accompanying the software the user receives with the system. This chapter introduces the concept of device drivers and explains driver operations and features.

1.1 Vectors and Control and Status Registers

A device controller has a unique address on the PDP-11 UNIBUS that identifies and distinguishes it from other hardware attached to the computer. A control and status register (CSR) containing data elements that allow software to operate and interrogate the related device is usually located at this address. The CSR resides in physical address space that is reserved for device registers and is referred to as the I/O, or peripheral, page. Other registers associated with the device are placed in contiguous addresses lower or higher than the CSR address. Software usually controls a device by accessing the CSR to enable interrupts, initiate a function, and respond to the resulting interrupt to continue or finish the function.

Many devices can have one or more two-word areas, called interrupt vectors, associated with them. A vector provides a connection between the device and the software that services the device and allows the device to trigger certain software actions because of some external condition related to it. When a device interrupts, it sends the processor the address of the interrupt vector. The first word of the interrupt vector contains the address of the interrupt service routine for that device. The processor uses the second word of the vector as a new Processor Status Word (PSW). Thus, when the processor services the interrupt, the first word of the vector is taken as the new Program Counter (PC) and the second word is the new PSW.

Space is reserved on the PDP-11 for the interrupt vectors. This space is in the low part of kernel I-space. The vectors are considered to be in kernel mode virtual address space and are thus mapped by the Executive. Because the interrupt vector is in kernel space, the Executive receives control of the processor on every interrupt. On a multiprocessor system, each central processor unit has its own vector space.

[1] The CINT$ directive provides a secondary but equally viable interface between software and hardware.The CINT$ directive enables a privileged task to gain control when a device interrupts and thereby to access device registers. The K-series Laboratory modules use this feature to perform I/O.

1.2 Service Routines

The service routine that is entered to process an interrupt is most frequently in the device driver. Device drivers vary in complexity depending on the capabilities of the type of device and the number of device units they service. A driver can reside within the Executive (a resident driver) or separate from the Executive (a loadable driver).

The distinction between resident and loadable drivers is mainly one of flexibility. A resident driver is built in during system generation as a permanent part of the Executive. It resides in the Executive address space and cannot be removed. A resident driver responds to interrupts slightly faster than a loadable driver.

Although linked into the Executive structures, a loadable driver resides in memory outside the virtual address space of the Executive. A user can add or remove a loadable driver by means of a Monitor Console Routine (MCR) or Virtual Monitor Console Routine (VMR) command. In addition, any driver not required for a period of time need not be loaded. The space normally occupied by the unloaded driver can hold user tasks or another driver. On systems with Executive data space support, all drivers must be loadable. On a system without Executive data space support, making a driver loadable frees virtual space in the Executive, which can be used for additional pool.

1.2.1 Executive and Driver Layout

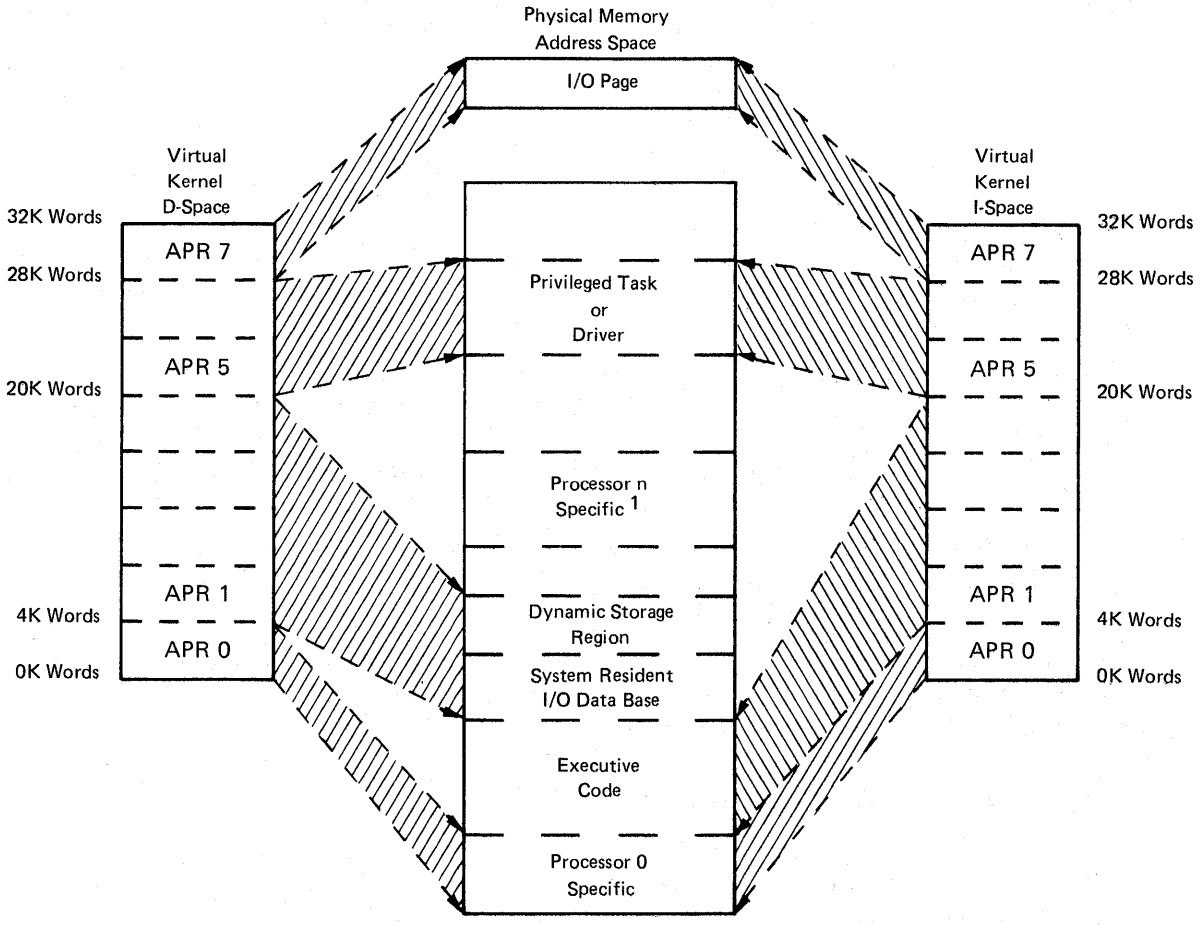
A device driver is a logical extension of the Executive that is not necessarily contiguous in physical memory with the Executive code. Active Page Registers (APRs) 0 through 4 map the Executive, whereas APR 7 is reserved to map the I/O page.[1] Resident drivers are mapped within the Executive space. Loadable drivers reside in a separate partition of memory and are mapped by APR 5. Therefore, a loadable driver is restricted by default to the 4K words of space mapped by APR 5, unless it controls its own mapping with APR 6 to gain access to an extra 4K words.

The virtual-to-physical mapping on a system with kernel data space support is shown in Figure 1–1.

Virtual addresses 0 through 4K words (APR 0) of I and D space overmap the same physical memory. The mapped area contains the interrupt vectors, processor stack, processor-specific memory locations, and interrupt control block (ICB) pool space as well as some Executive code. I-space virtual addresses 4K through 20K words (APR 1 through APR 4) map the remaining Executive code, which is therefore limited to 16K words. D-space virtual addresses 4K through 20K words (APR 1 through APR 4) map the dynamic storage region (or pool) and system data structures to a maximum of 16K words.

[1] Active Page Register is a term referring to the KT11 Memory Management register pair (Page Address Register (PAR) and Page Descriptor Register (PDR).)

Figure 1–1: Virtual to Physical Mapping for the Executive



[1]On multiple processor systems, each additional processor requires its own processor-specific area in the CPU partition.

ZK-245-81

Virtual addresses 20K through 28K words (APR 5 and APR 6) of I and D space overmap the same physical memory, which is reserved to map loadable drivers and privileged tasks in kernel mode. (Although APR 5 and APR 6 are reserved for drivers, the Executive maps only APR 5 when it calls a driver.) Finally, virtual addresses 28K through 32K words (APR 7) of I and D space overmap the I/O page.

Thus, a device driver is mapped with the Executive code and the I/O page. When a driver has control, it can access the device registers in the I/O page to perform its operations. It also has available all the Executive service routines to help it process I/O requests.

Because of the layout of the Executive and device drivers, many common functions related to I/O are centralized in the Executive as service routines. This commonality eliminates the need for repetitive coding in each and every driver. Coding in each driver is therefore reduced to handling the specific functions of the device supported.

1.2.2 Driver Contents

A device driver consists of two parts. One part is the executable instructions of the driver itself. This part has the entry points to the driver. The entry points are those places where the Executive calls the driver to perform a specific action, and their addresses are established in the driver dispatch table (DDT). The table contains addresses of routines in a fixed order so that the Executive can enter the driver at the appropriate place for a given action.

The other part of a device driver is the data structures forming the data base that describes the controllers and units supported by the driver. Two structures, the controller table (CTB) and the controller request block (KRB), describe the controller of the device being supported. Because the CTB supplies generic information about the controller type, only one CTB need exist for each controller type on a system. The KRB holds information related to a specific controller and therefore each controller has its associated KRB.

Three structures in the driver data base—the device control block (DCB), the unit control block (UCB), and the status control block (SCB)—describe the device as a logical entity. The DCB contains information related to the type of device, whereas the UCB holds information specific to an individual unit of the device. The SCB is used mainly to store data (driver context) concerning an operation in progress on the device unit.

The code of a driver must be in one continuous portion of main memory. Because the Executive is designed to respond to real-time activities, the driver code must run as fast as possible. Therefore, it cannot be overlaid.

The driver data structures are tailored to the number of controllers on the system, the number of units attached to each controller, and the types of features the devices support. The structures increase in complexity as the number of supported features increases.

1.3 Executive and Driver Interaction

The Executive and a driver interact by accessing and manipulating common data structures. An I/O activity typically begins when a task generates a request for input or output. The Executive performs preliminary processing of that request before it initiates the driver. This preliminary processing, called predriver initiation, is common for all drivers and eliminates a great deal of code from all drivers.

In performing predriver initiation, the Executive accesses the driver data structures to assess the legality of the I/O request. However, the Executive calls the driver only when the predriver initiation warrants it. For example, cells in the device control block (DCB) define the functions that the driver supports. If the function specified in the I/O request is not supported by the driver, the Executive does not call it, and the driver is not aware of the I/O request. For this reason, the Executive calls the driver only when the predriver initiation warrants it.

1.3.1 The Driver Process

When the Executive does call the driver to process an I/O request, the driver begins I/O initiation. Once an I/O request is created, a driver process is initiated. The Executive queues an I/O packet to the driver that must be processed to satisfy the request. Potentially, the system has as many driver processes as there are distinct units capable of being active simultaneously. (Moreover, some drivers supporting advanced features can have multiple I/O requests simultaneously active for a given unit. In this case, each active I/O request is part of a separate driver process. Refer to Section 1.4.7 for more information.)

The difference between a driver process and the driver code is central to a full understanding of a driver and the I/O structure. The driver code, which is pure instructions, invokes an Executive routine called $GTPKT to get an I/O packet to process. This activity generates data for the request being processed and for the unit doing the processing. Once initiated, the driver process starts the proper I/O function, waits for a completion interrupt, posts I/O status, and requests another I/O packet. This sequence of execution steps continues until the I/O queue is empty. The driver process then terminates.

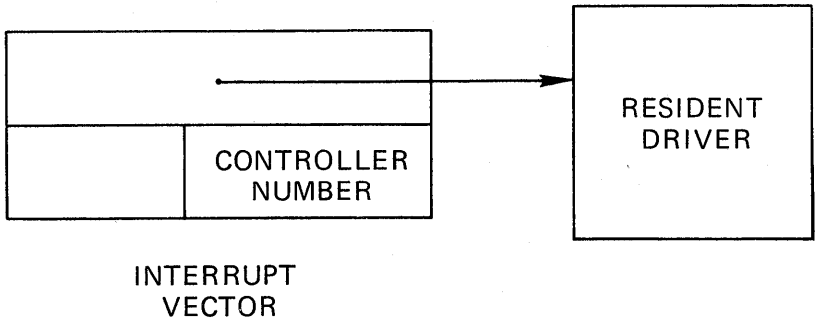
Because a driver may be capable of servicing several I/O requests in parallel, it is possible that, for a single driver, many driver processes exist at the same time. However, there is only one copy of driver code. The driver process is reentrant code, and the data that defines the state of the code is stored in the driver data base when the process is not executing (for example, when it is waiting for an interrupt). The driver process executes driver code for a particular device type on behalf of a specific unit. If independent units of a particular device type are concurrently active, then several driver processes are also active at the same time, each with its own set of data.

1.3.2 Interrupt Dispatching and the Interrupt Control Block

Once a driver starts an I/O function, it must await the I/O completion interrupt. When a device interrupt occurs, the processor pushes the current PSW and PC onto the current stack and loads the new PSW and PC from the device controller interrupt vector. By convention, the PSW in the interrupt vector is preset with a priority of 7 and with the number of the controller associated with the vector. (The controller number is in the low-order four bits.)

Because an interrupt must be serviced in kernel address space, how the interrupt is handled depends on whether the driver is resident or loadable. A resident driver, being mapped with the Executive in kernel address space, handles the interrupt directly (that is, the entry point address of the driver is the PC word of the interrupt vector). For a resident driver, then, the hardware dispatches directly to the interrupt service routine in the driver. Figure 1–2 shows this mechanism.

Figure 1–2: Interrupt Dispatching for a Resident Driver



When the interrupt service routine in the resident driver gains control, it runs at priority 7, which locks out further interrupts. The driver is therefore uninterruptable and, because the system must respond to real-time events, processing at this level cannot take too long.[1]

To ensure that a driver does not lock out other interrupts on the system or destroy the context of any interrupted process, a protocol has been established. By system convention, no process should run at an uninterruptable level for more than 100 microseconds. A common Executive coroutine, called interrupt save ($INTSV), exists to lower the priority level of the driver process to that of the interrupting device and to save two registers of the interrupted process. Therefore, by system convention, all resident drivers call the $INTSV coroutine, which saves the PSW and extracts the controller number. Because most instructions change the PSW bits that encode the controller number, under most circumstances the driver can do very little else without saving the controller number.

The $INTSV coroutine saves Registers 4 and 5, which are thereafter free for the driver to use. These registers are typically used by drivers to hold addresses of the data blocks containing unit status and control information, the SCB and UCB. (Most Executive routines assume these two registers hold pointers to the two structures. If the driver needs to use more registers, it saves them on the stack and restores them when it finishes.) When the interrupt save coroutine returns to the driver, the driver runs at the interrupt level of the device it is servicing and has two free registers it can use. This protocol makes the driver partially interruptable (that is, interruptable by devices with a higher priority) and preserves the context of the interrupted process.

The driver may then run for a short interval at the partially interruptable level. By convention, this interval should not exceed 500 microseconds. When the driver finishes processing the interrupt, it may execute a RETURN instruction to transfer control back to the coroutine, which gives control of the CPU to the next process.[2]

[1] On a multiprocessor system, a driver running at priority 7 is interruptable by a device of the same type on another CPU. To handle this situation, the driver being interrupted does not have to do any special processing beyond what is described in this manual.

[2] An executive interrupt exit routine, $INTXT, exists to standardize the way a driver exits from an interrupt. However, on RSX-11M-PLUS systems this routine is simply a RETURN instruction.

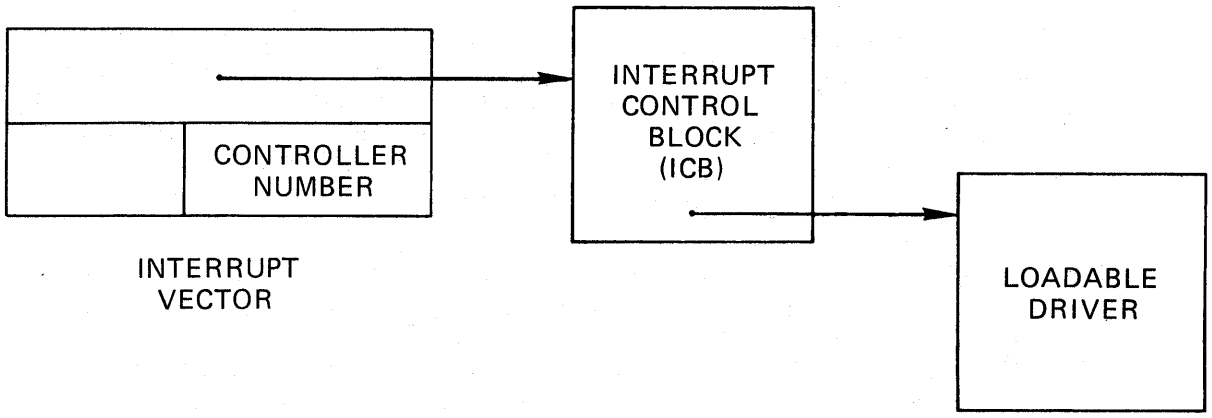
For a loadable driver, the hardware cannot dispatch directly to the interrupt service routine in the driver because the driver is mapped outside the address space of the Executive. Therefore, some code in the Executive must initially handle the interrupt, load the mapping context of the driver, and dispatch to the proper driver. This code resides in the Executive in a structure called an interrupt control block (ICB). Figure 1–3 shows this mechanism.

The ICB actually contains a Jump to Subroutine (JSR) instruction to an Executive interrupt save routine ($INTSI) and some data cells that enable the routine to do the following:

- Save Registers 4 and 5
- Save the kernel mapping (APR 5)
- Load APR 5 to map the driver
- Transfer control to the driver
- Restore the mapping after return from the driver
- Restore Registers 4 and 5

Thus, the interrupt vector for a controller serviced by a loadable driver points to an ICB rather than to the driver. Accordingly, the loadable driver does not (and must not) call the $INTSV routine as the resident driver does because the $INTSI routine saves the context on behalf of the loadable driver. When it gains control, the loadable driver is also partially interruptable as if it had called the $INTSV routine. After it gains control, the loadable driver is exactly like the resident driver; that is, it must also observe the protocols established on the system.

Figure 1–3: Interrupt Dispatching for a Loadable Driver



The ICB allows up to 128 controllers of the same type on a system. The low-order four bits in the PSW of the interrupt vector restricts the number of controllers to 16. In the ICB, the system maintains a controller group number and the PSW bits describe the controller number within the group. To obtain the real controller number, the Executive interrupt service routine adds the controller group number in the ICB and the controller number in the PSW. (Note that because a resident driver does not use the ICB mechanism, there can be at most 16 controllers of one type if the driver is resident. Furthermore, only the LOAD command in VMR supports more than 16 controllers of one type.)

In handling an interrupt, the simplest case is where a controller can have only one unit active at any one time. Multiple controllers may be active concurrently, yet only one unit per controller may be active. When an interrupt occurs, the driver can determine the saved controller number from information encoded in the low-order four bits of the PSW. The interrupt service routine in the driver uses the number to index a table in the CTB and to access the proper unit data and context.

The more complex case in dispatching an interrupt is where a controller can have multiple units operating in parallel. This is an advanced driver feature called overlapped seek I/O and is described in Section 1.4.1.

1.3.3 Interrupt Servicing and Fork Process

A driver (whether resident or loadable) handling an interrupt and operating at the partially interruptable level may need to (1) access structures in its data base or (2) call centralized Executive service routines that may access structures in the data base. Because a driver may have more than one process active simultaneously, the driver itself may need to access structures in the data base shared among separate, unrelated processes. A method must exist to coordinate access to the data structures shared among the processes and the Executive.

The mechanism that coordinates access to the shared structures is called the fork process. The Executive routine $FORK causes the driver to be placed in a queue of processes waiting for access to the shared data structures, to run at processor priority level 0, and to be completely interruptable.[1] Therefore, a driver must call the fork routine before it calls any other Executive service routine (except for $INTSV), or before it accesses any device-specific (nonprivate) structures in its data base. If a driver does not follow this protocol, it will corrupt the system data base and will eventually cause a system to fail.

By calling the fork routine, a driver requests the Executive to transform it into a fork process. The routine saves a snapshot of the process in a fork block. The snapshot is the context of the driver process—the PC of the process and the contents of Registers 4 and 5. The fork block itself resides in the I/O data structure that holds the status information of the device being serviced (that is, the status control block, or SCB). The Executive maintains a list of fork blocks in first-in-first-out (FIFO) order—a new fork block is added to the list after the last block in the list.

When the driver calls $FORK, the CPU priority is lowered to 0, which allows other interrupts to be serviced. When there are no more pending interrupts (either they have been dismissed or the drivers have called $FORK), the Executive checks to see whether the first interrupt preempted a priority 0 Executive process. If a preemption occurred, the Executive process is continued from where it was interrupted. If no priority level 0 Executive process was interrupted, the Executive executes the process at the head of the fork list. The Executive restores the saved context of the process from the SCB and returns control to the driver at the statement immediately following the call to the fork routine. The process is unaware that a pause of indeterminate length has elapsed.

[1] By convention, drivers may operate at a partially interruptable level for no more than 500 microseconds. Some drivers conceivably could need more time than this convention allows. Thus, an additional reason for the fork mechanism is to preserve the response time of the system and not lock out interrupts from lower-priority levels.

Fork processes are thereby granted FIFO access to the common I/O data structures. Once granted such access, a fork process has control of the structures until it exits. The protocol guarantees that the driver process has unrestricted access to shared system data structures. As one fork process exits, the next in the list is eligible to run and access the data structures. Thus, the fork mechanism allows both controlled access to the common data structures and sufficient time to process an interrupt without locking up the system.

The status of a fork process lies between an interrupting routine and a task requesting system resources. Interrupt routines are run first and can be interrupted only by higher-priority interrupts. Processes in the fork list run after other system processes either terminate or call $FORK themselves. Because system processes save and restore registers, a fork process can use all registers. The fork processes are completely interruptable. Tasks run only when the fork list is empty.[1]

The fork mechanism establishes linear, or serial, access to the shared data structures. For example, an Executive routine that completes I/O processing ($IODON) manipulates the I/O queue to deallocate an I/O packet that the driver processed. If Multiple processes were allowed to alter the queue at random times, the queue pointers could become disarranged. Without the fork mechanism, any process could be interrupted by a higher-priority process and not be able to complete. Because the Executive completes a currently active fork process before it starts the next fork process in the queue, the integrity of the I/O data structures is maintained, if all routines that call $IODON run at fork level.

Between the time that a driver process calls $FORK and the Executive starts the process at fork level, the driver cannot call $FORK again for that same device. If the $FORK routine is called again before the first process starts, context stored in the fork block for the first fork process is overwritten. However, once a fork process starts, the data in the fork block is stale and the process may call $FORK again while it is at fork level. If the driver does not ensure against unexpected interrupts, it may double fork as described above. As a result of the double fork, the driver may either miss an interrupt from the device or miss interrupts from several devices. As a further consequence, code encountered after the call to $FORK is executed twice for the same context with generally catastrophic results. For example, calling $IODON twice for the same I/O packet eventually causes the system to fail.

If all drivers adhere to the interrupt protocol, the integrity of the I/O data structures is preserved. Thus, when a device interrupt occurs while a fork process is executing, the protocol demands that the service routine handling the interrupt not destroy any of the registers. The registers are part of the context of the fork process. After the driver dismisses the interrupt or becomes a fork process itself, the interrupted fork process can safely resume execution with its proper context. If any driver violates the protocol, the integrity of the I/O data structures is endangered. (That is, the system fails in mysterious ways.)

[1] On a multiprocessor system, the fork list is not necessarily empty when the Executive returns control to a task. The Executive processes only those fork blocks that are to run on the current processor. To ensure that fork blocks remaining in the list are readily processed, the Executive running on one processor interrupts (using the interprocessor interrupt hardware) any other processor that has fork blocks waiting for processing.

1.3.4 Nonsense Interrupt Entry Points

The addresses of Executive nonsense interrupt entry points are contained in all vectors for off-line devices and in vectors for which there are no devices. Code at these special entry points exists to properly dismiss unexpected interrupts from these devices. If error logging is active, any unexpected interrupts are recorded as undefined interrupt errors. This feature helps in detecting faulty hardware.

1.4 Advanced Driver Features

Advanced drivers have certain optional and built-in special features. This section introduces these features to help you better understand the structures described in the remainder of the manual.

1.4.1 Overlapped Seek I/O

Some disk devices allow multiple device units attached to the same controller to execute operations in parallel. This is called overlapped seek support and is a software option designed to take advantage of a hardware feature found in most advanced disk drives. This feature allows all drives attached to the same controller to execute a seek function simultaneously. Each unit may perform a seek operation independent of what another unit may be doing. Only one data transfer can occur at any one time. Some types of drives allow seek functions to overlap a data transfer function, whereas other types do not.

The increased difficulty for overlapped seek devices stems from determining whether the controller or the unit generated the interrupt. Most control functions issued to the drive unit (including the positioning commands SEEK and SEARCH) terminate with a unit interrupt. The controller reports the physical unit number of the interrupting unit in its attention summary register. A controller interrupt indicates the termination of a function (usually a data transfer command) that changes the controller status from "busy" to "ready". Only one unit may issue a data transfer complete notification to a particular controller at any one time, because only one data transfer can be in progress at any one time. Most hardware defers seek termination interrupts until the current data transfer is complete.

To handle interrupts for a device that supports overlapped seek operations, a device-specific interrupt service routine built into the Executive examines the device registers to determine whether the interrupt was initiated by the controller or the drive unit. Using the controller number retrieved from the PSW in the interrupt vector, the routine forms an index (called the controller index) to use as an offset into a table of addresses in a structure (called the controller table or CTB) in the I/O data base. The routine accesses the table to determine the address of the I/O data structure of the controller (called the controller request block or KRB) that generated the interrupt. Accessing the KRB yields the address of the CSR of that controller and having the CSR address allows the routine to examine the device registers.

If the controller itself initiated the interrupt, the routine determines the data base structure of the unit that is active. This determination is possible because such a controller interrupt relates to a termination of a data transfer, and only one such unit can be active for a data transfer. A cell in the KRB has the address of the data structure describing the active unit (the unit control block or UCB). The routine can then determine the address of the driver dispatch table and transfer control to the driver.

If a device unit initiated the interrupt, the routine retrieves its unit number from the Attention Summary Register. Using the physical unit number, the routine indexes a table at the end of the KRB to yield the address of the related UCB. The driver is entered through the driver dispatch table.

1.4.2 Overlapped I/O Completion

In general, overlapped I/O completion support causes the execution of the Executive's I/O completion code for each I/O request to be postponed until the next request has been initiated. If I/O requests are in the driver's queue, this action causes the Executive to complete the I/O processing while the physical device services the next request. This feature speeds up I/O operation. A minor side effect is that multiple I/O requests to the same device may complete in an order other than the issued order.

When a driver requests the Executive to complete the I/O request, the Executive checks the queue of requests to the driver. If the queue is not empty, the Executive defers I/O completion by placing the current completion at the end of the fork list. Control returns to the driver, which assumes that the Executive has completed I/O processing. The driver can then initiate the next I/O operation. After the driver has initiated an I/O operation on the device, the driver returns to the Executive and fork processing begins. Thus, the I/O completion for the previous I/O can be processed to the end. When a hardware I/O operation completes, the driver receives an interrupt and then forks, which always causes the I/O completion to occur after the previously queued completion.

The exception to sequential completion occurs when an I/O operation does not require a hardware operation. In this case, the driver processes the I/O request and then calls the Executive's I/O completion routine. If there is an additional I/O request waiting on the queue, this completion also goes at the end of the fork list, and sequential processing is maintained. Nonsequential completion occurs when the very last request in the queue is a nonhardware-oriented I/O. In this case, that I/O request completes prior to any others waiting for completion. There are many cases of nonhardware-oriented I/O, such as inquiries into device state or attach and detach requests.

There is one case where overlapped I/O completion causes I/O requests to finish in an order other than the order in which they were issued. This occurs when the last I/O request queued to the controller does not require a device interrupt to complete.

1.4.3 Dual-Access Support

Some devices have multiple-access paths for both control and data transfer functions. Such devices are called dual access. A dual-access unit is connected to two controllers at one time and may be accessed from either controller at the option of the system software. Since a single device unit may have only one physical unit number, a dual-access unit must have the same unit number for both controllers. A dual-access unit may be accessed only from one port at a time. The system supports dual-access operation for those devices equipped with the necessary hardware capability. This feature is most useful on a multiprocessor system where each access path is to a different central processor unit.

To support dual-access operations, the I/O data structures must reflect the existence of alternate controllers. Particularly, the driver process context for I/O on a unit can be associated with either of two controllers. To decide which controller will provide access to the drive unit, the driver must call an Executive routine to request access to a particular controller. When the Executive grants access, the driver process context for a unit is associated with the assigned controller. A driver must have access to the assigned controller before actually changing the registers in the I/O page.

When a driver and a unit are given access to a controller, the controller status is set to busy. The unit becomes the device owned by the controller for the operation. A controller without an owned unit is considered a free controller. By this ownership mechanism, controller interrupts are sent to the correct unit for processing. After the operation completes, the driver requests the Executive to release the controller and thus frees it.

1.4.4 Delayed Controller Access

Drivers that support overlapped seeks also must request access to a controller before executing a function on an independent unit and must release access after completing the function. To take maximum advantage of simultaneous operation of units on one controller, the system delays controller access when the controller is busy.

The Executive maintains a request queue for the controller. Whenever a driver process requests access to a controller and must wait for access to the controller, the Executive places the associated fork block in the controller request queue. When a driver releases a controller, the Executive automatically grants access to the next driver process waiting for access. Precedence is given to positioning requests over requests for data transfer. The controller request queue thereby provides the means for the Executive to synchronize access.

1.4.5 Controller Reassignment and Load Sharing

Controller assignment for dual-access devices is dynamic. If one port (access path) to a device is busy, the system can request access on the other port. This switching between ports allows the system to share the load between the two controllers.

Note

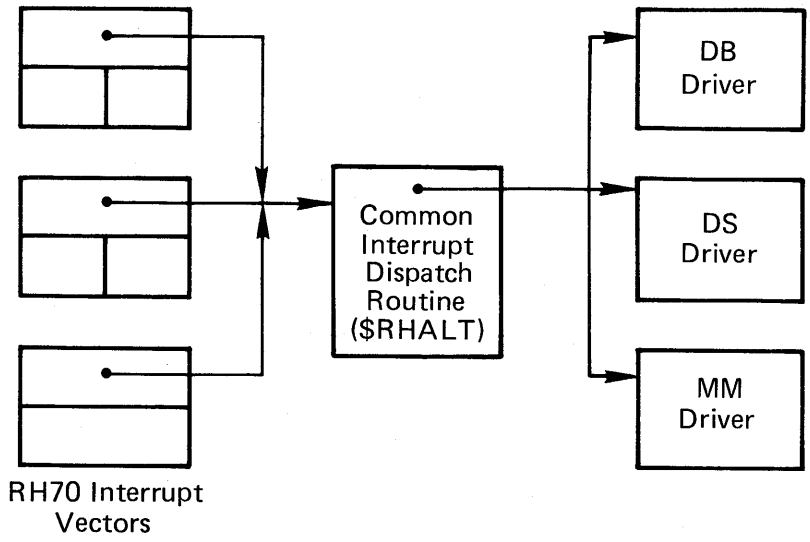
A dual-access device has both ports attached to the same system. DIGITAL does not support systems loosely coupled through a peripheral.

The system also maintains an I/O count to determine how busy each controller is. If one controller is not as busy as the other, the system can queue the access requests to the less busy controller. Whenever load sharing is done on a dual-access unit, the Executive makes any reassignment necessary before actually requesting access to the controller.

1.4.6 Common Interrupt Dispatching

To handle interrupts from a controller that supports more than one type of device, the Executive uses a mechanism called common interrupt dispatching. The RH70 MASSBUS controller can have different types of devices connected to it (RP04, RP05, and RP06 moving head disks; RM02, RM03, and RM05 moving head disks; RM80 and RP07 fixed-media disks; ML11 non-rotating memory; RS03 and RS04 fixed-head disks; and TE16, TU445, and TU77 magnetic tape drives). Interrupt dispatching for such devices is more difficult than for standard interrupt devices because multiple drivers are associated with one set of interrupt vectors. Thus, to dispatch interrupts a routine in the Executive must intervene. Figure 1–4 shows an example of common interrupt dispatching.

Figure 1–4: Interrupt Dispatching for Common Interrupt Devices



The vectors for such controllers point to a common interrupt dispatching routine in the Executive module DVINT. This common routine avoids having to duplicate code in drivers. This routine—in essence acting like an RH70 controller driver or a sophisticated ICB—determines which driver will receive control upon an interrupt. Operating like the routine that handles interrupts for overlapped seek devices, this routine determines the type of device that interrupted and dispatches to the proper driver.

1.4.7 Subcontroller Devices

Certain devices have two-level controllers, such as magtapes, where a TM03 connects to an RH70 MASSBUS controller and also connects to TE16 magtape drives. In such an arrangement, the TM03 is a subcontroller, or master unit, that controls slave units; a register in the master unit reports the number of the slave unit that generates an interrupt.

A subcontroller is associated with a data structure called a subcontroller request block (KRB1) that serializes access to the subcontroller. Therefore, a driver must request and receive access to both the subcontroller and the controller for a unit before executing any operations. The KRB1 is a subset of the KRB and every unit on the subcontroller points to the KRB1 of the subcontroller to which it is attached.

1.4.8 Full Duplex Input/Output

In certain circumstances it may be necessary for a driver to handle more than one I/O request on a unit at the same time. Typically, a driver processes only one I/O packet per unit at any one time. In normal operation the driver calls the Executive routine $GTPKT to get an I/O packet to process. When $GTPKT returns an I/O packet, it marks the device busy and does not allow additional I/O until the first I/O activity completes. Therefore, only one I/O process can be in progress at the same time on a device. Full duplex operation, however, allows more than one I/O process to be in progress on a device at the same time.

To allow full duplex operation, the $GTPKT routine has a special entry point called $GSPKT. A driver calling $GSPKT specifies an acceptance routine to which $GSPKT returns control when an eligible packet is found. The acceptance routine determines whether to accept or reject the packet. The criteria applied by the acceptance routine could be that a write request is accepted if a write has just completed or that a read request is accepted if a read has just completed. If the routine rejects the packet, it indicates so to $GSPKT, which continues to search for another packet. If the acceptance routine accepts the packet, $GSPKT dequeues the packet and passes it to the driver but does not modify U.BUF and U.CNT in the unit control block (UCB) nor does it mark the device busy. As a result, during full duplex operation the device appears idle even while it is processing an I/O request.

To complete an I/O request under full duplex operation, the driver calls the $IOFIN routine rather than the $IOALT or $IODON routine. $IOFIN does final processing without making the device look idle, as $IOALT and $IODON attempt to do. In full duplex operation, a unit will always appear idle to the system and the driver acceptance routine will determine whether the device can handle an I/O request.

A driver handling full duplex operations requires augmented data base structures. The conventional data base structures are defined for only one I/O request in progress per unit. Because the driver has to keep more information concerning a unit that allows two I/O requests in progress, you may have to alter the UCB and other data base structures to provide additional offsets. The DIGITAL-supplied full duplex terminal driver not only uses a lengthened UCB and a nonstandard SCB but also connects to a dynamically allocated UCB extension when the device is configured on line.

A driver that handles full duplex operations provides a specific example of software that handles concurrent I/O for individual units. Some devices (such as the microprocessor-based LPA11-K laboratory subsystem supplied by DIGITAL) can handle a number of simultaneously active I/O requests. The software to handle such concurrent I/O may require augmented driver data base

structures so that the context of each I/O process remains distinct and controllable. The driver for the LPA11-K relies on an extended user control block (UCB) to preserve the context of a maximum of eight simultaneously active I/O processes. User-written software for such a device must properly synchronize fork processing to prevent substituting the I/O context of one process for that of another. Moreover, the $GSPKT routine also might be used as described above to make a unit appear idle when it is busy.

1.4.9 Asynchronous System Traps

The primary purpose of an asynchronous system trap (AST) is to inform the task that a certain event has occurred—for example, the completion of an I/O operation. As soon as the task has serviced the event, it can return to the interrupted code.

Some directives can specify both an event flag and an AST; with these directives, you can use ASTs as an alternative to event flags or you can use the two together. Therefore, you can specify the same AST routine for several directives, each with a different event flag. Thus, when the Executive passes control to the AST routine, the event flag can determine the action required. However, it is standard programming practice to use the I/O status block (IOSB) rather than the event flags to determine which I/O operation is completed, so that when control is passed to an AST from a QIO$, the I/O status block is on top of the stack. Use this IOSB to determine which I/O has completed.

The Executive queues ASTs in a first-in-first-out queue for each task and monitors all asynchronous service routine operations. Because asynchronous traps may be the end result of I/O-related activity, the task cannot control the occurrence of the ASTs directly. A typical asynchronous trap condition is the completion of an I/O request. The timing of such an operation clearly cannot be predicted by the requesting task. If the task does not specify an AST service routine in an I/O request, a trap does not occur and normal task execution continues.

However, the task may, under certain circumstances, block recognition of ASTs to prevent simultaneous access to a critical data region. When access to the critical data region has been completed, the queued ASTs may again be honored. The Disable AST Recognition (DSAR$A) and Enable AST Recognition (ENAR$S) Executive directives provide the mechanism for doing this.

Associating asynchronous system traps with I/O requests enables the requesting task to be truly event driven. The system executes the AST service routine contained in the initiating task as soon as possible, consistent with the task's priority. Using the AST routine to service I/O-related events provides a response time that is considerably better than a polling mechanism and provides for better overlap processing than the simple QIO$ and WTSE$ macros. Asynchronous system traps also provide an ideal mechanism for use in multiple buffering of I/O operations.

The Executive inserts all ASTs in a first-in-first-out (FIFO) queue on a per task basis as they occur (that is, the event they are to signal has expired). The Executive executes them one at a time whenever the task does not have ASTs disabled and is not already in the process of executing an AST service routine. Executing the AST includes storing certain information on the task's stack, including the task's WTSE$ mask word and address, the directive status word (DSW), the program status (SP), the program counter (PC), and any trap-dependent parameters. The task's general-purpose registers, Registers 0 through 5, are not saved; therefore, AST service routines must save and restore all registers used. If the registers are not restored after an AST has occurred, the task's subsequent execution may be unpredictable.

After an AST is processed, the trap-dependent parameters (if any) must be removed from the task's stack and an AST Service Exit (ASTX$S) macro executed. The ASTX$S macro, described in the *RSX-11M-PLUS and Micro/RSX I/O Drivers Reference Manual*, issues the AST Service Exit directive. On AST service exit, control returns to another queued AST, to the executing task, or to another task waiting to run.

The *RSX-11M-PLUS and Micro/RSX Executive Reference Manual* describes AST service routines and all Executive directives that handle them.

1.4.10 Using Asynchronous Buffered I/O

Typically, data for input and output requests are transferred directly to and from task memory. To allow the successful transfer of data, the task cannot be checkpointed until the transfer is complete. For most high-speed devices, the transfer occurs quickly enough so that a task does not occupy memory for too long a time. For slow-speed devices, however, some mechanism must be available to avoid binding memory to a task for too long a time while the task is performing I/O.

Using the routines $TSTBF, $INIBF, and $QUEBF in the Executive module IOSUB, a driver can execute an I/O request for a slow-speed device and allow the task to be checkpointed while the request is in progress. To perform the I/O request, the driver buffers the data in its memory while the task is checkpointed and while the I/O request is in progress.

To test whether a task is in a proper state to initiate I/O buffering, the driver calls the $TSTBF routine and passes it the address of the I/O packet. By extracting the address of the task control block (TCB) from the I/O packet, $TSTBF can examine various task attributes (for example, if the task is checkpointable, buffered I/O can be performed). $TSTBF then returns to the driver and indicates whether buffered I/O can be performed.

If buffered I/O can be performed, the driver performs two operations. First, it establishes the buffering conditions. For an output request, it copies the task buffers to dynamically allocated pool space. For an input request, it allocates sufficient pool space to receive the incoming data. Second, the driver calls the $INIBF routine to initiate the I/O buffering. $INIBF decrements the task I/O count, increments the task's buffered I/O count in T.TIO, and releases the task for checkpointing and shuffling. If the task is currently blocked, the task state is transformed into a stop state until the task is unblocked or buffered I/O completes, or both. Checkpointing the task is subject to the normal requirements of an active or stop state as described in the *RSX-11M-PLUS and Micro/RSX Executive Reference Manual.*

After the driver transfers the data, it calls the $QUEBF routine to queue the buffered I/O for completion. $QUEBF sets up a kernel asynchronous system trap (kernel AST) for the buffered I/O request and, if necessary, activates the task. When the task is active again, a routine in the Executive module SYSXT notices the outstanding AST and processes it. (If the request is for input, the routine copies the buffered data to task memory.) This mechanism occurs transparently to the task, thus the name kernel AST. The routine then calls the driver to deallocate the buffer from pool. $IOFIN completes the processing.

1.4.11 I/O Queue Optimization

Without I/O queue optimization, the operating system groups input and output requests in the queue by highest priority on a FIFO basis. The first request at the highest priority appears first in the queue and is processed first. Other requests within that priority are then processed sequentially until the last request at that priority is serviced.

With I/O queue optimization, however, the next I/O request at the highest priority is not necessarily the next sequential request to be processed. I/O queue optimization allows the queue to be scanned and each request to be examined. The I/O request, according to the method of optimization then in effect, is the next one dequeued and passed to the I/O driver for processing. The highest priority requests are still serviced first; however, throughput is improved by the reordering of requests within a priority.

There are three methods of I/O queue optimization available:

- Nearest Cylinder
- Elevator
- Cylinder Scan

The Nearest Cylinder method processes the I/O request that is closest to the one at which the disk head is currently positioned. The Elevator method processes requests as the disk head moves from the perimeter to the innermost track of the disk. Once the disk head reaches the innermost track, the direction is reversed and requests are processed along the disk as the head moves back to the perimeter. The Cylinder Scan method operates like the Elevator method, except that requests are only processed as the disk head moves from the perimeter to the innermost track. Once at the innermost track, the disk head returns to the perimeter and begins processing new requests.

The method you choose for your system depends on the I/O processing requirement of your application, the frequency with which tasks access certain data areas on the disk, and the physical location of data on the disk. Refer to the *RSX-11M-PLUS and Micro/RSX System Management Guide* for information on selecting I/O queue optimization methods.

Before an I/O request can be queued to the driver, all three queue optimization methods require the starting cylinder number of the I/O request. To find the cylinder number, the logical block number (LBN) of each I/O request is converted to cylinder, track, and sector form. The routine $DRQRQ in the Executive module DRSUB begins this conversion. Because the cylinder, track, and sector form is specific to the device geometry, this conversion must be completed by a separate routine in the driver. The routine $DRQRQ locates the conversion routine in the driver through offset D.VCHK in the driver dispatch table.

The routine $DRQRQ calls the conversion routine for all I/O requests. However, if the functions are not logical transfer functions, such as ACP functions or Attach and Detach operations, the conversion routine does not complete the conversion, but rather returns to $DRQRQ.

Drivers without queue optimization call the routine $BLKCK in the Executive module MDSUB to check the limits of the I/O request. If $BLKCK locates an error, the routine $IOALT in the Executive module IOSUB is called for the I/O request and the driver is returned to the initiation entry point. If you chose queue optimization, a return to the initiation entry point is not desirable because the necessary functions of $DRQRQ will not be completed. Therefore, to

ensure the correct return to $DRQRQ if an error is detected, your completion routine must call the routine $BLKC2 in the Executive module MDSUB instead of $BLKCK.

The routine $GTPKT in the Executive module IOSUB performs the actual optimization. The driver calls the Executive routine $GTPKT for an I/O request to process. $GTPKT scans the queue of I/O packets to select those of the highest priority. The routine then chooses the correct packet within that priority based on the optimization method currently in effect, dequeues that packet, and returns control to the driver to process that I/O request.

1.5 Distributed I/O

On a multiprocessor system, a task may issue an I/O request to any device on any processor. The Executive must be responsible for distributing the I/O request to the correct processor. To ascertain to which processor a device is attached and to have the driver execute on the correct processor, the Executive must perform some processor-specific functions. The following sections introduce the data structure and the processing routines used by the Executive for processor-specific functions.

1.5.1 UNIBUS Run Mask

To help describe devices attached to a processor, the software relies on a concept called UNIBUS run. A UNIBUS run consists of a group of distinct devices, all electrically connected to the same UNIBUS and not separated by any bus reconfiguration devices. Each UNIBUS run is attached to the same processor at the same time because of the way the devices are physically attached to the UNIBUS. (Devices attached to a MASSBUS of a processor are also on the processor's UNIBUS run.) The UNIBUS run, then, is the smallest fragment of a particular UNIBUS capable of being switched (or not switched) between processors.

Essential to understanding UNIBUS runs is the concept of a switched bus. A switched bus is a portion of a UNIBUS that can be physically connected to one of multiple UNIBUSes. A device on the UNIBUS, called the DT07 UNIBUS switch, controls the connection and allows a switched bus to be connected to any one of a maximum of four UNIBUSes. Any UNIBUS device or devices, except a processor or another bus switch, may be connected to a switched bus. Moreover, because of the electrical delay associated with the bus switch, some high-speed devices (such as the DMC-11) cannot be on a switched bus.

In a multiprocessor system, the DT07 allows the switch bus to be physically switched from the UNIBUS of one processor to the UNIBUS of another processor. When the switch is connected to a particular processor's UNIBUS, all peripherals on the switched bus operate as if they were permanently connected to that UNIBUS. By means of reconfiguration software, a switched bus can be disconnected from one UNIBUS and be available for connection to another processor's UNIBUS. Because a user task can direct an I/O request to any device on the system, the Executive must be able to perform the operation on the specific processor to which the device is connected.

A UNIBUS run is represented in a cell called a UNIBUS run mask (or URM). The URM is a 16-bit word containing a bit for every possible UNIBUS run. UNIBUS runs are numbered from 0 to 15, and the system is restricted to a maximum of 16 UNIBUS runs. There are four UNIBUS runs reserved for the maximum of four processors. The numbering allows a maximum of 12 switched buses. However, a switched UNIBUS cannot be connected to another switched UNIBUS. A primary UNIBUS run would contain a processor, its UNIBUS, and the peripherals

directly attached to its UNIBUS; a secondary run would consist of a switched bus and the devices attached to it.

In the I/O data structures for each controller in the multiprocessor system is an associated UNIBUS run mask. The bit set in the URM defines the UNIBUS run to which the controller is attached. In the Executive, there is a table of connectivity masks—one UNIBUS run mask for each processor in the system. The table represents the UNIBUS runs to which each processor is attached. A bit set in the table mask word for a processor indicates that the UNIBUS run is currently associated with that processor.

To ascertain whether a controller is attached to the current processor, the Executive compares the controller URM with the mask for the processor in the connectivity table. If the same bit is set in both words, the controller is attached to the current processor. If a bus is switched from one processor to another, the system need alter only the connectivity masks of the processors affected.

1.5.2 Conditional Fork

The conditional fork routine ($CFORK) is the method by which the Executive distributes I/O requests to devices connected to another processor. In a multiprocessor system, peripheral devices are generally accessible to only one UNIBUS run. Devices that do have dual-access capability are not necessarily accessible from every UNIBUS. The Executive ensures that, when a driver accesses a controller, the driver process executes under control of the processor in whose I/O space the controller registers reside. An exception is when the Executive passes control to a driver for special processing of an I/O packet. In this case, the driver is responsible for ensuring that the process executes on the correct CPU. See the discussion of the UC.QUE bit in Section 4.4.4.

The conditional fork routine is necessary because the system allows processors to remain anonymous as far as task execution is concerned. The system does not restrict execution of a user task to the processor associated with the device to which the task directs I/O. Basically, it is the driver processes that need to execute on specific processors.

1.5.3 Processor-Specific Functions

When the Executive calls a driver to initiate I/O, the driver may not be executing on the processor associated with the device unit to which I/O is directed. When the driver requests an I/O packet to process, the Executive must ensure that the driver executes on the correct processor because the driver can access the I/O page. Therefore, the Executive routine ($GTPKT) that dequeues an I/O packet for the driver performs a conditional fork. A cell in the fork block for the device unit contains a UNIBUS run mask. This mask defines the processor to which the unit's controller is attached. The conditional fork routine accesses this cell to ascertain what action to take.

The URM of the device to which the I/O request is directed therefore determines whether the driver can execute on the current processor. If the URM of the device intersects the current processor URM, the conditional fork routine returns and the I/O packet is immediately passed to the driver. The driver then normally proceeds to start the proper I/O function. If execution must be continued on another processor, the conditional fork routine performs a fork (that is, calls the $FORK routine). The driver has no indication that it has become a fork process (that is, the action is transparent to the driver).

To ensure that the driver executes on the correct processor, the fork routine performs two operations. First, it creates and queues a fork block for the processor on which the driver must execute. Second, it returns to the driver in such a manner as to force the driver to dismiss itself. As soon as possible, the fork processor restarts the driver process executing on the appropriate processor.

For devices that do not have an assigned controller, the system may defer determining whether the driver executes on the current processor. Therefore, for overlapped seek and dual-access devices, the conditional fork routine is entered after the Executive routine that assigns the controller.

1.5.4 Vectoring for Privileged Tasks and the Executive

RSX-11M-PLUS Executives allow vectored access to their code by a privileged task. Accessing entry points in a vectored Executive requires that the privileged task be slightly altered to reference these entry points through local vectors in the task. When the privileged task is built, the addresses in these vectors are resolved to an offset in the directive common. At run time, the task resolves the offset to a virtual address by using the directive common.

A vectored access to Executive code provides the following advantages:

- For some systems, space can be saved on distribution kits by distributing common privileged tasks with multiple and different system images.
- In most cases, privileged tasks need not be rebuilt when the Executive is updated.
- In most cases, it allows software sold separately to span multiple versions of RSX systems without relinking the tasks.

A vectored Executive requires a special directive common in the system. This common is used by the Task Builder when it builds a task and resolves the Executive routine addresses. Each vectored Executive routine has a unique fixed offset in the directive common.

Privileged tasks built against the dummy symbol table file RSXVEC.STB have all Executive references resolved as offsets into the special directive common, rather than as virtual Executive addresses. These offsets are translated into virtual addresses at task run time.

For a system that does not support vectoring, a modified task can be rebuilt by using a system-specific RSX11M.STB symbol table file. This will resolve all symbols at task build time. Both RSXVEC.STB and RSX11M.STB are distributed on all RSX-11M-PLUS systems.

Any privileged task that directly references Executive code must be modified if you want to take advantage of vectoring. An area at the beginning of such a task should be set aside to store vectors of the Executive entry points that allow the task to address the required Executive locations. Therefore, privileged tasks that take advantage of vectoring will increase in size by approximately one word for each unique Executive reference. The task's vectors in this area contain offsets into the special directive common previously described.

When a task is invoked, a special GIN$ directive is issued to translate the offsets in the task to virtual addresses found in the directive common. The addresses obtained in the common overwrite the offsets located in the task in memory. The task uses these addresses to reference Executive routines.

The GIN$ directive is located within the vector directive common; therefore, a task that uses the GIN$ directive will fail on a system that has not been vectored. To run a vectored task on a system that has not been vectored, you must remove the GIN$ directive and rebuild the task, linking it to a specific RSX11M.STB file.

Additional changes are necessary to make it possible for drivers to be vectored. Drivers, which are mapped through APR 5, must use a different scheme because the GIN$ directive also uses APR 5. The driver must be modified to address the entry points through a known fixed low-core memory location (112_8). This location contains a pointer to SYSCM where the information needed by the driver is located.

1.5.5 Converting to Vectored Access in Privileged Tasks

For a task to use vectored entry, a table of entry addresses must be created within the task. A convenient way to do this is to omit the dollar sign from global labels, where possible, and to omit the period from offset definitions. For example, suppose a privileged task contains a reference to the Executive global data cell $ACTHD, calls routine $MPLND, and uses offset definition S.PKT, which is one of the floating data structure offsets.

The first step in converting a privileged task is to construct a table within the task similar to the following example:

```
;          EXECUTIVE ENTRY POINT VECTOR TABLE

EXEVEC:
           .WORD  0            ; FLAG FOR VECTOR NOT YET FILLED
ACTHD:     .WORD  $ACTHD       ;
MPLND:     .WORD  $MPLND       ;
SPKT:      .WORD  S.PKT        ;

EXEVCL=<<<.-EXEVEC>/2>-1>
```

The first word of the table is made zero to indicate that the table has not been initialized yet. Each subsequent word of the table is merely a .WORD reference to the desired address. When the privileged task is linked, it must be linked with RSXVEC.STB, which provides internal offset values for the symbols, instead of RSX11M.STB.

The second step in converting a privileged task requires that you change all the references to Executive symbols in the task to access those symbols through the vector. Some examples of this process follow:

```
;          CONVERTING EXECUTIVE REFERENCES IN A PRIVILEGED TASK

MOV        $ACTHD,R5           ;; GET ADDRESS OF FIRST TCB
                    becomes
MOV        @ACTHD,R5           ;; GET ADDRESS OF FIRST TCB

CALL       $MPLND              ;; FOLLOW REDIRECTION LIST
                    becomes
CALL       @MPLND              ;; FOLLOW REDIRECTION LIST

MOV        S.PKT(R3),R0        ;; GET ADDRESS OF FIRST PACKET
                    becomes
ADD        SPKT,R3             ;; POINT TO I/O QUEUE LISTHEAD
MOV        (R3),R0             ;; GET ADDRESS OF FIRST PACKET
```

For the last step, a GIN$ directive must be included to perform the translation of the vector. An example section of code is as follows:

```
            .MCALL  GIN$
TRNVEC:     GIN$    GI.VEC,EXEVEC,EXEVCL  ; Translate Executive vector
START:      DIR$    TRNVEC                ; Translate Executive vector
                      .
                      .
                      .
            .END    START
```

Before executing the GIN$ directive, the vector is filled with internal offset definitions and the first word is 0. The GIN$ directive replaces each of the offset definitions with the real address of the symbol in the Executive. The GIN$ directive also fills in the first word of the vector with a 1. If the first word is nonzero when the GIN$ function is issued, the function is made a no-op. (A no-op function is one that is considered successful as soon as it is issued.) The foregoing procedure allows the GIN$ directive to be executed any number of times without corrupting the vectors.

1.5.6 Converting to Vectored Access in Drivers

Drivers cannot issue a GIN$ directive; therefore, a routine that is callable at system state is provided to fill in the vector. Because this routine is at a variable address, some fixed address entry is needed. Location 112 points to a table of information. This table contains one datum, which is the address of the APR bias of the common containing the vectoring routine. The entry point to the routine is offset 4 in that common. The following source statements are an example of a calling sequence:

```
;           CALLABLE ROUTINE FOR CONVERTING EXECUTIVE REFERENCES IN A DRIVER

    MOV         @ 112,R0            ; GET ADDRESS OF TABLE OF ENTRIES
    MOV         (R0),R0             ; GET ADDRESS OF APR BIAS -- FIRST WORD IN TABLE
    MOV         KINAR6,-(SP)
    MOV         (R0),KINAR6         ; MAP COMMON THROUGH I-SPACE APR 6
    MOV         #EXEVEC,R3          ; POINT TO VECTOR
    MOV         #EXEVCL,R2          ; SPECIFY LENGTH OF VECTOR
    CALL        @#140004            ; TRANSLATE THE VECTOR
```

1.6 Overview of Incorporating a User-Written Driver into RSX-11M-PLUS

How you incorporate a user-written driver into an RSX-11M-PLUS system depends mainly on whether you make your driver loadable or resident. If your driver is loadable, its data base can be either loadable or resident. If your driver is resident, both its data base and its code are resident. Thus, because you build the Executive image during system generation, you can include any resident driver elements in the Executive image only during system generation. If your driver is loadable and has a loadable data base, you can incorporate it at any time after you build the Executive under which the driver will run.

Micro/RSX does not perform the system generation process, so you must incorporate a new driver on the currently running system. For this reason, you can only incorporate a loadable driver with a loadable database into a Micro/RSX system. You cannot incorporate a resident driver or a loadable driver with a resident data base.

During system generation, you answer questions concerning the types and quantity of peripheral devices on your system. Based on your answers, the system generation software creates the device data base source files. The file SYSTB.MAC contains the data base definitions for all the DIGITAL-supplied devices that were generated with resident data bases. The files xxTAB.MAC, where xx is the device mnemonic, contain the data base definitions for each of the DIGITAL-supplied devices that were generated with loadable data bases. The files xxDRV.MAC, where xx is the device mnemonic, contain the driver code to support the devices. The system generation software assembles and task-builds these modules. The resident driver and data base modules are linked into, and become a permanent part of, the Executive. The loadable driver and data base modules are task-built separately for loading into memory after the Executive has been built.

A privileged system task called LOAD is responsible for loading into memory a driver that is not resident. LOAD creates the necessary interrupt control blocks (ICBs) for accessing a driver and establishes the linkage between the data base structures in the system device tables and the driver code being loaded. Another system task, CONFIGURE (CON), initializes the interrupt vectors to point to the ICBs and actually places the devices on line. CON can also change the vector and CSR address assignments in a device's data base. Another privileged system task called UNLOAD can remove a loadable driver from memory. (Although UNLOAD removes a loadable driver, it does not remove a loadable data base.)

To incorporate a user-written driver into RSX-11M-PLUS, you first create two modules—one in which you define the data base and the other in which you include the driver code itself. You then must integrate your driver data base and the driver code modules into the system device tables. If your data base is resident, your data base module must satisfy the following linkages:

- The link of the controller table (CTB) list
- The link of the device control block (DCB) list

The linkage for the driver code connects the DCB for the device that your driver supports to the driver dispatch table (DDT). If your driver and data base are loadable, you must supply symbols and labels in your code that LOAD needs. Your device interrupt vectors are initialized and the devices are placed on-line by CON.

When you load a driver, LOAD checks to see whether a data base is resident for the type of device whose driver is being loaded. If a data base is not resident, LOAD reads the driver symbol definition file to find the start and end of the data base in the driver image. (Thus, if your driver data base is to be loadable, you must have defined its start and end in the data base source code.) Knowing the start and end, LOAD reads the data base from the driver image. LOAD places the data base in the system pool so that it resides in Executive address space, relocates pointers and links within the data base to be valid Executive addresses, and connects the CTB and DCBs in the data base to the system device tables. Moreover, so that the system device tables are not corrupted by an incorrect data base, LOAD performs many consistency and validity checks on the data base being loaded.

If your driver is loadable and has a loadable data base, you will build the following structures:

- A loadable image containing the driver code module followed by the driver data base module.

  A symbol definition file that LOAD depends on to find critical data base and driver locations.

You will link the driver image to the Executive under which the driver will run. However, the driver image will be separate from the Executive image. LOAD is responsible for loading both your driver data base and driver code, for connecting the data base to the system device tables, and for connecting your driver code to the data base.

If your driver is loadable but has a resident data base, you will need to perform a system generation and build the Executive under which the driver will run to link your driver data base modules into the system device tables. This operation makes your driver data base resident with the system device tables. You must also build the following structures:

- A loadable image containing the driver code.
- A symbol definition file that LOAD will use to locate the driver dispatch table.

LOAD is responsible for loading your driver code and for connecting your driver code to the data base resident with the system device tables.

If your driver is resident, you will need to perform a system generation and build the Executive to link the driver data base into the system device tables and to include the driver code in the Executive image.

Whatever type your driver is, you will use the CON task to initialize the device interrupt vectors and place the device on-line.

Because LOAD provides consistency and validity checks on a data base that is being loaded, DIGITAL recommends that you make your driver and its data base loadable. (Additional rationale for making your driver fully loadable is given in Section 1.7.) Furthermore, with a loadable driver and loadable data base, you can more easily modify your driver and its data base. You need not rebuild your Executive and privileged tasks. To change the driver code, you need only build a new driver image, unload the current version, and reload the new version. To change the driver data base, you must build a new driver image (which incorporates the modified data base module), rebootstrap your system, and load the new driver (which causes the modified data base to be loaded). (You must bootstrap your system to change the data base because UNLOAD does not unload a data base, and because LOAD does not load a data base for a driver if one is currently loaded for that driver.)

Using a loadable driver with a loadable data base saves work in the long term. During debugging, data base inconsistencies are likely to be caught by LOAD. Thus, you prevent many such errors from later creating system problems.

A resident driver or a loadable driver with a resident data base is more difficult to debug and to modify. LOAD does not perform consistency and validity checks on a resident data base. Thus, a valuable debugging aid is not available. Moreover, to modify such drivers, you must rebuild the Executive, which generally implies rebuilding the privileged tasks.

1.7 SPR Support

The capability to incorporate a user-written driver into your system is a supported feature of RSX-11M-PLUS. Because a user-written driver is considered a system modification, DIGITAL may not support the system that results after it is incorporated. Since it is a part of the Executive, your driver can subtly corrupt it. Therefore, DIGITAL cannot guarantee support that entails debugging user-written drivers.

Fixing a problem in a system is largely a matter of being able to reproduce the problem reliably. If a problem on your system can be shown to have no relation to your driver and DIGITAL can reproduce the problem, SPR support can be provided. A good reason for using a loadable driver with a loadable data base is that you can more easily attain an unmodified system by not loading your driver and its data base. You can then reproduce a suspected problem in an unmodified system and can submit an SPR that DIGITAL can answer. You can save both yourself and DIGITAL time in answering the SPR if you attempt to re-create a suspected problem on your system without your driver and its data base.

Chapter 2
Device Driver I/O Structures

This chapter deals mainly with structures at the block level, their relationship to the hardware configuration and functionality supported, and their relationships to each other. The precise description of each structure is given in Chapter 4.

2.1 I/O Structures

The main elements in the driver I/O environment define the logical and physical characteristics of the supported hardware and establish the links and connections by which routines can access and manipulate driver data. The following subsections describe the control blocks that a driver data base module defines and explain in general terms the purposes for each block.

2.1.1 Controller Table (CTB)

A controller table defines a unique controller type on the system. A CTB must exist for each physical controller type. All controller tables are linked together in a list, with the head of the list $CTLST in the Executive common area. The list of the controller tables is one of the threads running through the system data base to provide access to all device-related data. The link in the last CTB in the list has a value of zero.

Associated with each CTB is a two-character ASCII controller name that must be unique throughout the system. This unique name allows the Executive to find the correct CTB for the controller type. For example, the RH11/70 controller has the name RH instead of DB, DS, DR, or MM.

A CTB is a static structure created during system generation. Any user-written driver data base, therefore, must have its own CTB. The user-created controller table must also be linked into the system CTB list.

A CTB has generic status information, links, and pointers to other structures on the system. The Executive handles interrupts for the controller type and dispatches to the correct driver routine using the table of KRB addresses in the CTB.

2.1.2 Controller Request Block (KRB)

The controller request block is the means by which the Executive maintains controller- or hardware-specific information and accesses the correct information for a unit owned by its associated controller. One KRB exists for each device controller in the configuration. It stores data such as vector and CSR location, status, and UNIBUS run mask.

In a configuration where a device has only one access path to a controller and the controller allows only one operation at a time, the KRB is combined with another structure called the status control block (SCB), which holds context for a unit while an operation is in progress. Because only one access path is possible in such a configuration, unit context is always associated with the same controller. Moreover, because only one operation is possible at a time, the same context storage area can be used for all units attached to the controller. Thus, in a conventional driver operating environment, the context storage is merely an extension of the controller request block.

In a configuration where multiple operations in parallel on the same controller are possible, the controller context is separate from each independent unit context. Therefore, each unit capable of operating independently on a controller has the context of the current I/O operation stored in an SCB separate from the controller KRB. In such an operating environment, any unit can access the controller while other operations are pending, but only one unit can have access at a time. The KRB, then, indicates which unit owns the controller for the current operation and synchronizes access among driver processes on the same controller.

Where multiple parallel operations are allowed on a controller, there must be some way to delay access to the controller when it is busy. Therefore, in the KRB the Executive holds the head of a list of access requests called the controller request queue. The list contains fork blocks for driver processes awaiting controller access. The queue is the means by which the Executive serializes access to the controller.

When a controller allows parallel operations, the software must have a means of determining which of several units generated an interrupt. The KRB, therefore, contains a table of addresses that associate the controller with all the units connected to it. This table, indexed by physical unit number, must appear if the controller in question supports overlapped seek operations. When a device has multiple-access paths, the controller-specific information in the KRB is separate from each independent unit context. In a situation where a device accesses alternate controllers, a driver must request the Executive to assign the unit to a specific controller. The unit assignment involves temporarily associating unit context with the KRB of the specific controller. The SCB, then, holds information connecting it to the KRB of the currently assigned controller.

The KRB also holds the configuration status of the controller. If the KRB indicates that the controller is offline, no activity can take place on any unit connected to the controller.

2.1.3 Device Control Block (DCB)

The device control block describes the static characteristics of a device type and of units associated with a certain device type. The DCB is the means of access to the driver dispatch table and thus to the driver. At least one DCB exists for each logical type of device on a system. There may be more than one DCB for a device type. For example, there are two device control blocks for the device TT: on a system that supports terminals connected by both DL11 and DZ11 interfaces.

A cell in each device control block forms a link in a forward-linked list, with the head of the list starting in a cell ($DEVHD) in the Executive common area. This list, as with the CTB list, is a main thread running through the system data structures to device-related data. The link in the last DCB in the list has a value of zero.

The static data in the DCB gives such information as the generic device name, unit quantity and links to individual unit data, the address of the driver dispatch table, and the types of I/O functions supported by the driver. Typically, the Executive QIO directive processing code, not the driver code, accesses the DCB.

2.1.4 Unit Control Block (UCB)

The unit control block holds much of the static information about an individual device unit and contains a few dynamic parameters. Although unit control blocks need not be any prescribed length for different devices, all unit control blocks for the same device type must be of equal length. (The UCB length is stored in the device control block.) This condition allows the UCB to contain varying amounts of unit- and device-independent data for different types of devices.

A UCB, one of which exists for each device unit, enables a driver to access most of the other structures in the I/O environment. A UCB provides access to most of the dynamic data associated with I/O operations. Given the address of a UCB, a driver may readily find most of the other data structures in which it is interested because the proper links exist. Because of this access information, the UCB is a key control block in the driver I/O structure.

The static data in the UCB includes pointers to other I/O structures, definitions of unit control bits that regulate directive processing, definitions of unit status bits that describe operational conditions, and definitions of unit- and device-dependent characteristics and storage cells.

Data in the UCB is accessed and modified by both the Executive and the driver.

2.1.5 Status Control Block (SCB)

The status control block holds driver context for operations on a device unit. The SCB stores data such as the pointer to the head of the queue of input/output requests; the link to the fork blocks queued for the unit; the fork process context; timeout, unit status, and error logging information; and the address for the controller request block (KRB) representing the device controller (if the device has a controller).

The Executive accesses the SCB to set up an I/O request, to store context while a request is in progress, and to post results and status. When the driver accesses the SCB, it is usually for read access only.

The number of status control blocks depends on the processing support in the Executive. If the controller itself cannot handle parallel operations, only one SCB is needed for each controller. In such a case, a controller can have only one unit processing a command at one time, and there is no need to store context for more than one unit at a time. There is also no need for a physically separate controller request block (KRB) to separate generic data from unit context. Therefore, the driver data base contains the required KRB cells in the status control block.

If the controller allows parallel operations and the Executive supports this feature, there must be one SCB to store context for each unit capable of operating independently on the controller. In such a configuration, a cell in each SCB points to the KRB of the controller to which the units are connected.

2.2 Driver Dispatch Table (DDT)

The driver dispatch table[1] contains the entry points to, and the interrupt entry addresses for, the driver. An entry point is the location at which the Executive calls the driver to perform a specific function. An interrupt entry address is a location to which the central processor or the Executive transfers control within the driver for servicing hardware interrupts. The pointer to the interrupt entry address resides either in an interrupt control block if the driver is loadable or in the device interrupt vector in the system common area of the Executive if the driver is resident.

Every driver has four conventional entry points as follows:

- I/O initiation
- Cancel I/O
- Device timeout
- Device powerfail

The following two entry points are added for controller and unit online and offline status changes:

- KRB status change
- UCB status change

For many devices, these status-change entry points are merely a return to the Executive calling routine.

The following two entry points have been added for advanced driver features:

- Deallocate buffers and next command (FDX TTDRV)
- Address checking and conversion (queue optimization disk drivers)

[1] The DDT is not a structure in the strict sense of the word because it is defined in the instruction part of the driver code. However, because it contains addresses for dispatching code, it is included in the data structure description.

2.2.1 I/O Initiation

The Executive transfers control to this entry point to inform the driver that work for it is waiting to be done. To make work for the driver, the Executive performs predriver-initiation processing. (Predriver initiation is described in Chapter 3). If, at the end of predriver processing, the Executive has I/O packets queued for the driver, it calls the driver at this entry point.

When the driver gets control at its I/O initiation entry point, Register 5 contains the address of the UCB for the unit on which the request is to be processed. To establish access to the I/O packet, the driver calls an Executive routine that either returns information in registers concerning both the packet to be processed and the associated data in order to gain access to the data structures[1] or causes the driver to dismiss itself. (There may be no packet to process or the driver may already be busy.)

Once control is returned to a driver and there is a request to process, the driver must extract the information from the registers, establish data within the control blocks, and process the request.

Typically, a driver is called at this entry point when there is a packet in the I/O queue. However, a driver can be called before a packet is placed in the I/O queue. Refer to the description of the U.CTL control flag UC.QUE in Section 4.4.4 for information on queuing an I/O packet to the driver.

2.2.2 Cancel I/O

To terminate an in-progress I/O operation, the system flushes the I/O queue and calls the driver at this entry. There are many situations where a task must terminate I/O. When such a termination becomes necessary, a task issues an Executive request and the Executive relays the request to the driver by calling it at this entry point.

The driver is responsible for checking that the I/O operation in-progress was issued from the task that is forcing the termination and for completing or terminating the operation before returning to the caller.

Typically, a driver is called at this entry point only when an I/O operation is in progress. A driver can be called even if the unit specified is not busy. Refer to the description of the U.CTL control flag UC.KIL in Section 4.4.4 for information on unconditional cancelling of I/O.

2.2.3 Device Timeout

When a driver initiates an I/O operation, it can establish a timeout count. If the operation fails to complete within the specified interval, the Executive notes the lapse and calls the driver at this entry point. Using this facility, a driver can wait for an interrupt but need not hang up if the interrupt never occurs. Thus, no driver should ever stall on a request because a hardware failure prevented an expected interrupt from happening.

[1] The $GTPKT routine, which gets a packet for the driver to process, is described in Chapter 7.

2.2.4 Device Power Failure

The Executive calls the power failure entry point when power is restored after a failure any time the controller is busy (that is, when I/O is in progress). Typically, a driver responds to a power failure in the same manner it responds to a timeout. In such cases, the power failure entry point may simply be a return to the caller because recovery will occur by means of the timeout entry point. The driver is called for both controller and unit power failure unless the driver is associated with a common interrupt controller. For common interrupt controllers, the driver is called at this entry point only for unit power failure and is called at a special entry defined in the common interrupt table for controller power failure.

A driver can be called when power is restored regardless of the existence of an outstanding I/O operation. Refer to the description of the U.CTL control flag UC.PWF in Section 4.4.4 for information on unconditional call on power failure.

2.2.5 Controller and Unit Status Change

Two entry points are required for configuration status changes of the controller and units. The Executive enters one entry point to put the controller on line and take it off line. The other entry point, called once for each unit whose status changes, is for putting units on line and taking them off line. The driver must show successful completion of the online or offline request or the Executive will not effect the status change. The driver has 60 seconds to perform whatever synchronization it requires before returning to the Executive. In most cases, however, the driver will return immediately.

2.2.6 Device Interrupt Addresses

Control passes to an interrupt address when a device, previously initiated by the driver, completes an I/O operation and causes an interrupt in the central processor. A device may have associated with it more than one interrupt entry. For example, a full duplex device such as a terminal will have two interrupt addresses. The interrupt entry differs from an entry point in that the connection between the device and the driver is more direct—the Executive is not involved.

The interrupt addresses are arranged in a block in the DDT. The arrangement is general enough to support multicontroller drivers such as the terminal driver. The block defines the addresses to include in the vector for the driver. There is no restriction on the number of vectors each controller has, and the number of vectors is implied by the number of addresses in the interrupt address block.

2.3 Typical Control Relationships

This section presents different arrangements of the control structures found in RSX-11M-PLUS. The section concentrates on the relationships among device control, unit control, status control, and controller request blocks and controller tables based on hardware and functions supported. Descriptions of the detailed contents of the structures is left to Chapter 4, where the coding requirements are presented. Some of the arrangements are not conventional but are shown to convey the flexibility you can find in a system. Section 2.4 shows how such arrangements fit into the overall system I/O data structure.

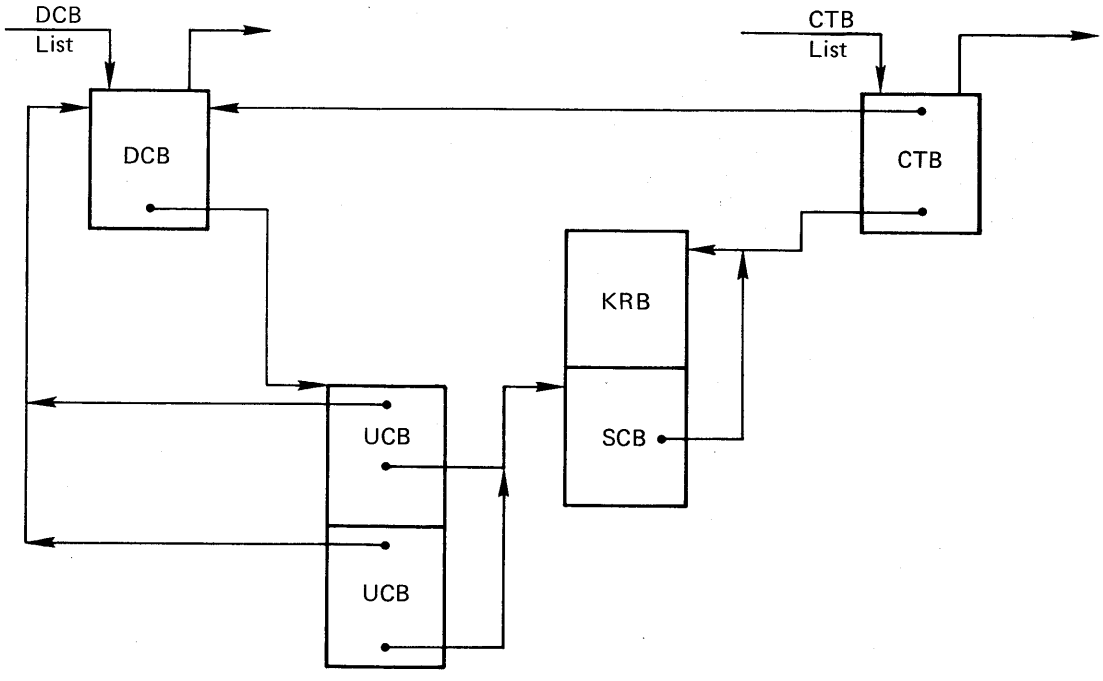
The arrangements described in this section illustrate the strategy of offering a flexible I/O data structure. There need be only one controller table for each controller type. Multiple-device control blocks for a single device type reflect the capability to handle varying characteristics. The existence of one or more status control blocks depends on the degree of parallelism possible: one SCB for each controller servicing several units (no parallelism); or one for each device unit combination on the same controller (unit operation in parallel).

The I/O data structure reflects the hardware configuration that the data structures describe. The flexibility in the data structure arrangements provide flexibility in configuring I/O devices. The information density in the structures themselves reduces the coding requirements for the associated drivers.

2.3.1 Multiple Units per Controller, Serial Unit Operation

A typical arrangement of structures for a user-written driver is shown in Figure 2–1. The arrangement could represent an RK05J controller with two RK05 drives attached. A single controller table (CTB) defines the existence of the controller type on the system. One device control block (DCB) establishes the characteristics for the type of device running on the controller.

Figure 2–1: Multiple Units per Controller, Serial Unit Operation



The status control block (SCB) and controller request block (KRB) are contiguous in this arrangement because the software does not allow another I/O operation to begin while the controller is busy. A separate unit control block (UCB) describes each unit attached to the controller. The UCBs are associated with the SCB, which contains the context of the operation currently in progress.

2.3.2 Two Controllers, Serial Operation

Another typical conventional arrangement of structures for a user-written driver is shown in Figure 2–2, which could represent two LP11 controllers: one with an LP04 and the other with an LP05 attached. It represents the simplest case of driver processing. Figure 2–2 shows what is required for a controller that allows only a single I/O operation for each controller. A single controller table defines the existence of the controller type on the system. One device control block establishes the characteristics for the type of device running on the controller.

The status control and controller request blocks are contiguous in this arrangement because, while the controller is busy, another I/O operation cannot begin. Only one SCB is necessary to store the context of the unit operation. The UCB points to the SCB, which in turn points to the KRB of the unit's controller. Because the system must handle interrupts from multiple controllers, the controller table points to the KRB of each controller present.

2.3.3 Parallel Unit Operation

Some devices, such as the RK06, allow multiple units to have seek operations in progress at the same time. In particular, the RK06 allows such operations to overlap a data operation. Figure 2–3 shows the arrangement needed in the software structures to support parallel operations on one controller.

Two additional structural changes are required from the serial operation arrangement. First, because more than one unit may have an operation pending at the same time, a structure is needed to store unit context. Therefore, for each unit (and each unit control block) there is a separate status control block. Second, because interrupts can come from more than one unit, there must be a way to access the proper unit. As a result, the controller request block contains a table of unit control block addresses that allows the driver to find the structures for the unit generating an interrupt.

Figure 2–2: Two Controllers, Serial Operation

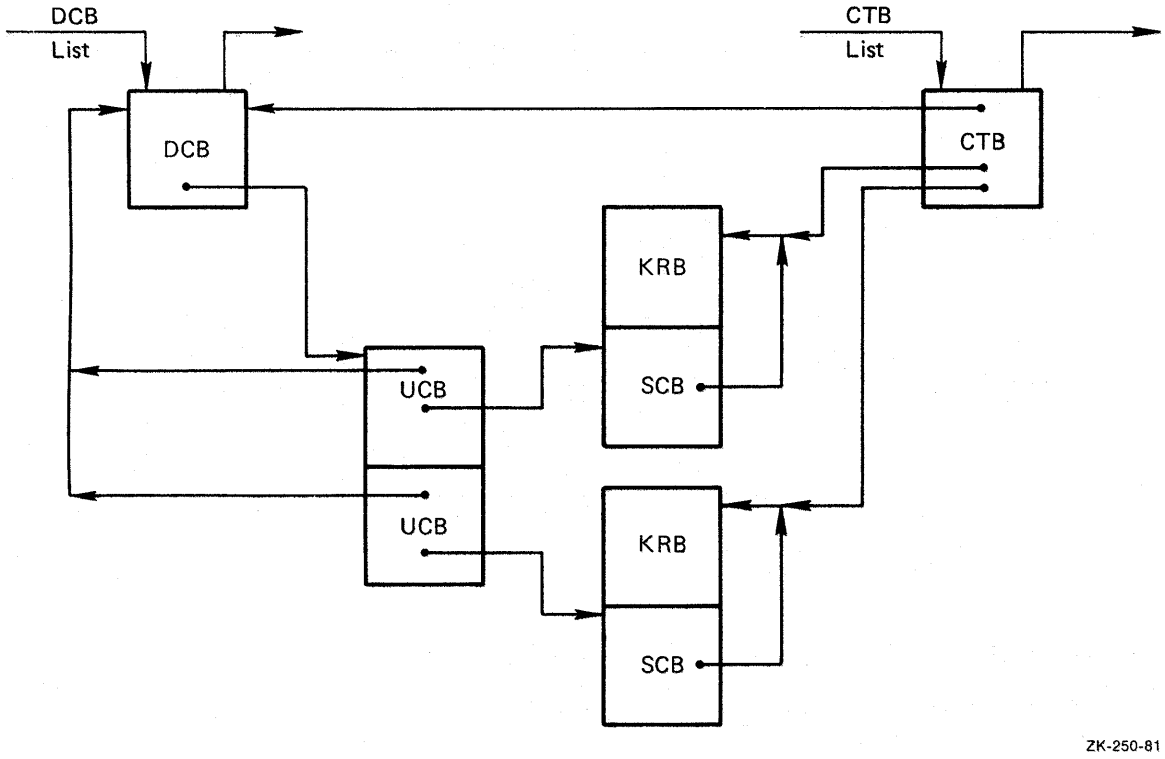
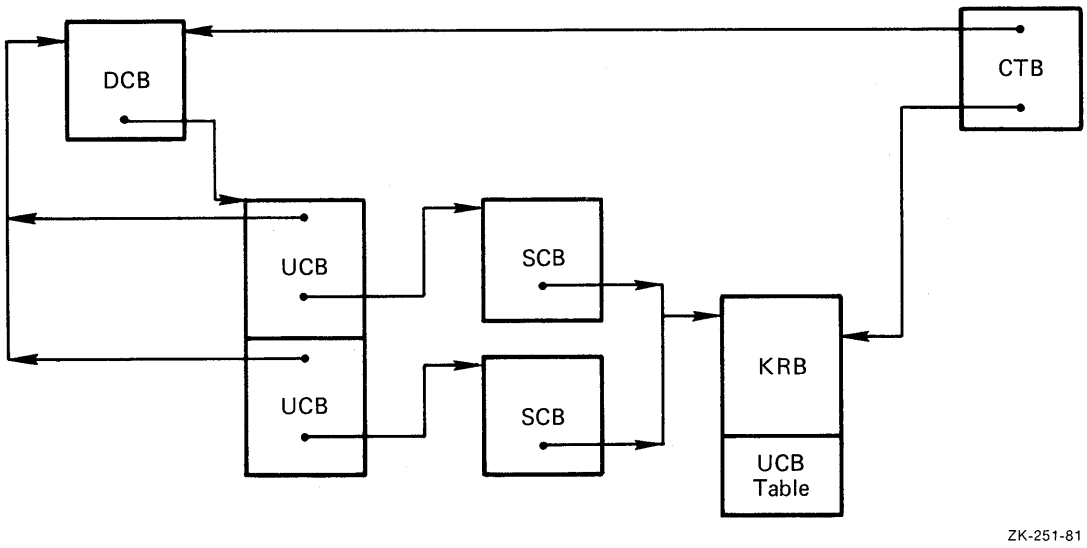


Figure 2–3: Parallel Unit Operation (Overlapped Seek)



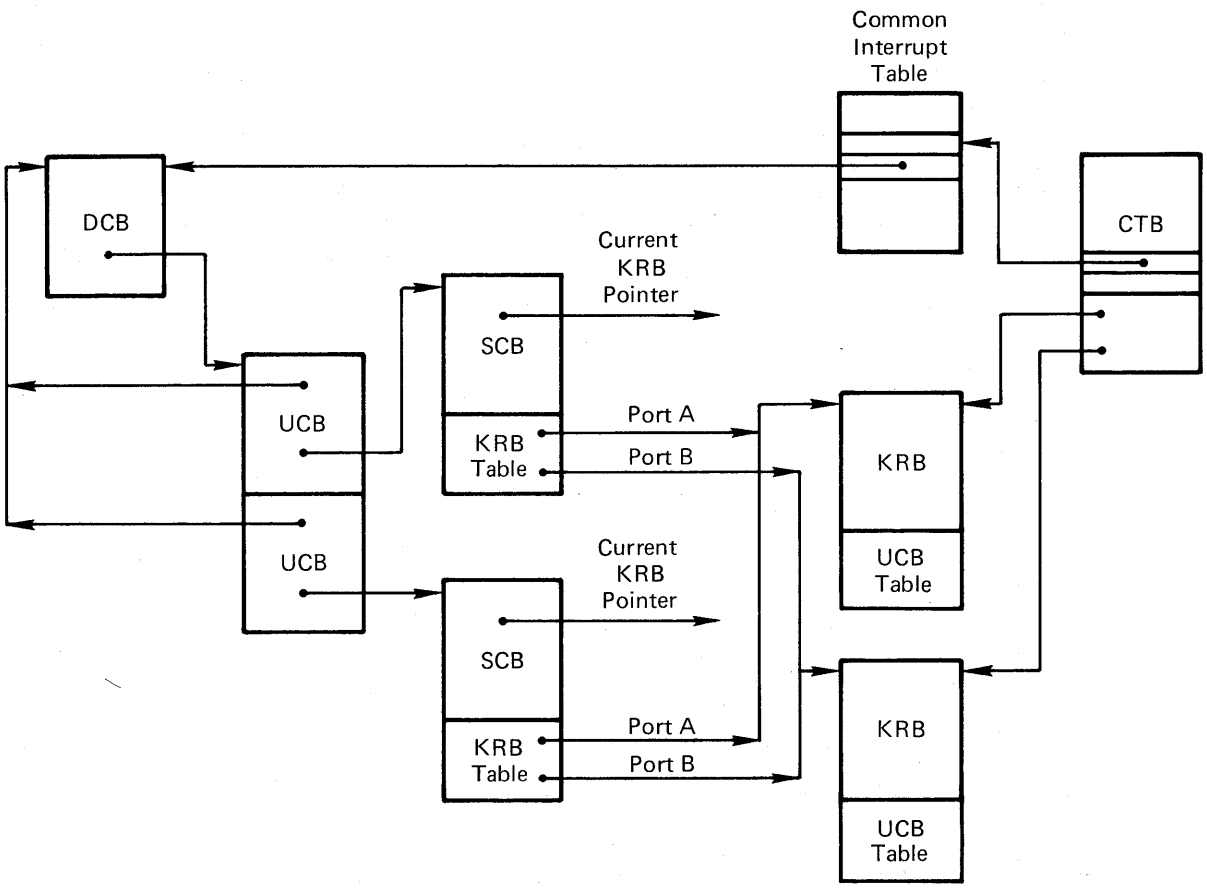
2.3.4 Multiple-Access (Dual-Access) Operation

Some devices, such as the RK06, have a dual-port option that provides multiple-access paths to units. On the RK06, dual ports on the unit enable a single unit to be electronically switched between two controllers. Figure 2–4 shows the changes in the structures needed to support dual-access operations.

Separate status control blocks are needed for each unit because, if one controller is currently busy, the alternate controller can be idle and allow the operation to proceed. The difference in the dual-access structure is that the SCB no longer points to the same controller request block all the time as in the overlapped seek arrangement. The Executive can change the SCB pointer to a KRB to reflect the capability to electronically switch a unit between two controllers.

To enable the software to determine which controllers may access a unit, the SCB has a table of KRB addresses. For dual-access disks, the table has two entries containing the addresses of the controller request blocks for each controller between which the unit can be switched.

Figure 2–4: Dual-Access Operation



2.4 Overview of Data Structure Relationships

This section presents an overview of the relationships among the user-written driver data structures previously introduced in this chapter, the Executive I/O structures, and DIGITAL-supplied driver structures. The goal of the section is to convey the general manner in which user-written structures and code link into the system I/O scheme and to describe generally the way the system uses the structures. The specific user-written structures are simplified somewhat so that the emphasis is placed on the linkages with other parts of the system rather than on the details of user-written structural relationships.

This section should be used with Section 2.3 to understand the general structural concepts. For example, Section 2.3 describes various arrangements of unit control, status control, and controller request blocks based on hardware functions the software structures support, while this section treats such arrangements as an engineering black box that is oriented in the general I/O environment. Thus, in the generalized I/O data structure depicted in this section, the pointers in the KRB table of the SCB are not shown and the table is simply marked KRB Table.

Figure 2–5, which provides the basis for the presentation of the I/O data structure, shows the individual elements and the important link fields within them. To simplify the discussion and to guide you through the data structures, the numbers in the figure correspond to the numbers in the lead paragraphs of the following text:

❶ The location represented by the Executive symbol $DEVHD is a cell in system common (SYSCM). It is the head (or start) of a singly linked, unidirectional list of all device control blocks in the system. The first word in each DCB is a link to the next DCB.

The list of device control blocks is one of the two threads running through the system data tables for device-related information. For example, the list is the means by which executive routines scan the data structures to determine what devices are on the system and what is the status of units. User-written device control blocks must be linked into the list of system-defined DCBs.

❷ Every loadable driver is associated with a partition control block (PCB). The PCB defines the characteristics of the memory area into which the driver is loaded. The Executive and tasks such as LOA and UNL reference the data in the PCB. A driver is not concerned with the PCB.

❸ If a task is attached to a unit, the UCB has a pointer to the task control block (TCB) of that task.

❹ The task header is an independent entity in the I/O data structure and the driver never accesses it. A copy of the task header is taken from the task partition and stored in the Executive dynamic storage region whenever the task is actually in memory. This copy is then used by the Executive.

A logical unit table (LUT) entry in the task header has two items of interest: a pointer to an associated unit control block and, if a file is being accessed, a pointer to a window block. The Executive accesses the logical unit table of a task during a QIO request and indexes the table by the logical unit number specified in the QIO request.

❺ A device control block has a pointer to the unit control block of the first related unit. Because the length of a UCB is stored in the DCB and all UCBs are allocated in a continuous area, access to all the UCBs related to that DCB is possible. This arrangement allows software to access all related unit information for a device type.

A DCB also has a pointer to the start of the driver dispatch table. This pointer allows the Executive to call the driver at its entry points to process an I/O-related or a reconfiguration request.

❻ Each unit control block contains a pointer back to its related DCB. This backpointer allows the Executive interrupt dispatch code to enter the proper driver (through the pointer to the driver dispatch table).

A status control block (SCB) is associated with each UCB . The SCB is shared by all units for a device type that does not require units to operate in parallel. When units can operate in parallel, each UCB has its own associated SCB.

❼ As part of processing a QIO directive (queued I/O request), the Executive builds a structure called an I/O packet. Storage for packets is in the system dynamic storage region (the pool). The Executive connects the packets by a pointer in each packet to form a singly linked list called the I/O queue. The Executive maintains two pointers in the SCB to the

list of packets. The first pointer is to the start of the list and the second pointer is to the last packet in the list.

The driver should not access the list of I/O packets directly. When the Executive transfers control to the driver to initiate processing of an I/O request, the driver immediately calls an Executive service routine to get a packet to process. The routine passes data to the driver to process the request (for example, the address of the packet). Thus, the Executive—and not the driver—removes a packet from the queue of packets. However in performing the I/O request, the driver can access certain fields in the packet to be processed because a pointer to the currently active I/O packet is kept in the SCB. [1]

The Executive determines the ordering of packets in the queue. Typically, higher-priority requests are placed at the head of the queue.

❽ At least one status control block exists for each controller. Where a controller and software support operations in parallel on multiple units, one SCB exists for each unit capable of operating independently. A pointer in the SCB connects to the controller request block (KRB) of the controller to which the related unit is connected. If multiple-access paths between a unit and controller are possible, the KRB pointer is dynamic; therefore, the KRB to which the SCB points at one instant is considered to be the currently assigned KRB. To reflect the existence of alternate controllers, a table of pointers to all the possible KRBs is contained in the SCB, separate from the pointer to the currently assigned KRB.

The fork block in the SCB contains some of the driver process context. The driver executes an Executive routine so that processing will occur at fork level. To preserve processing status, the routine stores some context in the fork block. When the driver eventually runs again, the fork processor recovers the proper context from the fork block.

On multiprocessor systems, the fork block contains an extra cell to define the processor on which the driver must execute the I/O request. The Executive routine that preserves context in the fork block ensures that the driver code is processed on a particular processor.

The fork blocks for pending driver processes are connected in a singly-linked list, the head of which is in a location ($FRKHD) in the Executive region. Generally, the fork processing routines link a fork block in FIFO order. At location $FRKHD +2 the Executive maintains a pointer to the last fork block in the list.

❾ A file control block (FCB) is associated with each open file on a mounted volume. The file system alone uses the FCB to control access to the file.

❿ For each open file on a mounted volume, a window block exists for each task that has the file open. The window block holds pointers to areas on the volume on which the file resides and speeds up the process of retrieving data items from the file. (The associated ACP need not be called to convert a virtual block number in a file to a logical block number on the device.) The driver is not concerned with the window block.

⓫ The driver dispatch table (DDT) is part of the driver code and, through the vector and the interrupt control block, is the means by which the device interrupts are passed to the driver.

⓬ The controller request blocks (KRB) are linked into the I/O data structure through the pointers in the controller table (CTB). The table of KRB addresses in the CTB is static.

[1] Normally, the driver does not directly manipulate the I/O queue. An exception is when a driver needs to examine an I/O packet before it is queued or instead of having it queued. This exception involves a status bit in a control byte of the unit control block. For more information on queuing of I/O packets to the driver, refer to the description of the UC.QUE bit in Section 4.4.4.

The KRB table gives the Executive access to the structures for a controller when it initiates an interrupt. To report the termination of a data transfer command, a controller initiates an interrupt. (While such a controller-initiated interrupt is in progress, the hardware delays interrupts from units.) The Executive determines the correct KRB by indexing the CTB with the controller number from the PSW in the vector.

For a controller that allows unit operation in parallel (overlapped seek support), the related KRB must have a table of UCB addresses. This table allows the driver to access the structures of the unit that generates an interrupt. When a unit interrupts, its controller records (in the attention summary register) the physical number of the interrupting unit. The driver must retrieve the number and use it to index the UCB table in the KRB to access the proper unit control block.

To support parallel unit operations, the KRB also contains a queue to regulate controller access. This queue—the controller request queue—is a list of fork blocks for driver processes that have requested and have been denied access to the controller. The driver requests access to a controller. If the controller is busy, the Executive forces the driver to wait for access by placing the fork block in the queue of processes waiting for access. The Executive gives precedence to control access over requests for data transfer by placing positioning requests onto the front of the queue and adding data transfer requests to the end of the queue. When a unit is given access, the controller status is set to busy and the unit UCB address is set to connect the KRB to the owned UCB.

To indicate what unit to process on a controller-initiated interrupt, a cell in the KRB points to the unit control block (UCB) of the unit that currently owns the KRB.

The KRB queue cells have two words. The first word points to the fork block in the SCB of the next unit to get access. The second word points to the fork block in the SCB of the last unit to get access. If the first word is 0, then the second word points to the first and no unit is waiting for access to the controller.

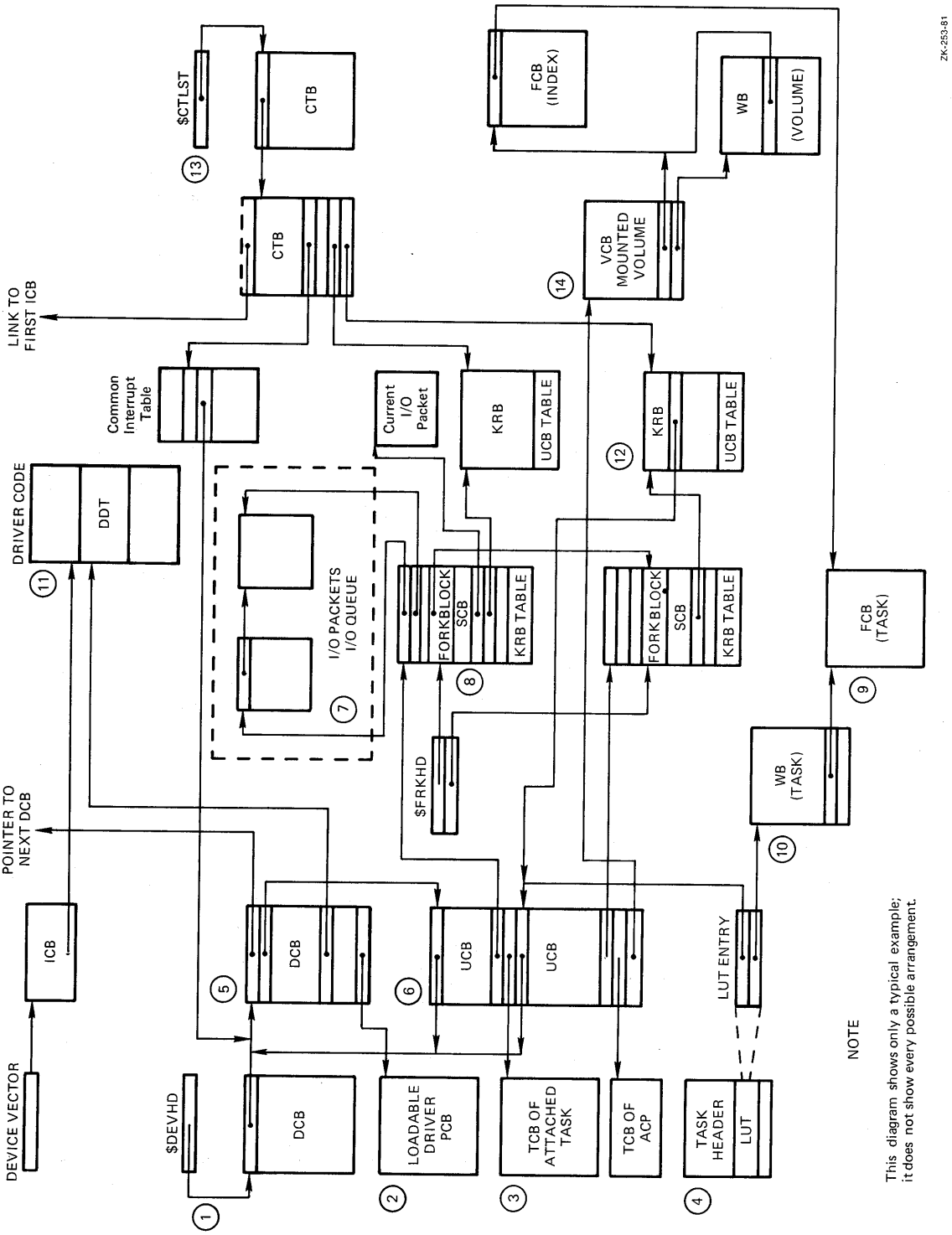
⓭ The location represented by the Executive symbol $CTLST is a cell in system common (SYSCM). It is the head (or start) of a singly linked, unidirectional list of all controller tables (CTBs) in the system. A word in each CTB is a link to the next CTB. The last CTB in the list contains a link word of 0.

The list of controller tables is one of the two threads running through the system for device-related information. (The list of device control blocks is the other thread.) A user-written controller table must be linked into the list of system-defined CTBs. This list is the mechanism by which system routines, such as those for reconfiguration, access I/O data structures for hardware information.

⓮ One volume control block (VCB) exists for each mounted volume in the system. The VCB maintains volume-dependent control information.

Pointers within the VCB connect to the file control block (FCB) and window block (WB). The FCB and WB control access to the volume's index file, which is a file of file headers. All FCBs for a volume form a linked list starting from the index file FCB. These linkages aid in keeping file access time to a minimum. A conventional driver does not access any of these structures.

Figure 2–5: Composite I/O Data Structures



Chapter 3
Executive Services and Driver Processing

This chapter describes the flow of I/O requests through the system.

The Executive provides services related to I/O drivers. Some services are provided before a driver process is initiated and are therefore called predriver initiation services. The predriver initiation services are those performed by the Executive during its processing of a QIO directive; these services are not available as Executive calls.

Predriver initiation processing extracts from the QIO directive all I/O support functions not directly related to the actual issuance of a function request to a device. If the outcome of predriver initiation processing does not result in the queuing of an I/O packet to a driver, the driver is unaware that a QIO directive was issued. Many QIO directives do not result in the initiation of an I/O operation.

Other services are available to the driver after it has been given control, either by the Executive or as the result of an interrupt. They are available as needed by means of Executive calls.

An important concept used in this section and in Chapter 4 is the state of a process. In RSX-11M-PLUS, a process can run in one of two states: user or system. Since drivers operate entirely in the system state, the programming standards described in Chapter 4 apply only to system-state processes.

3.1 Flow of an I/O Request

Following an I/O request through the system at the functional level requires that the following limiting assumptions be made about the state of the system when a task issues a QIO directive:

- The system is running and ready to accept an I/O request. All required data structures for supporting devices attached to the system are intact.
- The only I/O request in the system is the sample request under discussion.
- The example progresses without encountering any errors that would prematurely terminate its data transfer; thus, no error paths are discussed.
- The controller in question executes only a single operation at a time.

3.1.1 Predriver Initiation Processing

The Executive performs the following processing before it passes control to a driver's I/O request:

1. Task issues QIO directive.

   The user program first creates, either statically (by QIOW$C, QIOW$, QIO$C, or QIO$) or dynamically (by QIOW$S or QIO$S) a directive parameter block (DPB) containing information about what I/O is to be performed on what device. It then issues the directive.

   All Executive directives are called by means of EMT 377. The EMT causes the processor to push the processor status word (PSW) and program counter (PC) onto the stack and to pass control to the Executive's directive processor.

2. Executive issues QIO dispatch.

   The Executive directive dispatcher DRDSP ascertains that the EMT is a QIO directive and calls the QIO directive processor DRQIO.

3. DRQIO performs first-level validity checks.

   The QIO directive processor validates the logical unit number (LUN) and the unit control block (UCB) pointer. DRQIO checks whether the LUN supplied in the directive parameter block is a legal value. If it is not a legal value, the directive is rejected. If the LUN is legal, DRQIO checks whether a valid UCB pointer exists in the logical unit table (LUT) for the specified LUN. This check ascertains whether the LUN is assigned. If the check fails, the directive is rejected. If both these checks are successful, DRQIO then performs the redirect algorithm.

4. DRQIO performs redirect algorithm.

   Because the UCB may have been dynamically redirected by a Redirect command, QIO directive processing traces the redirect linkage until the target UCB is found. The target UCB provides the links to most of the other structures of the device to which the I/O operation will be directed.

5. DRQIO performs additional validity checks.

   The event flag number (EFN) is validated as well as the address of the I/O status block (IOSB). If either is illegal, the directive is rejected. Immediately following successful validation, DRQIO resets the event flag and clears the I/O status block.

6. DRQIO obtains storage for and creates an I/O packet.

   The QIO directive processor now acquires a 20-word block of dynamic storage for use as an I/O packet. It inserts into the packet the device-independent data items that are used subsequently by both the Executive and the driver in fulfilling the I/O request. Most items originate in the requesting task's directive parameter block (DPB).

   At this point DRQIO sets the directive status to +1, which indicates directive acceptance. Note that a directive rejection is a return to the caller with the C bit set. In addition, a directive rejection is transparent to the driver.

7. DRQIO validates the requested function.

If the function is legal, DRQIO checks to see whether the unit is on line. If the unit is off line, the packet is rejected. The function is one of four possible types:

Control
No-op
ACP
Transfer

With the exception of Attach/Detach, control functions are queued to the driver. If the bit UC.ATT is set, Attach/Detach will also be queued to the driver. If the requested function does not require a call to the driver, the Executive takes the appropriate action and calls the I/O Finish routine ($IOFIN).

No-op functions do not result in data transfers. The Executive performs them without calling the driver. No-ops return a status of IS.SUC in the I/O status block.

ACP functions may require processing by the file system. More typically, the request is a read or write virtual function that is transformed into a read or write logical function without requiring file-system intervention. When transformed into a read or write logical function, the function becomes a transfer function (by definition).

Transfer functions are address checked and queued to the proper driver. This means that DRQIO checks the address of the I/O buffer, the byte count, and the alignment requirement for the specified device. If any of these checks fails, DRQIO calls the I/O Finish routine ($IOFIN), which returns an I/O error status and clears the I/O request from the system. If the checks succeed, DRQIO either places the I/O packet in the driver request queue according to the priority of the requesting task or, if the UC.QUE bit is set, gives the packet directly to the driver. (See Section 4.4.4 for a description of the UC.QUE bit.)

3.1.2 Driver Processing

Once the Executive has successfully completed predriver initiation, the driver performs the I/O operation, as follows:

1. Driver requests work.

   To obtain work, the driver calls Get Packet ($GTPKT). $GTPKT either provides work, if it exists, or informs the driver that no work is available or that the SCB is busy. If no work exists, the driver returns to its caller; if work is available, $GTPKT sets the device controller and unit to busy, dequeues an I/O request packet, and returns to the driver.

2. Driver issues I/O.

   From the available data structures, the driver initiates the required I/O operation and returns to its caller. A subsequent interrupt may inform the driver that the initiated function is complete, assuming the device is interrupt driven.

3. Driver processes interrupt.

   When a previously issued I/O operation interrupts, the interrupt causes the driver to be entered. The driver processes the interrupt according to the programming protocol described in Chapter 1. According to the protocol, the driver may process the interrupt at priority 7, at the priority of the interrupting device, or at fork level. If the processing of the I/O request associated with the interrupt is still incomplete, the driver initiates further I/O on

the device (step 9). When the processing of an I/O request is complete, the driver calls $IODON.

4. $IODON finishes I/O processing.

   $IODON removes the busy status from the device unit and controller, queues an AST if required, and determines whether a checkpoint request pending for the issuing task can now be effected. The IOSB and event flag, if specified, are updated, and $IODON returns to the driver. The driver branches to its initiator entry point and looks for more work (step 8). This procedure is followed until the driver finds the queue empty, whereupon the driver returns to its caller and the driver process vanishes.

   Eventually, the processor is granted to another ready-to-run task that issues a QIO directive, starting the I/O flow anew.

3.2 Executive Services Available to a Driver

Once a driver is given control following an I/O interrupt or by the Executive itself, a number of Executive services are available to it. These services are discussed in detail in Chapter 7.

However, four Executive services merit special emphasis because virtually every driver in the system uses them:

- Get Packet ($GTPKT)
- Interrupt Save ($INTSV)
- Create Fork Process ($FORK)
- I/O Done ($IODON, $IOALT, or $IODSA)

3.2.1 Get Packet ($GTPKT)

The Executive, after it queues an I/O packet, calls the appropriate driver at its I/O initiation entry point. The driver then immediately calls the Executive routine $GTPKT to obtain work.[1] If work is available, $GTPKT delivers to the driver the highest-priority, executable I/O packet in the driver's I/O queue and sets the SCB status to busy. If the driver's I/O queue is empty or if the driver is busy, $GTPKT returns a no-work indication.

If the SCB related to the device is already busy, $GTPKT so informs the driver, and the driver immediately returns control to the Executive.

Note that, from the driver's point of view, no distinction exists between no-work and SCB busy, because an I/O operation cannot be initiated in either case.

[1] An exception is a driver that handles special user buffers. Such a driver must call certain other Executive routines before calling $GTPKT. See Section 4.4.4 for a description of the UC.QUE bit.

3.2.2 Interrupt Save ($INTSV)

A driver should not directly call the $INTSV coroutine but should use the INTSV$ macro call. Therefore, if the driver is loadable, it need not call $INTSV and the macro will not generate the call in the driver. (The interrupt save processing is done by either the interrupt control block or the appropriate common interrupt routine in the Executive.) If a driver is resident, the INTSV$ macro call generates the call to the $INTSV coroutine. The coroutine saves code in the driver because the call is shorter than the code to save and restore the conventional Registers 4 and 5. More importantly, the $INTSV coroutine gets the driver onto the system stack if it is not already there. The INTSV$ macro is described in more detail in Section 4.3 and the interrupt entry point is described in Section 4.5.

3.2.3 Create Fork Process ($FORK)

Synchronization of access to shared data bases is accomplished by creating a fork process. When the driver needs to access a shared data base, it must do so as a fork process; the driver becomes a fork process by calling $FORK. The SCB contains preallocated storage for a four- or five-word fork block. See Section 4.4.5 for a description of the fork block. Section 7.4 contains details on $FORK. After $FORK is called, a routine is fully interruptable (priority 0) and its access to shared system data bases is strictly linear.

3.2.4 I/O Done ($IODON or $IOALT)

At the completion of an I/O request, the subroutines $IODON or $IOALT perform a number of centralized checks and additional functions:

- Store status if an IOSB address was specified
- Set an event flag if one was requested
- Determine whether a checkpoint request can now be honored
- Determine whether an AST should be queued

$IODON and $IOALT also declare a significant event, reset the SCB and device unit status to idle, and release the dynamic storage used by the completed I/O operation.

Chapter 4
Programming Specifics for Writing an I/O Driver

Chapters 2 and 3 give overviews of data structures and Executive services, respectively. This chapter summarizes programming standards, presents overviews of programming requirements for user-written driver code and data, and gives details of the data structures and driver code. Executive services are covered in Chapter 7.

4.1 Programming Standards

I/O drivers function as integral components of the RSX-11M-PLUS Executive, and this manual enables you to incorporate I/O drivers into your system. User-written drivers must follow the same conventions and protocol as the Executive itself if they are to avoid complete disruption of system service. Failure to observe the internal conventions and protocol described fully in Chapter 1 can result in poor service and reductions in system efficiency.

The programming conventions used by RSX-11M-PLUS system components are identical to those described in Appendix E of the *PDP-11 MACRO-11 Language Reference Manual*. DIGITAL urges you to adhere to these conventions.

4.1.1 Programming Protocol Summary

Drivers are required to adhere to the following internal conventions when processing device interrupts:

1. No registers are available for use unless $INTSV has been called or unless the driver explicitly performs save and restore operations. If $INTSV is called, Registers 4 and 5 are available; any other registers must be saved and restored. If the driver is to call $INTSV directly, it must do so immediately because $INTSV attempts to retrieve the controller number from the PSW.
2. Noninterruptable processing must not exceed 20 instructions, and processing at the priority of the interrupting source must not exceed 500us.
3. Only a fork process should modify system data bases.

4.1.2 Accessing Driver Data Structures

All the driver data structure elements have symbolic offsets. Because the physical offset values may vary from one version of the Executive to another, your user-written driver code should always use the symbols to access the elements.

Accordingly, your driver code should not step from one structural element to another (relying on the juxtaposition of data structures and individual words in a data structure) but should access each element by symbolic offset. By following this coding practice, you can reduce debugging time when converting an RSX-11M driver to run on RSX-11M-PLUS. Many of the offsets in the RSX-11M SCB differ physically from those in the RSX-11M-PLUS SCB but have the same symbolic values.

On the other hand, it is a common coding practice to assume that zero offsets (particularly link pointers such as D.LNK) will remain zero. This assumption allows the saving of one word per instruction by substituting an instruction such as MOV (R3),R3 for MOV D.LNK(R3),R3. DIGITAL recognizes that such practices are followed and consequently attempts to keep such offsets zero.

4.2 Overview of Programming User-Written Driver Data Bases

You should create the source code for your user-written driver data base in a file separate from that of the driver code. You assemble this file to create the driver data base module. If you make your data base resident, your data base module will be linked separately from the driver code and will be linked to the system device tables module SYSTB.OBJ. (The source code for the SYSTB module is created in UFD [11,10] during system generation.) If your data base is in a separate module and is to be loadable, it will be linked to the end of the driver code module. If your driver data base is in the same module as that of your driver code, it must be at the end of the driver code.

The detailed descriptions of the driver data structures are in Section 4.4. A few fields in the structures are conditional on certain features in the Executive. For this reason, you must use conditional assembly directives and some system-wide symbols defined in the Executive assembly prefix file RSXMC.MAC, which is created during system generation.

To create the source code, you need to know, in addition to the detailed structures, what ordering and labeling are required. These requirements, though not extensive, are important in linking and loading your driver data base. The general coding requirements for both loadable and resident driver data bases are described in the following subsections.

4.2.1 General Labeling and Ordering of Data Structures

If you are creating a loadable data base, you must specify two global labels for the LOAD routines as follows:

$xxDAT:: Marks the start of the user-written driver data base

$xxEND:: Marks the end of the user-written driver data base (immediately following the final word of the data base)

The characters xx represent the two-character mnemonic of the device that your driver data base supports. If either or both of these labels are not defined, LOAD cannot determine the length of your data base when you attempt to load your driver.

There is no mandatory ordering of the different structures in a driver data base. DIGITAL suggests, however, that you place the DCB first, followed by the UCB, the SCBs, the KRBs, and the CTB. If you do not follow this ordering scheme, you must specify the starting location of the first (or only) DCB as described in Section 4.2.2.

4.2.2 Device Control Block Labeling

If the data base for a driver is to be loadable, the LOAD routines require either that the first (or only) DCB be identified by the global label $xxDCB:: or that the DCB be at the start of the data base.

If the data base for a driver is to be resident, you must define the start of the first (or only) DCB with the global label $USRTB::. This label is required to link the last DCB defined in the SYSTB module with the DCB in your driver data base. If you fail to supply this symbol, the Task Builder will generate an undefined reference error when it builds the Executive.

4.2.3 Unit Control Block Ordering

All the UCBs associated with a specific device control block (DCB) must be contiguous and must be of equal length. These requirements are necessary because the DCB has only one link to the UCBs, and that link is to the first UCB. Two data elements, the UCB length and the number of units, are stored in the DCB; together with the link to the first UCB, they are used to locate subsequent UCBs. If you do not follow these requirements, no software can access the UCBs.

4.2.4 Status Control and Controller Request Blocks

All user-written drivers that do not need separate storage for independent unit context should use the continuous allocation of the KRB and SCB. (For an explanation of when independent unit context is required, refer to the discussion of overlapped seek I/O in Section 1.4.1.) Therefore, the KRB and SCB are contiguous and some fields of each structure overlap. This arrangement saves space that would otherwise be required for one SCB for each independent unit. Because only one unit can be active at any one time, all units attached to the same controller can share the SCB. This arrangement of the KRB and the SCB is described in Section 4.4.7.

4.2.5 Controller Table

You must define the start of the table of KRB addresses in the CTB with the global label $xxCTB::. Both the INTSV$ macro call and the Executive LOAD routines require this label.

If your data base is resident, you must use the CTB macro at the CTB link word L.LNK. The CTB macro automatically generates a global label that provides the linkage between the last CTB defined in the SYSTB module and the CTB defined in your driver tables module. (The definition of the CTB macro is created in the file RSXMC.MAC during system generation.

4.3 Overview of Programming User-Written Driver Code

To create the source code to drive a device, observe the following guidelines:

- Thoroughly read and understand this manual.
- Familiarize yourself, in detail, with the physical device and its operational characteristics.
- Determine the level of support required for the device.
- Determine actions to be taken at the driver entry points.
- Create the driver source code.

You can write driver code for RSX-11M-PLUS that does one of the following:

- Supports standard functions and runs on RSX-11M-PLUS only.
- Supports standard functions and is written so that it is compatible with use on both RSX-11M and RSX-11M-PLUS. (This driver needs separate data bases for each system.)
- Supports advanced features and runs on RSX-11M-PLUS only. (Although Chapter 1 discusses advanced features, this manual does not describe how to program advanced features. Your best guide to utilizing advanced features is to use a DIGITAL-supplied driver as a model.)

To assist you in generating proper code for your user-written driver and to provide a stable user-level interface from one release of the system to another, RSX-11M-PLUS provides the macro calls listed in Table 4–1.

Table 4–1: System Macro Calls for Driver Code

Macro Name	General Functions
DDT$	Used conventionally at the start of the driver code as follows: 1. To allocate storage for and to generate a driver dispatch table containing the addresses of entry points in the order in which the Executive expects them 2. To generate special global labels required by the Executive 3. To pass the following information to the Executive LOAD routines: • Which controllers the driver supports • How many interrupt vectors each controller supports • The association between the interrupt vectors and the driver interrupt entry points • To generate default controller and unit status change entry point procedures (for on-line and off-line transitions)
GTPKT$	Used at the I/O initiator entry point to generate the call to the $GTPKT routine and to generate code to save the address of the currently active unit's UCB

Table 4–1 (Cont.): System Macro Calls for Driver Code

Macro Name	General Functions
INTSV$	Used at an interrupt entry point to conditionally generate a call to the $INTSV routine and to generate code to load the UCB address of the interrupting device into R5

The definitions of the system macro calls for drivers are in the Executive assembly prefix file RSXMC.MAC. The following subsections describe the format of the macro calls and other features of user-written driver code. Driver code details (such as labeling requirements and entry point conditions) are presented in Section 4.5.

4.3.1 Generate Driver Dispatch Table Macro Call—DDT$

The DDT$ macro call facilitates generation of the driver dispatch table. The format of the DDT$ macro call is as follows:

Format

```
DDT$ dev,nctrlr,iny,inx,ucbsv,NEW,OPT,BUF
```

Parameters

dev

The two-character device mnemonic.

nctrlr

The number of controllers that the driver services (counting from 1).

iny

Allows the definition of no interrupt entry point or multiple interrupt entry points. If you leave the argument null, the macro generates as the interrupt entry point address the location defined by the conventional label $xxINT.

If you specify NONE, no interrupt entry point is generated for the controller.

If you specify an argument list of the form <aaa,bbb,...> , the macro generates multiple cells containing addresses defined by unconventional labels of the form $xxaaa and $xxbbb. This latter mechanism allows you to define multiple interrupt entry points in the driver. For example, the argument list <INP,OUT> generates two interrupt address labels of the form $xxINP and $xxOUT, the typical names used by drivers with two interrupt entry points.

inx

Uses an alternate I/O initiation entry point address label instead of the conventional xxINI form. If you specify inx, the macro uses as the only I/O initiation entry point address the location defined by the label xxinx.

ucbsv

For compatibility with RSX-11M drivers. If you are writing a driver for RSX-11M-PLUS, you should leave this argument blank. As a result, the macro does not allocate the space for the table of UCB addresses. For guidelines on specifying this argument, refer to Section 4.3.4.

NEW

Distinguishes between RSX-11M-PLUS and RSX-11M drivers. If you specify this argument (any character except null), the macro generates two cells to hold the controller and unit status change entry point addresses. The referenced driver entry points must be labelled xxKRB: and xxUCB:. If your driver uses these entry points, it cannot be compatible with RSX-11M unless the two routines are conditionalized.

If the argument is null, the macro generates code to use the xxPWF entry point for controller and unit on-line and off-line status changes. RSX-11M drivers implicitly handle controller and unit on-line and off-line status changes as power failures. Although this default operation (enabled by code generated from leaving this argument null) is not optimal for operation on RSX-11M-PLUS, the driver will probably function properly without being changed to include the xxKRB and xxUCB entry points.

OPT

Indicates that the driver supports seek optimization. The referenced entry point must be labelled xxCHK:. The routine corresponding to that label should qualify the I/O request and convert it to cylinder track and sector.

BUF

Required if the driver performs buffered input and output. The entry point xxDEA: is generated.

Description

The DDT$ macro constructs the DDT, using as addresses those locations indicated by the standard labels. The macro has arguments that allow you to tailor some of the standard entry points. The format of the DDT generated by the DDT$ macro is described in Section 4.5.1.

4.3.2 Get Packet Macro Call—GTPKT$

The GTPKT$ macro call standardizes use of the Executive $GTPKT routine, which retrieves an I/O packet for the driver to process.

Format

```
GTPKT$ dev,nctrlr,addr,ucbsv,suc
```

Parameters

dev

The two-character device mnemonic.

nctrlr

The number of controllers that the driver services (counting from 1).

addr

The local label defining the location at which to continue execution if there is no I/O packet available. A driver typically executes a RETURN instruction when the $GTPKT routine indicates that there is no I/O packet to process. If you leave this argument null, therefore, the macro generates a RETURN instruction.

ucbsv

For compatibility with RSX-11M drivers. If you are writing a driver for RSX-11M-PLUS, you should leave this argument null. The macro then generates code to load the pointer S.OWN with the address of the UCB returned by $GTPKT. For guidelines on using the argument, refer to Section 4.3.4.

suc

Indicates single unit controller. If you are writing a driver for RSX-11M-PLUS that supports a controller type such as the LP11, to which only a single unit can be attached, you should specify this argument (any characters except null). If you specify this argument, you should ensure that the offset K.OWN/S.OWN in the KRBs of your driver data base points to the UCBs of the units to which the controllers is attached. Thus, the macro does not generate code that stores the UCB address in the KRB (a gratuitous operation) for a device that has only one UCB per KRB.

If your RSX-11M-PLUS driver has multiple units attached to the same controller, you should leave this argument null. The macro therefore generates code to store in the KRB the UCB address of the unit to process.

For multiple SCB controllers that support parallel unit operations but do not require synchronization, K.OWN must be set dynamically by the driver code. For this case, you must specify the suc argument in the GTPKT$ macro.

Description

This macro call generates the call to the Executive $GTPKT routine. You should place it at the I/O initiation (xxINI) entry point because the $GTPKT routine is the way a driver commonly receives work from the Executive. When the driver receives control at its xxINI entry point, the Executive has loaded R5 with the address of the UCB of the unit that the driver must service. Because of the code the macro call generates, the driver immediately calls $GTPKT, which can set the C bit to indicate that no work is pending. The call additionally generates the BCS instruction that returns control to the calling routine when there is no work. If you specify an address as an argument in the macro call, it is used as the destination of the BCS instruction. The address is typically that of a RETURN instruction but does not have to be. Eventually the driver must execute a RETURN to the system.

The $GTPKT routine indicates that the driver has an I/O packet to process by clearing the C bit. Therefore, when the test of the BCS instruction is false, execution continues inline and the driver can process the I/O packet that the Executive queued to it. The $GTPKT routine leaves information in the driver registers to enable the driver to process the request. Refer to the description of the $GTPKT routine in Chapter 7.

4.3.3 Interrupt Save Macro Call–INTSV$

You should specify the INTSV$ macro call at each interrupt entry point in the driver. The macro conditionally generates a call to the Executive $INTSV routine based on whether the driver is loadable. The format of the INTSV$ macro call is as follows:

Format

```
INTSV$ dev,pri,nctrlr,pswsv,ucbsv
```

Parameters

dev

The two-character device mnemonic.

pri

The processor priority (PR4, PR5, or PR6) at which the device runs and at which the $INTSV coroutine will run.

nctrlr

The number of controllers that the driver services (counting from 1).

pswsv

For compatibility with RSX-11M drivers. If you are writing an RSX-11M-PLUS driver, leave this argument null. If your driver is an RSX-11M driver, this argument has no effect.

ucbsv

For compatibility with RSX-11M drivers. If you are writing a driver for RSX-11M-PLUS, you should leave this argument null. The macro generates code that uses the controller index returned in R4 by $INTSV, calculates the KRB of the interrupting controller, and loads the UCB address of the interrupting unit into R5. For guidelines on specifying this argument, refer to Section 4.3.4.

Description

If the symbol LD$xx (where xx is the device mnemonic) is not defined, the macro generates the call to $INTSV and defines the priority at which the interrupt service routine will run. Not defining LD$xx indicates that the driver is resident. (For loadable drivers, the interrupt service routine in the Executive dispatches the interrupt.) For both loadable and resident drivers, however, the macro generates the code to load R5 upon an interrupt.

4.3.4 Use of UCBSV Argument in Macro Calls

The DDT$, GTPKT$, and INTSV$ macro calls allow you to specify the ucbsv argument to maintain compatibility with RSX-11M drivers. RSX-11M-PLUS does not need to utilize the ucbsv argument. The ucbsv argument in the DDT$ macro allocates nctrlr words of storage (one word for each controller that the driver supports) and labels the first word ucbsv. This storage is the CNTBL area used by RSX-11M drivers to contain the address of the unit control block of the interrupting devices for each controller. Both the GRPKT$ and INTSV$ macro calls may use this same area.

If you specify the argument ucbsv in the GTPKT$ macro call, it must be the same label that you supplied for the ucbsv argument in the DDT$ and INTSV$ macro calls. The macro generates code to move the UCB address returned by $GTPKT to the correct location in the table starting at the label ucbsv.

If you specify the argument ucbsv in the INTSV$ macro call, it should be the same label you supplied for the ucbsv argument in the DDT$ and GTPKT$ macro calls. The macro uses ucbsv to locate the UCB address of the interrupting unit and then generates code to load the address into Register 5.

4.3.5 Specifying a Loadable Driver

To specify that a driver is loadable and to enable generation of conditional code, you must define the symbol LD$xx. The definition can appear in either the driver source code or the assembly prefix file RSXMC.MAC. It is usually more convenient to define the symbol in the driver source code because you probably will not have cause to edit RSXMC.MAC. When the symbol is defined, the INTSV$ macro does not generate the call to $INTSV.

4.3.6 Loadable Driver Entry Points for LOAD and UNLOAD

A loadable driver that requires additional initialization and completion functions can define two entry points by labels of the form $xxLOA and $xxUNL (where xx is the two-character device mnemonic). Because these two labels do not appear in the DDT itself, their format is fixed; you must use the exact format in your driver code. When you load the driver, the LOAD routines check for the $xxLOA entry point.

Note

The LOAD routines can perform this function only from MCR. If you attempt to load a driver that has the $xxLOA entry point from VMR, the load operation is terminated with the error message DRIVER REQUIRES RUNNING SYSTEM FOR LOAD/UNLOAD.

The driver is entered, once per UCB, at the $xxLOA entry point at priority zero. At this stage, the driver data base has been loaded and pointers have been relocated. The driver is mapped through APR 5, and the following registers are set up:

R3 = Controller index (undefined if S.KRB = 0)

R4 = Address of the status control block

R5 = Address of the unit control block

The driver may use all the registers. When you unload the driver, the UNLOAD routine calls it at the $xxUNL entry point with the same conditions.

These two entry points in the loadable driver are independent of the controller and unit status change entry points used by Executive reconfiguration software. That is, the two entry points $xxLOA and $xxUNL are used for initialization and completion at LOAD and UNLOAD time and not at on-line and off-line status change time.

4.4 Driver Data Structure Details

The following elements in the I/O data structure are of concern to the programmer writing a driver:

- The I/O packet
- The DCB
- The UCB
- The SCB
- The KRB
- The CTB

The I/O data structure and, in particular, the previously listed control blocks, contain an abundance of data pertaining to input/output operations. Drivers themselves are involved with only a subset of the data.

Note

Except where explicitly noted otherwise, all unused bits, fields, and words in all driver data base structures are reserved for DIGITAL system use and expansion.

Most data fields (words or bytes) are classified by one of five descriptors. Each descriptor is a different configuration of the following two conditions and indicates:

- Whether the field is initialized in the data-structure source
- What sort of access to the field the driver has during execution

The five descriptors are as follows:

<initialized, not referenced>
This field is supplied in the data-structure source code and is not referenced by the driver during execution.

<initialized, read-only>
This field is supplied in the data-structure source code and may be read by the driver.

<not initialized, read-only>
Either an agent other than the driver establishes this field or the driver sets it up once and thereafter references it read-only.

<not initialized, read/write>
Either the driver or some other agent establishes this field, and the driver may read it or write over it.

<not initialized, not referenced>
This field does not involve the driver in any way.

These descriptors cover most of the fields in the control blocks described in this section. No system software or hardware checks or enforces any of the access described. Exceptions are noted in the text.

4.4.1 The I/O Packet

Figure 4–1 shows the layout of the I/O packet, which is constructed and placed in the driver I/O queue by QIO directive processing and is subsequently delivered to the driver by a call to $GTPKT. The data processing block (DPB) from which the I/O packet is generated is illustrated in Section 4.4.2.

QIO directive processing dynamically builds the I/O packet from the data in the DPB. Fields in the I/O packet are classified as one of the following types:

- Not referenced
- Read-only
- Read/write

The fields in the I/O packet are as follows:

I.LNK
Links I/O packets queued for a driver. A zero ends the chain. The listhead is in the SCB (S.LHD).

Driver access: Not referenced.

I.EFN
Contains the event flag number as copied by QIO directive processing from the requester's DPB.

Driver access: Not referenced.

I.PRI
Priority copied from the TCB of the requesting task.

Driver access: Not referenced.

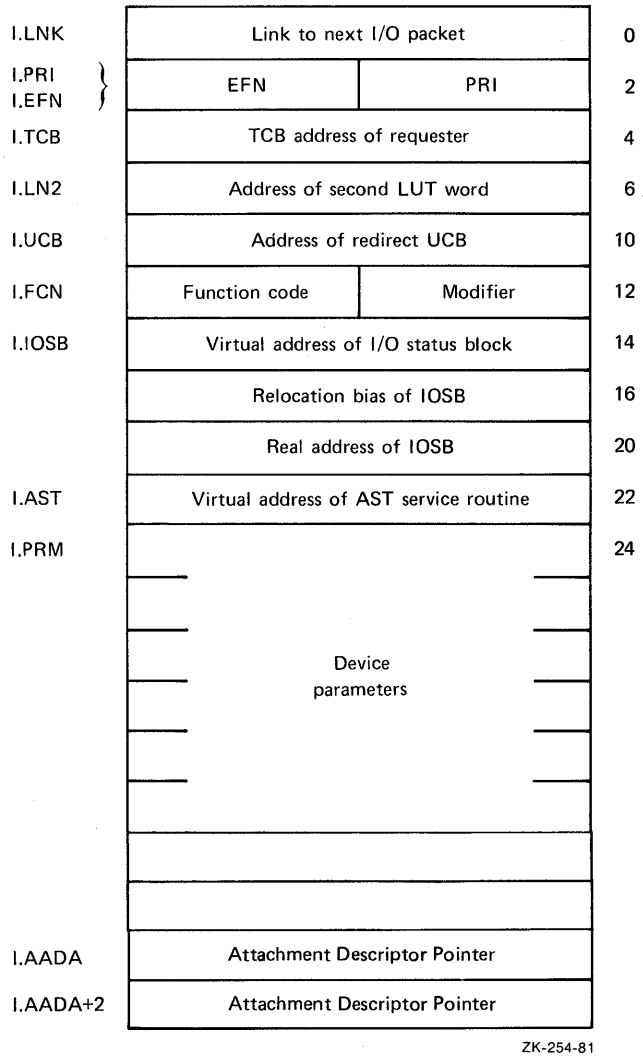
I.TCB
TCB address of the requesting task.

Driver access: Usually not referenced; sometimes referenced at I/O cancel and power failure.

I.LN2
Contains the address of the second word of the LUT entry in the task header to which the I/O request is directed. For open files on file-structured devices, this word contains the address of the Window Block; otherwise, it is zero.

Driver access: Not referenced.

Figure 4–1: I/O Packet Format



I.UCB

Contains the address of the unit to which I/O is to be directed. I.UCB is the address of the Redirect UCB if the starting UCB has been subject to an MCR Redirect command. The field is referenced by the $GTPKT routine.

Driver access: Not referenced by conventional driver; frequently referenced by full duplex drivers.

I.FCN

Contains the function code for the I/O request. It consists of two bytes. The high-order byte contains the function code; the low-order byte contains modifier bits. During predriver initiation the Executive compares the function code with a function mask value in the DCB. The driver interprets the modifier bits.

Driver access: Read-only.

I.IOSB

Contains the virtual address of the I/O status block (IOSB) if the IOSB is specified, or zero if the IOSB is not specified.

I.IOSB+2 and I.IOSB+4 contain the address doubleword for the IOSB (see Section 7.2 for a detailed description of the address doubleword). The first word contains the relocation bias of the IOSB; the bias is, in effect, the number of the 32-word block in which the IOSB starts.

The second word is formatted as follows:

Bits 0 through 5	Displacement in block (DIB)
Bits 6 through 12	All zeros
Bits 13 through 15	6

The displacement in block is the offset from the block base. The value 6 in bits 13 through 15 is constant. It is used to cause an address reference through kernel address page register 6 (APR 6).

Discussion of the address doubleword is deferred to Section 7.3 because you seldom have to be concerned with its contents or format in writing a conventional driver. Its construction and subsequent manipulation are normally external to the driver. Subroutines are provided as Executive services for programmed I/O to render the manipulations of I/O transfers transparent to the driver itself.

Driver access: Not referenced.

I.AST

Contains the virtual address of the AST service routine to be executed at I/O completion. If no address is specified, the field contains zero.

Driver access: Not referenced.

I.PRM

Device-dependent parameters constructed from the last six words of the DPB. Note that if the I/O function is a transfer (refer to the description of D.MSK in Section 4.4.3), the buffer address (first DPB device-dependent parameter) is translated to an equivalent address

doubleword. Therefore, the virtual buffer address, which occupied one word in the DPB, occupies two words in I.PRM. As a result, all other parameters in I.PRM are shifted by one word so that device-dependent parameter n is copied to I.PRM +(2 *n) +2.

Most DIGITAL-supplied drivers treat these words as a read/write storage area after their initial contents have been used.

When the last word of the device-dependent parameters is nonzero, the value can have one of several special meanings to the Executive. For example, if the value is nonzero and could be an Executive address, the Executive assumes that the value is a block locking word. Therefore, if the driver uses the word, it should restore its contents before calling $IODON.

Driver access: Read/write.

I.AADA
I.AADA+2

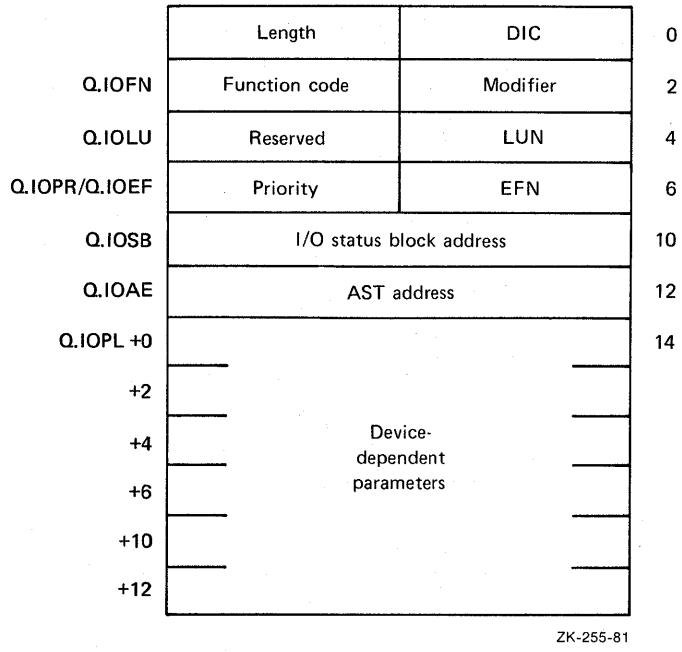
Two pointers, each to an attachment descriptor block of the region in which the task I/O buffer resides. These pointers account for I/O by region and enable the Executive to lock a region to make it noncheckpointable while I/O is in progress and to unlock a region after I/O completes.

Driver access: Not referenced; maintained by the Executive transparently to the driver.

4.4.2 The QIO Directive Parameter Block (DPB)

The QIO DPB is constructed as shown in Figure 4-2. Usually drivers never access the DPB; the information is supplied here for general reference.

Figure 4–2: QIO Directive Parameter Block (DPB)



Parameters

Length (required)

The length of the DPB, which for the RSX-11M and RSX-11M-PLUS QIO directive is always fixed at 12 words

DIC (required)

Directive Identification Code (For the QIO directive, this value is 1; for QIOW it is 3)

Q.IOFN (required)

The code of the requested I/O function (0 through 31)

Modifier

Device-dependent modifier bits

Reserved

Reserved byte; must not be used

Q.IOLU (required)
Logical Unit Number

Q.IOPR
Request priority (ignored by RSX-11M-PLUS, but space must be allocated for IAS compatibility)

Q.IOEF (optional)
Event flag number (zero indicates no event flag)

Q.IOSB (optional)
This word contains a pointer to the I/O status block, which is a two-word, device-dependent, I/O-completion data packet formatted as follows:

Byte 0	I/O status byte.
Byte 1	Augmented data supplied by the driver.
Bytes 2 and 3	The contents of these bytes depend on the value of byte 0. If byte 0 = 1, then these bytes usually contain the processed byte count. If byte 0 does not equal 0, then the contents are device-dependent.

Q.IOAE (optional)
Address of the user-written I/O done AST service routine

Q.IOPL
Up to six parameters specific to the device and to the I/O function to be performed. The following four are typically used for data transfer functions:

- Buffer address
- Byte count
- Carriage control type
- Logical block number

The fields for any optional parameters not specified must be filled with zeros.

4.4.3 The Device Control Block (DCB)

Figure 4–3 is a schematic layout of the DCB. The DCB describes the static characteristics of a device controller and the units attached to the controller. All fields must be specified.

The fields in the DCB are described as follows[1] :

[1] Some fields require that you initialize a value in the data base source code. The descriptions indicate the required values in parentheses, following the symbolic offset.

D.LNK (link to next DCB)

Address link to the next DCB. If this cell is in the last (or only) DCB, you should set its value to zero. If you are incorporating more than one user-written driver at one time, then this field should point to another DCB in a DCB chain, which is terminated by a value of zero.

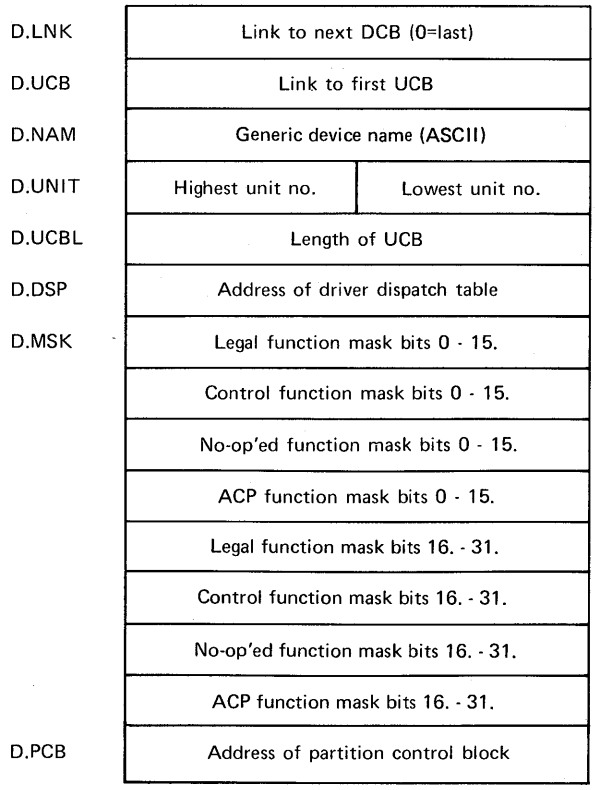
Driver access: Initialized, not referenced.

D.UCB (pointer to first UCB)

Address link to the U.DCB field of the first, and possibly the only, unit control block associated with the DCB. For a given DCB, all UCBs are in contiguous memory locations and must all have the same length.

Driver access: Initialized, not referenced.

Figure 4–3: Device Control Block



ZK-256-81

D.NAM (ASCII device name)

Generic device name in ASCII by which device units are mnemonically referenced.

Driver access: Initialized, not referenced.

D.UNIT (unit number range)

Unit number range for the device. The low-order byte contains the lowest unit number; the high-order byte contains the highest unit number. This range covers those logical units available to the user for device assignment. Typically, the lowest number is zero, and the highest is n −1, where n is the number of device-units described by the DCB.

Driver access: Initialized, not referenced.

D.UCBL (UCB length)

The unit control block can have any length to meet the needs of the driver for variable storage. However, all UCBs for a given DCB must have the same length. The specified length must include prefix words (such as U.LUIC and U.OWN), if present.

Driver access: Initialized, not referenced.

D.DSP (driver dispatch table pointer)

Address of the driver dispatch table, which is located within the driver code. (When the Executive wishes to enter the driver at any of the entry points contained in the driver dispatch table, it accesses D.DSP, locates the appropriate address in the table, and calls the driver at that address.) For a resident driver, your code references the symbol $xxTBL, which is generated by the DDT$ macro to mark the start of the driver dispatch table. For a loadable driver, then, you should initialize this field to zero, which indicates that the driver is not in memory.

Driver access: Initialized, not referenced.

D.MSK (driver-specific function masks)

Eight words, beginning at D.MSK, are critical to the proper functioning of a device driver. The Executive uses these words to validate and dispatch the I/O request specified by a QIO directive. The following description applies only to non-file-structured devices.[1] Four masks, with two words per mask, are described by the bit configurations that you establish for the following words:

- Legal function mask
- Control function mask
- No-op function mask
- Ancillary Control Processor (ACP) function mask

The QIO directive allows for 32 possible I/O functions. The masks, as stated, are filters to determine validity and I/O requirements for the subject driver.

[1] Although no DIGITAL publication describes writing drivers for file-structured devices (drivers that interface with F11ACP), you could write a disk driver by using a DIGITAL-supplied driver as a template. For example, the RK11 driver (DKDRV) is one that does not use advanced features.

The Executive filters the function code in the I/O request through the four masks. The I/O function code is the high-order byte of the function parameter issued with the QIO directive. The decimal representation of that high-order byte is equivalent to the decimal bit number of the mask. If you want the function to be true in one of the four masks, you must set the bit in that mask in the position that numerically corresponds to the function code. For example, the code for IO.RVB is 21_8 and its decimal representation is 17. If you want IO.RVB to be true for a mask, therefore, you must set bit number 17 in the mask.

The masks are laid out in memory in two 4-word groups. Each 4-word group covers 16 function codes. The first 4 words cover the function codes 0 through 15; the second 4 words cover codes 16 through 31. The exact layout used for the driver example in Chapter 8 is shown in the following source statements:

```
.WORD   177477          ; LEGAL FUNCTION MASK CODES 0-15.
.WORD   70              ; CONTROL FUNCTION MASK CODES 0-15.
.WORD   0               ; NO-OP FUNCTION MASK CODES 0-15.
.WORD   177200          ; ACP FUNCTION MASK CODES 0-15.
.WORD   377             ; LEGAL FUNCTION MASK CODES 16.-31.
.WORD   0               ; CONTROL FUNCTION MASK CODES 16.-31.
.WORD   0               ; NO-OP FUNCTION MASK CODES 16.-31.
.WORD   377             ; ACP FUNCTION MASK CODES 16.-31.
```

The Executive filters the function code through the mask words sequentially as follows:

Legal Function Mask

Legal function values have the corresponding bit position in this word set to 1. Function codes that are not legal are rejected by QIO directive processing, which returns IE.IFC in the I/O status block, provided the IOSB address was specified.

Control Function Mask

If any device-dependent data exists in the DPB and this data does not require further checking by the QIO directive processor, the function is considered to be a control function. Such a function allows QIO directive processing to copy the DPB device-dependent data directly into the I/O packet.

No-op Function Mask

A no-op function is any function that is considered successful as soon as it is issued. If the function is a no-op, QIO directive processing immediately marks the request successful; no additional filtering occurs.

ACP Function Mask

If a function code is legal but specifies neither a control function nor a no-op, then it specifies either an Ancillary Control Processor (ACP) function or a transfer function. If a function code requires intervention of an ACP, the corresponding bit in the ACP function mask must be set. ACP function codes must have a value greater than 7.

In the specific case of read/write virtual functions, the corresponding mask bits may be set at your option. If the corresponding mask bits for a read/write virtual function are set, QIO directive processing recognizes that a file-oriented function is being requested to a non-file-structured device and converts the request to a read/write logical function.

This conversion is particularly useful. Consider a read/write virtual function to a specific device, where the QIO$ directive processing has the following effect:

- If the device is file-structured and a file is open on the specified LUN, the block number specified is converted from a virtual block number in the file to a logical block number on the medium. Moreover, the request is queued to the driver as a read/write logical function.
- If the device is file-structured and no file is open on the specified LUN, then an error is returned and no further action is taken.
- If the device is not file-structured, then the request is simply transformed to a read/write logical function and is queued to the driver. (The specified block number is unchanged.)

Transfer Function Processing

Finally, if the function is not an ACP function, then it is by default a transfer function. All transfer functions cause the QIO directive processor to check the specified buffer for legality (that is, inclusion within the address space of the requesting task) and proper alignment (word or byte). In addition, the processor checks the number of bytes being transferred for proper modulus (that is, nonzero and a proper multiple). By convention, the first user-supplied parameter is the buffer address and the second is the byte count.

Creating Mask Words

Creating function mask words involves the following five steps:

1. Establish the I/O functions available on the device for which driver support is to be provided.
2. Build the legal function mask: Check the standard RSX-11M-PLUS function mask values in Table 4–3 for equivalences. Only the IO.KIL function is mandatory. IO.ATT and IO.DET functions, if used, must have the RSX-11M-PLUS system interpretation. DIGITAL suggests that functions having an RSX-11M-PLUS system counterpart use the RSX-11M-PLUS code, but this is required only when the device is to be used in conjunction with an ACP. From the supported function list in Table 4–2, you can build the two legal function mask words.
3. Build the control function mask by answering the following question:

   Does this function carry a standard buffer address and byte count in the first two device-dependent parameter words?

   If it does not, then either it qualifies as a control function or the driver itself must effect the checking and conversion of any addresses to the format required by the driver. See Section 8.3 for an example of a driver that does this. (Buffer addresses in standard format are automatically converted to Address Doubleword format.)

   Control functions are essentially those functions whose DPBs do not contain buffer addresses or counts.
4. Create the no-op function mask by deciding which legal functions are to be no-op. Typically, for compatibility with File Control Services (FCS) or Record Management Services (RMS) on non-file-structured devices, the file access/deaccess functions are selected as legal functions, even though no specific action is required to access or deaccess a non-file-structured device; thus, the access/deaccess functions are no-op.

5. Finally, include the ACP functions Write Virtual Block and Read Virtual Block for those drivers that support both read and write. (Include only one related ACP function if the driver supports only read or write). Other ACP functions that might be included fall into the nonconventional driver classification and are beyond the scope of this document.

D.PCB (0)

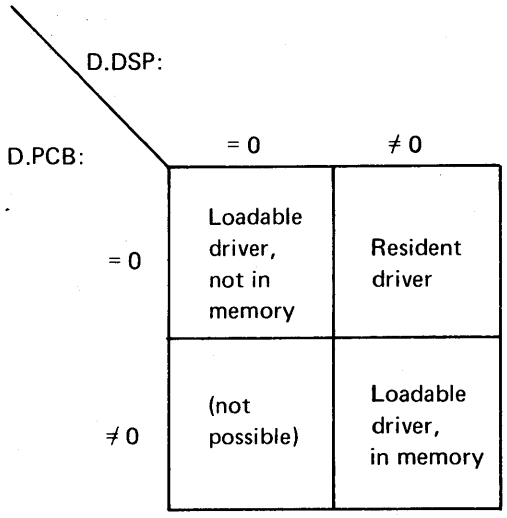
Address of the driver's partition control block (PCB). The driver data base source code must initialize the address to zero. The DCB can be extended by adding words after D.PCB.

A PCB exists for every partition in a system. A driver PCB describes the partition in which it resides.

The Executive uses D.PCB together with D.DSP (the address of the driver dispatch table) to determine whether a driver is loadable or resident and, if loadable, whether it is in memory. Zero and nonzero values for these two pointers have the meanings shown in Figure 4–4.

Driver access: Initialized, not referenced.

Figure 4–4: D.PCB and D.DSP Bit Meanings



4.4.3.1 Establishing I/O Function Masks

Table 4–2 can help you determine the proper values to set in the function masks. The mask values are given for each I/O function used by DIGITAL-supplied drivers. The bit number allows you to determine which mask group to use: for bits numbered 0 through 15, use the mask value for a word in the first 4-word group; for bits numbered 16 through 31, use the mask value for a word in the second 4-word group.

Of the function mask values listed in Table 4–2, only IO.KIL is mandatory and has a fixed interpretation. However, if IO.ATT and IO.DET are used, they must have the standard meaning. (Refer to the *RSX-11M-PLUS and Micro/RSX I/O Drivers Reference Manual* for a description of standard I/O functions.) If QIO directive processing encounters a function code of 3 or 4 and the code is not no-op, QIO assumes that these codes represent Attach Device and Detach Device,

respectively. The other codes are suggested but not mandatory. You are free to establish all other function-code values on non-file-structured devices. However, the mask words must still reflect the proper filtering process.

If you are writing a driver for a file-structured device, you must establish the standard function mask values of Table 4–2.

Table 4–2: Mask Values for Standard I/O Functions

Bit	Mask Value	Related Symbolic	I/O Function
0	1	IO.KIL	Cancel I/O
1	2	IO.WLB	Write Logical Block
2	4	IO.RLB	Read Logical Block
3	10	IO.ATT	Attach Device
4	20	IO.DET	Detach Device
5	40		General Device Control
6	100		General Device Control
7	200		General Device Control
8	400		Diagnostics
9	1000	IO.FNA	Find File in Directory
10	2000	IO.ULK	Unlock Block
11	4000	IO.RNA	Remove File from Directory
12	10000	IO.ENA	Enter file in Directory
13	20000	IO.ACR	Access File for Read
14	40000	IO.ACW	Access File for Read/Write
15	100000	IO.ACE	Access File for Read/Write/Extend
16	1	IO.DAC	Deaccess File
17	2	IO.RVB	Read Virtual Block
18	4	IO.WVB	Write Virtual Block
19	10	IO.EXT	Extend File
20	20	IO.CRE	Create File
21	40	IO.DEL	Mark File for Delete
22	100	IO.RAT	Read File Attributes
23	200	IO.WAT	Write File Attributes
24	400	IO.APC	ACP Control
25	1000		Unused
26	2000		Unused
27	4000		Unused
28	10000		Unused
29	20000		Unused
30	40000		Unused
31	100000		Unused

To determine the proper bit masks for disks use Table 4–3 as a guide.

Table 4–3: Mask Word Bit Settings for Disk Drives

Bit	Function	Related Symbolic
0	c	IO.KIL
1	t	IO.WLB
2	t	IO.RLB
3	c	IO.ATT
4	c	IO.DET
5	c	IO.STC
6		
7	sa	IO.CLN
8	sd	Diagnostic
9	a	IO.FNA
10	a	IO.ULK
11	a	IO.RNA
12	a	IO.ENA
13	a	IO.ACR
14	a	IO.ACW
15	a	IO.ACE
16	a	IO.DAC
17	a	IO.RVB
18	a	IO.WVB
19	a	IO.EXT
20	a	IO.CRE
21	a	IO.DEL
22	a	IO.RAT
23	a	IO.WAT
24	a	IO.APC
25		
26		
27		
28		
29		
30		
31		

t—transfer function, bit set only in legal function mask
c—control function, bit set in legal and control function masks
a—ACP function, bit set in legal and ACP function masks
sa—special case, bit set only in ACP function mask, but not legal
sd—special case, bit set only if diagnostic support in system and driver

To determine the proper bit masks for tapes use Table 4–4 as a guide.

Table 4–4: Mask Word Bit Settings for Magnetic Tape Drives

Bit	Function	Related Symbolic
0	c	IO.KIL
1	t	IO.WLB
2	t	IO.RLB
3	c	IO.ATT
4	c	IO.DE
5	c	IO.STC
6	c	
7	sa	IO.CLN
8	sd	Diagnostic
9	a	IO.FNA
10		IO.ULK
11		IO.RNA
12	n	IO.ENA
13	a	IO.ACR
14	a	IO.ACW
15	a	IO.ACE
16	a	IO.DAC
17	a	IO.RVB
18	a	IO.WVB
19	a	IO.EXT
20		IO.CRE
21		IO.DEL
22	a	IO.RAT
23		IO.WAT
24	a	IO.APC
25		
26		
27		
28		
29		
30		
31		

t—transfer function, bit set only in legal function mask
c—control function, bit set in legal and control function masks
n—no-op function, bit set in legal and no-op function masks
a—ACP function, bit set in legal and ACP function masks
sa—special case, bit set only in ACP function mask, but not legal
sd—special case, bit set only if diagnostic support in system and driver

To determine the proper bit masks for unit record devices (such as terminals, card readers, line printers, paper tape punches, and readers), use Table 4–5 as a guide.

Table 4–5: Mask Word Bit Settings for Unit Record Devices

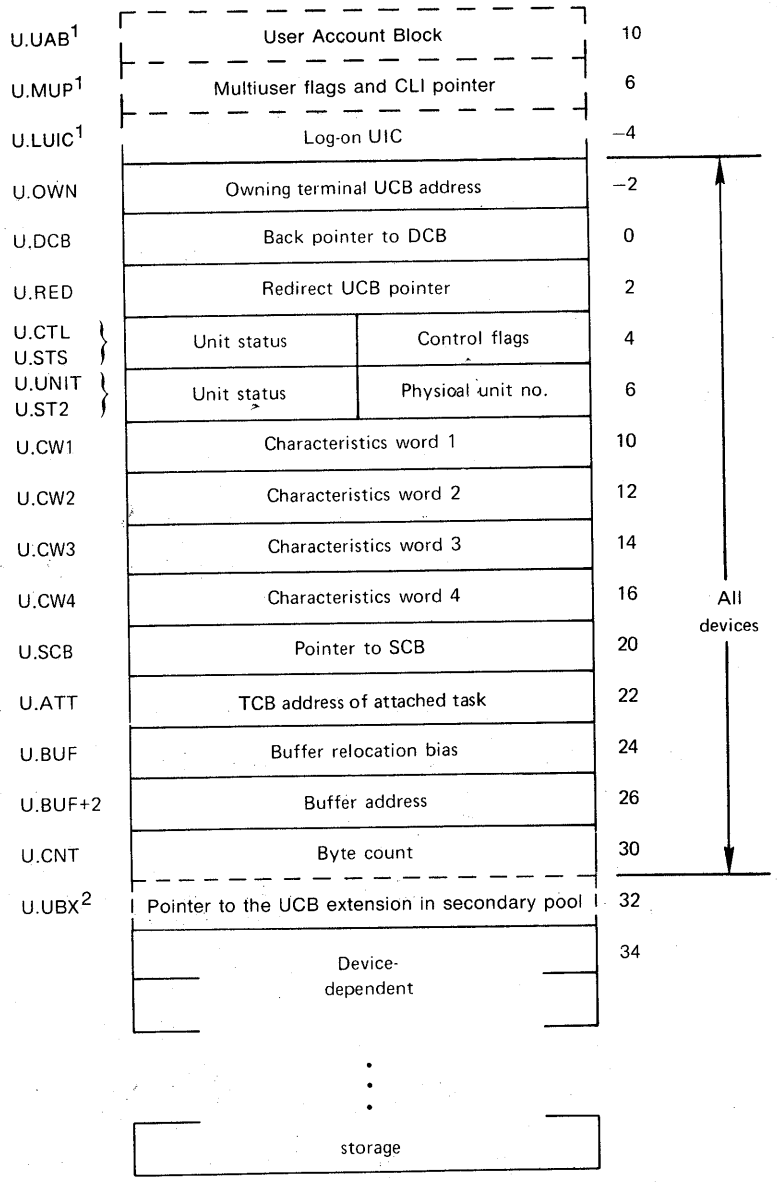
Bit	Function	Related Symbolic
0	c	IO.KIL
1	t	
2	t	IO.RLB
3	c	IO.ATT
4	c	IO.DE
5	c	IO.STC
6		
7	sa	IO.CLN
8	sd	Diagnostic
9	a	IO.FNA
10	a	IO.ULK
11	a	IO.RNA
12	a	IO.ENA
13	a	IO.ACR
14	a	IO.ACW
15	a	IO.ACE
16	a	IO.DAC
17	a	IO.RVB
18	a	IO.WVB
19	a	IO.EXT
20	a	IO.CRE
21	a	IO.DEL
22	a	IO.RAT
23	a	IO.WAT
24	a	IO.APC
25		
26		
27		
28		
29		
30		
31		

t—transfer function, bit set only in legal function mask
c—control function, bit set in legal and control function masks
a—ACP function, bit set in legal and ACP function masks
sa—special case, bit set only in ACP function mask, but not legal
sd—special case, bit set only if diagnostic support in system and driver

4.4.4 The Unit Control Block (UCB)

Figure 4–5 is a layout of the UCB (a variable-length control block). One UCB exists for each physical device-unit generated into a system configuration. For user-added drivers, this control block is defined as part of the source code for the driver data structure.

Figure 4–5: Unit Control Block



1. This offset appears only for terminal devices (that is, devices that have DV.TTY set) in multiuser systems.
2. This offset appears only for those devices that have DV.MSD set.

ZK-257-81

The fields in the UCB are described as follows[1] :

U.UAB (0)

This field is for terminal UCBs only. It is required only if accounting support is on the system (A$$CNT is defined) but may be present if accounting support is not on the system. This value is used to access the user accounting block in secondary pool.

Driver access: Initialized, not referenced.

U.MUP

For terminal UCBs only. Bits 1 to 4 contain an index to a table which contains the address of the CLI parser block (CPB) for the current CLI; the remaining bits are used for other terminal-specific features and are defined as follows:

UM.OVR	Override CLI indicator
UM.CLI	CLI indicator
UM.DSB	Terminal disabled because CLI eliminated
UM.NBR	No broadcast
UM.CNT	Continuation of command line in progress
UM.CMO	Command is in progress from this terminal
UM.SER	Terminal is in serial mode
UM.KIL	TTDRV should tell MCR to flush all pieces of a continued command if the user types CTRL/C

Driver access: Not initialized, not referenced.

U.LUIC

For terminal UCBs only: the login UIC of the user at the particular terminal. This offset must exist for any device on a multiuser system for which the DV.TTY bit is set. This word is altered by logging in to the system.

Driver access: Not initialized, not referenced.

U.OWN (0)

The UCB address of the owning terminal for allocated devices.

Driver access: Initialized, not referenced.

U.DCB (pointer to associated DCB)

This word is a pointer to the corresponding device control block. Because the UCB is a key control block in the I/O data structure, access to other control blocks usually occurs by means of links implanted in the UCB.

Driver access: Initialized, not referenced.

U.RED (pointer to start of this UCB(.-2))

Contains a pointer to the unit control block to which this device-unit has been redirected. This field is updated as the result of an MCR Redirect command. The redirect chain ends when this word points to the beginning of the UCB itself (U.DCB of the UCB, to be precise).

Driver access: Initialized, not referenced.

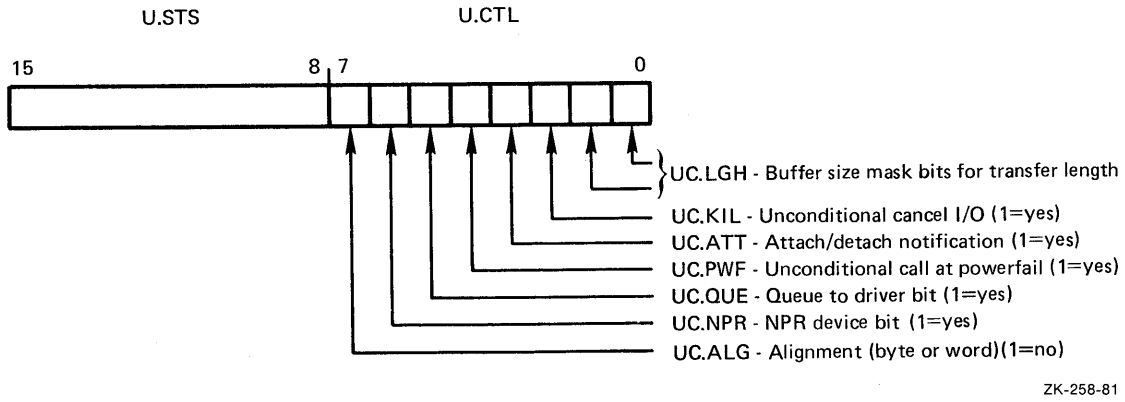
[1] Some fields require that you initialize a value in the data base source code. The descriptions indicate the required values in parentheses, following the symbolic offset.

U.CTL (device-dependent values)

U.CTL and the function mask words in the device control block control QIO directive processing. Figure 4–6 shows the layout of the unit control byte.

Driver access: Initialized, not referenced.

Figure 4–6: Unit Control Byte



The driver data base code statically establishes this bit pattern. Any inaccuracy in the bit setting of U.CTL produces erroneous I/O processing. The bit symbols and their meanings are described in Table 4–6.

Table 4–6: Unit Control Byte Bit Symbols

Bit Symbol	Function	Description
UC.ALG	Alignment bit	If this bit is 0, then byte alignment of data buffers is allowed. If UC.ALG is 1, then buffers must be word-aligned. UC.ALG and UC.LGH are independent settings.
UC.ATT	Attach/Detach notification	If this bit is set, then the driver is called when $GTPKT processes an Attach/Detach I/O function. Typically, the driver does not need to obtain control for Attach/Detach requests, and the Executive performs the entire function without any assistance from the driver.
UC.KIL	Unconditional Cancel I/O call bit	If set, the driver is called on a Cancel I/O request, even if the unit specified is not busy. Typically, the driver is called on Cancel I/O only if an I/O operation is in progress. In any case, the Executive flushes the I/O queue.

Table 4–6 (Cont.): Unit Control Byte Bit Symbols

Bit Symbol	Function	Description
UC.QUE	Queue-to-driver bit	If set, the QIO directive processor calls the driver at its I/O initiation entry point without queuing the I/O packet. After the processor makes this call, the driver is responsible for the disposition of the I/O packet. Typically, the processor queues an I/O packet before calling the driver, which later retrieves it by a call to $GTPKT. The most common reason for a driver to examine a packet before queuing is that the driver employs a special user buffer, other than the normal buffer used in a transfer request. Within the context of the requesting task, the driver must address-check and relocate such a special buffer. See Section 8.3 for an example of a driver that does this. On multiprocessor systems, certain restrictions apply to this form of I/O processing. No driver should process an I/O packet received directly from the QIO processor without first performing a conditional fork operation (that is, call $CFORK) to guarantee execution on the correct processor. Unless the driver is running on the correct processor, it must not process a packet that causes access to the device registers. The restriction does not apply if the driver merely uses the current task context to map secondary I/O buffers and then queues the I/O packet itself. In summary, packets received directly from $DRQIO may not be processed directly unless they cause no activity on the I/O page (and thereby do not need to be executed on a particular processor) or unless an intervening call to $CFORK has been performed.
UC.PWF	Unconditional call on power failure bit	If set and the unit is on-line, the driver is always to be called when power is restored after a power failure occurs. Typically, the driver is called on power restoration only when an I/O operation is in progress. See the discussion in Sections 4.3.6 and 4.5 of the entry points in the DDT for LOAD and UNLOAD and for controller and unit status change.
UC.NPR	NPR device bit	If set, the device is an NPR device. This bit determines the format of the two-word address in U.BUF (details given in the discussion of U.BUF below).

Table 4–6 (Cont.): Unit Control Byte Bit Symbols

Bit Symbol	Function	Description
UC.LGH	Buffer size mask bits (two bits)	These two bits are used to check whether the byte count specified in an I/O request is a legal buffer modulus. You select one of the values below by ORing into the byte a 0, 1, 2, or 3. 00—Any buffer modulus valid 01—Must have word alignment modulus 10—Combination invalid 11—Must have double word-alignment modulus UC.ALG and UC.LGH are independent settings.

Note

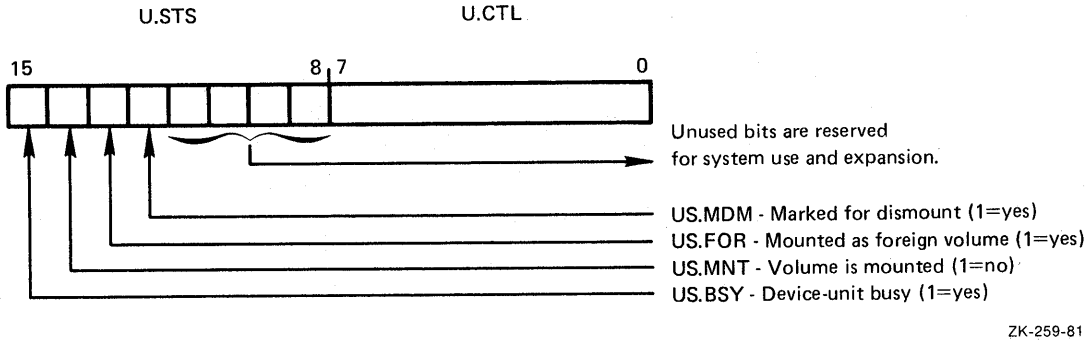
UC.ATT, UC.KIL, UC.QUE, and UC.PWF are usually zero, especially for conventional drivers. Every driver, however, must be concerned with its particular values for UC.ALG, UC.NPR, and UC.LGH. The driver is totally responsible for the values in these bits, and erroneous values are likely to produce unpredictable results.

U.STS (0)

This byte contains device-independent status information. Refer to the UCBDF$ macro definition in Appendix A. Figure 4–7 shows the layout of the unit status byte.

Driver access: Initialized, not referenced.

Figure 4–7: Unit Status Byte



US.MDM, US.MNT, and US.FOR apply only to mountable devices.[1]

The bit meanings are as follows:

[1] If your user-written driver services a mountable device, refer to Section 4.5.12 for information on volume valid processing.

US.BSY	If set, device-unit is busy
US.MNT	If set, volume is not mounted
US.FOR	If set, volume is mounted foreign
US.MDM	If set, device is marked for dismount

U.UNIT (unit number)

This byte contains the physical unit number of the device unit serviced by this UCB. If the controller for the device supports only a single unit, the unit number is always zero.

This is the physical unit number of the device and not the logical unit number. The range of this number is from zero to n, where n is device dependent. The logical designation DB0: does not necessarily imply a zero in this byte.

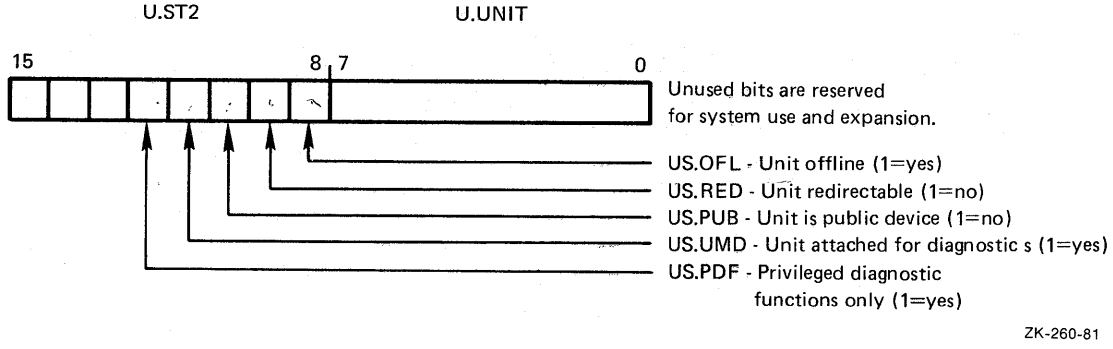
Driver access: Initialized, read-only.

U.ST2 (US.OFL)

This byte contains additional device independent status information. Different parts of the system set and clear these bits. The layout of the unit status extension byte is shown in Figure 4–8.

Driver access: Initialized, not referenced.

Figure 4–8: Unit Status Extension 2



The bit meanings are as follows:

US.OFL	=	1	If set, the device is off-line (that is, not in the configuration). This bit should be initialized to 1.
US.RED	=	2	If set, the device cannot be redirected.
US.PUB	=	4	If set, the device is not a public device.
US.UMD	=	10	If set, the device is attached for diagnostics.
US.PDF	=	20	If set, this unit can be used for a privileged diagnostic function only.

U.CW1 (device specific characteristics)

The first of a 4-word continuous cluster of device characteristics information. U.CW1 and U.CW4 are device independent, whereas U.CW2 and U.CW3 are device dependent. The four characteristics words are retrieved from the UCB and placed in the requester's buffer on issuance of a Get LUN information (GLUN$) Executive directive. It is your responsibility to supply the contents of these four words in the assembly source code of the data structure.

Driver access: Initialized, not referenced.

U.CW1 is defined as follows (if a bit is set to 1, the corresponding characteristic is true for the device):

DV.REC	=	1	Record oriented device
DV.CCL	=	2	Carriage control device
DV.TTY	=	4	Terminal device (If DU.TTY is set, then the UCB contains extra cells for U.LUIC, U.CLI, and optionally U.UAB.)
DV.DIR	=	10	Directory device
DV.SDI	=	20	Single directory device
DV.SQD	=	40	Sequential device
DV.MSD	=	100	Mass Storage device
DV.UMD	=	200	Device supports user-mode diagnostics
DV.EXT	=	400	Unit is on an extended 22-bit controller
DV.SWL	=	1000	Unit is software write-locked
DV.ISP	=	2000	Input spooled device
DV.OSP	=	4000	Output spooled device
DV.PSE	=	10000	Pseudo device (If this bit is set, the UCB does not extend past the U.CW1 offset.)
DV.COM	=	20000	Device mountable as a communications channel
DV.F11	=	40000	Device mountable as a FILES-11 device
DV.MNT	=	100000	Device mountable

U.CW2 (device-specific characteristics)

Specific to a given device driver (available for working storage or constants).[1]

Driver access: Initialized, read/write.

U.CW3 (device-specific characteristics)

Specific to a given device driver (available for working storage or constants).[1]

Driver access: Initialized, read/write.

U.CW4 (device-specific characteristics)

Default buffer size in bytes. This word is changed by a system command (SET with the /BUF keyword). The value in this word affects FCS, RMS, and many utility programs.

Driver access: Initialized, read-only.

[1] An exception is that, for block-structured devices, U.CW2 and U.CW3 may not be used for working storage. In drivers for block-structured devices (disks and DECtape), these two words must be initialized to a double-precision number giving the total number of blocks on the device. Place the high-order bits in the low-order byte of U.CW2 and the low-order bits in U.CW3.

U.SCB (SCB pointer)

This field contains a pointer to the status control block for this UCB. In general, Register 4 contains the value in this word when the driver is entered by way of the driver dispatch table, because service routines frequently reference the SCB.

Driver access: Initialized, read-only.

U.ATT (0)

If a task has attached itself to the device-unit, this field contains its task control block address.

Driver access: Initialized, not referenced.

U.BUF (reserve two words of storage)

U.BUF labels two consecutive words that serve as a communication region between $GTPKT and the driver. If a nontransfer function is indicated (in D.MSK), then U.BUF, U.BUF+2, and U.CNT receive the first three parameter words from the I/O packet.

For transfer operations, the initial format of these two words depends on the setting of UC.NPR in U.CT. The driver does not format the words; all formatting is completed before the driver receives control. The format is determined by the UC.NPR bit, which is set for an NPR device and reset for a program-transfer device.

The format for program-transfer devices is identical to that for the second two words of I.IOSB in the I/O packet. See Section 4.4.1 for a description of I.IOSB in the I/O packet.

In general, the driver does not manipulate these words when performing I/O to a program-transfer device. Instead, it uses the Executive routines Get Byte ($GTBYT), Get Word ($GTWRD), Put Byte ($PTBYT), and Put Word ($PTWRD) to effect data transfers between the device and the user's buffer.

For non-processor request (NPR) device drivers, these two words represent what the driver uses to initiate the transfer operation. For both UNIBUS and MASSBUS NPR devices, word 2 contains the low-order 16 bits of the physical address. For a UNIBUS NPR device, bits 4 and 5 in word 1 are memory extension bits; for a MASSBUS or 22-bit Q BUS NPR device (the KS.MBC bit is set), bits 8 through 13 are the memory extension bits. It is the driver's responsibility to set the function code, interrupt enable, and go bits. This action must be accomplished by a Bit Set (BIS) operation so that the extension bits are not disturbed. The driver must move these words into the device control registers to initiate the I/O operation.

For a typical UNIBUS NPR device driver, the word layout is as follows:

Word 1	
Bit 0	Go bit initially set to zero
Bits 1,2,3	Function code–set to zeros
Bits 4,5	Memory extension bits
Bit 6	Interrupt enable–set to zero
Bits 7–15	Zero
Word 2	
Bits 0–15	Low-order 16 bits of physical address

The construction of U.BUF, U.BUF+2, and U.CNT occurs only if the request function is a transfer function; if it is not, these three words contain the first three words of the device parameter list in the I/O packet. (See Figure 4–1.)

The details of the construction of the Address Doubleword appear in Section 7.2.

Driver access: Not initialized, read/write.

U.CNT (reserve one word of storage)

Contains the byte count of the buffer described by U.BUF. The driver uses this field in constructing the actual device request.

U.BUF and U.CNT keep track of the current data item in the buffer for the current transfer (except for NPR transfers). Because this field is being altered dynamically, the I/O packet may be needed to reissue an I/O operation (for instance, after a powerfail or error retry).

Driver access: Not initialized, read/write.

U.UCBX

This field contains a pointer to the UCB extension in secondary pool for mass storage devices with DV.MSD set (DV.MSD=1). For information on formatting, see the description of the UCBDF$ macro.

Driver access: Not initialized, not referenced.

U.PRM (device-dependent words)

The driver establishes this variable-length block of words to suit device-specific requirements. For example, a disk driver uses the first words to store the disk geometry as follows:

```
.BLKB 1                 ; # OF SECTORS PER TRACK
.BLKB 1                 ; # OF TRACKS PER CYLINDER
.BLKW 1                 ; # OF CYLINDERS PER VOLUME
```

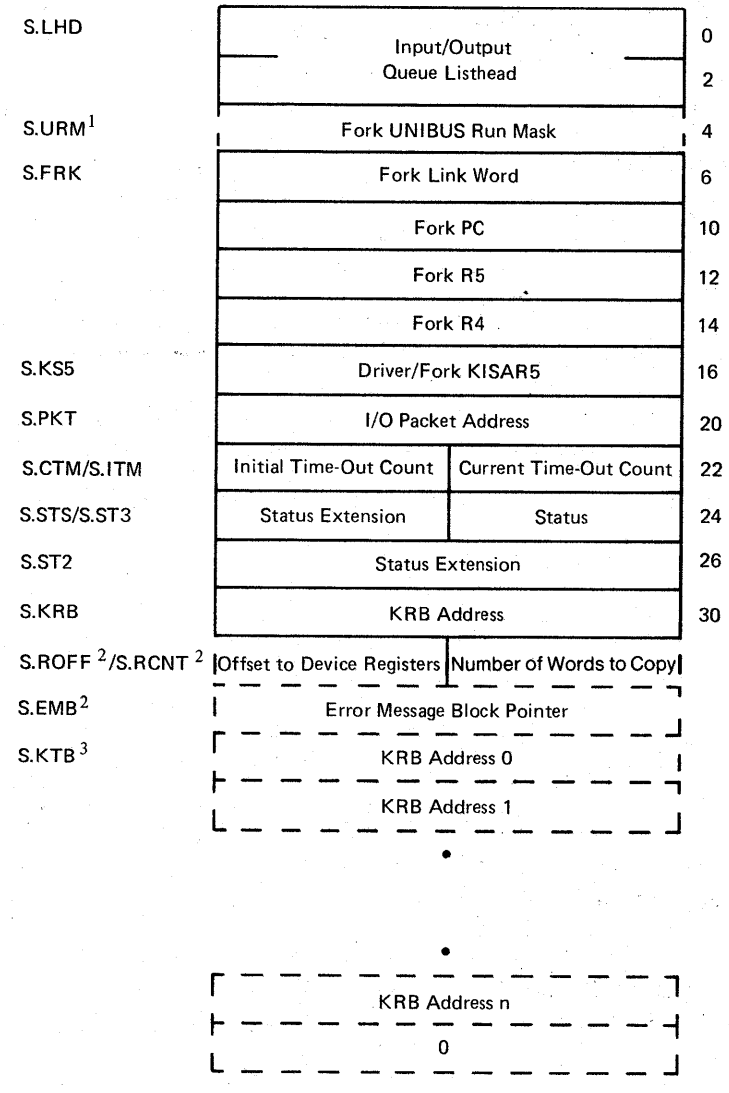
The driver can call the $CVLBN routine (described in Chapter 7) to convert a logical block number to a disk address based on the values in U.PRM and U.PRM+2.

Driver access: Not initialized, read/write.

4.4.5 The Status Control Block (SCB)

Figure 4–9 is a layout of the SCB. The SCB contains the context for a unit operation and describes the status of a unit that can run in parallel with all other units.

Figure 4–9: Status Control Block



If the symbols below are defined at system generation time, the related cells marked with a number appear in the structure.

[1] Multiprocessor support (M$$PRO)

[2] Appears only if driver supports error logging

[3] If the system has multiaccess device support (M$$ACD) and the driver is multiaccess (S2.MAD)

ZK-261-81

The fields in the SCB are described as follows[1] :

S.LHD (first word equals zero; second word points to first)

Two words forming the I/O queue listhead. The first word points to the first I/O packet in the queue, and the second word points to the last I/O packet in the queue. If the queue is empty, the first word is zero, and the second word points to the first word.

Driver access: Initialized, not referenced.

S.URM (controller UNIBUS run mask)

This word appears only in a multiprocessor system (that is, one on which M$$PRO is defined). It contains a UNIBUS run mask that defines the UNIBUS run to which the currently assigned controller is attached. When controller assignment is made, this cell is set from K.URM; when running a driver on the correct processor, S.URM is used exclusively and independently of the value of S.KRB or K.URM. If S.KRB is not equal to zero, and if S.URM is not equal to K.URM (an unusual situation), then the driver must take into account the fact that it will run on a different processor from the one its currently assigned KRB would normally warrant. It is possible that the processor on which the driver will run has the CSRs at a location different from that stored in the current KRB.

Note that S.URM must be adjacent to the fork block (S.FRK).

Driver access: Initialized, not referenced.

S.FRK (reserve four words of storage)

The four words starting at S.FRK are used for fork-block storage if and when the driver deems it necessary to establish itself as a fork process. Fork-block storage preserves the state of the driver, which is restored when the driver regains control at fork level. This area is automatically used if the driver calls $FORK.

Note that in order for your driver to use the fork processor, S.URM and S.KS5 must be adjacent if the required support is generated into the system.

Driver access: Initialize words to zero, not referenced.

S.KS5 (0)

This word contains the contents of KISAR5 necessary to correctly alter the Executive mapping to reach the driver for this unit. It has no meaning for a driver that is not loadable. It is set by LOAD, and whenever a fork block is dequeued and executed, this word is unconditionally inserted into KISAR5.

Note that S.KS5 must be adjacent with the fork block.

Driver access: Initialized, not referenced.

S.PKT (reserve one word of storage)

Address of the current I/O packet established by $GTPKT. The Executive uses this field to retrieve the I/O packet address upon the completion of an I/O request. S.PKT is not modified after the packet is completed.

Driver access: Not initialized, read-only.

[1] Some fields require that you initialize a value in the data base source code. The descriptions indicate the required values in parentheses following the symbolic offset.

S.CTM (0)

RSX-11M-PLUS supports device timeout, which enables a driver to limit the time that elapses between the issuing of an I/O operation and its termination. The current timeout count (in seconds) is typically initialized by moving S.ITM (initial timeout count) into S.CTM. The Executive clock service (in module TDSCH) examines active times, decrements them, and, if they reach zero, calls the driver at its device timeout entry point.

The internal clock count is kept in one-second increments. Thus, a time count of 1 is not precise because the internal clocking mechanism is operating asynchronously with driver execution. The minimum meaningful clock interval is 2 if you intend to treat timeout as a consistently detectable error condition. If the count is zero, then no timeout occurs; a zero value is, in fact, an indication that timeout is not operative. The maximum count is 250. The driver is responsible for setting this field. Resetting occurs at actual timeout or within $FORK and $IODON.

Driver access: Not initialized, read/write.

S.ITM (initial timeout count)

Contains the initial timeout value that the driver can load into S.CTM to begin device timeout.

Driver access: Initialized, read-only.

S.STS (0)

Establishes the controller as busy/not busy (nonzero/zero). This byte is the interlock mechanism for marking a driver as busy for a specific controller. The byte is tested and set by $GTPKT and reset by $IODON.

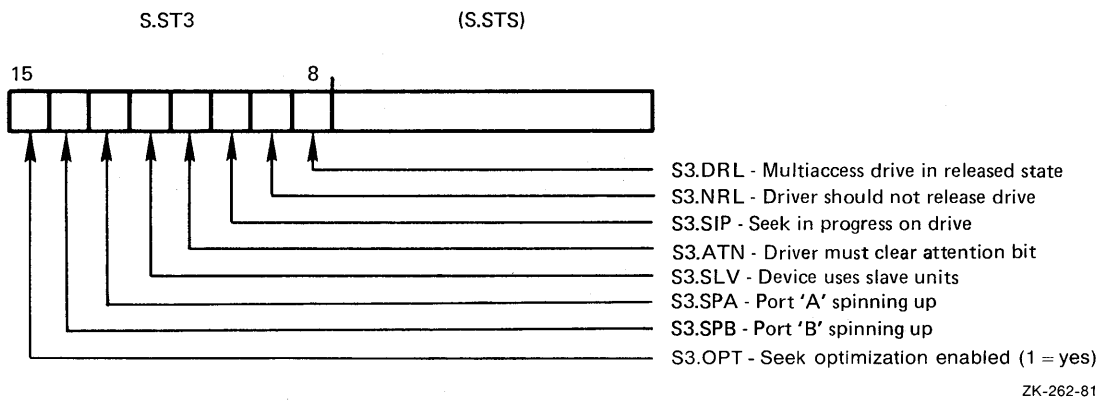
Driver access: Initialized, not referenced.

S.ST3 (driver-specific status byte)

This status byte is reserved for driver-specific bits concerning driver-to-executive or driver-to-driver communication. Figure 4–10 shows the layout of this byte.

Driver access: Initialized, referenced by driver for synchronization.

Figure 4–10: Controller Status Extension 3



All currently defined bits are used by mass storage devices. Table 4–7 describes the currently defined bits for this status byte.

Table 4–7: Controller Status Extension 3-Bit Settings

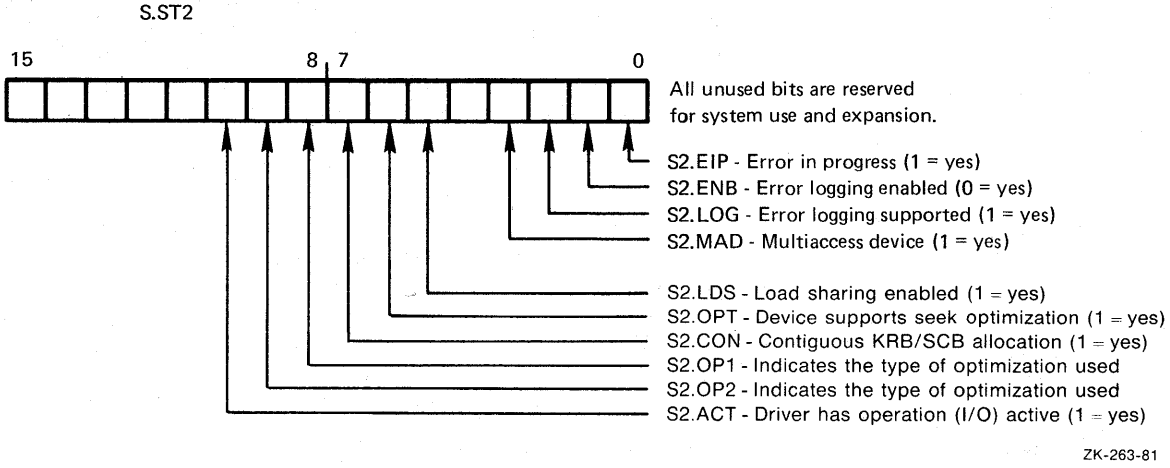
S3.DRL	=	1	If this bit is set, the drive is in the released state. Drivers that support dual-access (dual-port) operation set this bit after completion of the release command by the drive. The Executive routines Request Controller for Control Function ($RQCNC) and Request Controller for Data Transfer ($RQCND) test this bit to determine whether the drive is in a released state and whether the Executive should attempt load balancing by switching ports.
S3.NRL	=	2	If this bit is set, a driver does not release the drive. This bit exists solely for DIGITAL to maintain the device and to debug the driver. Drivers that support dual-access (dual-port) operation examine this bit and, if it is set, do not issue the release command to the drive and do not set the bit. If this bit is set, reconfiguration dual-port activity (that is, port on-line and off-line operations) will not function properly.
S3.SIP	=	4	If this bit is set, the drive has a seek in progress. A driver that supports overlapped seek operations examines this bit to keep track of whether the drive is seeking. For a driver that does not support overlapped operations but does support error logging (that is, cassette and magtape), this bit is set to indicate that a positioning operation is in progress.
S3.ATN	=	10	This bit is used only by MASSBUS devices. The Executive common interrupt module DVINT checks this bit; if it is set, then the driver must clear the attention bit in the Attention Summary Register. If this bit is not set, DVINT itself clears the attention bit in the Attention Summary Register.
S3.SLV	=	20	If this bit is set, the device connects to slave units. Certain devices, such as magnetic tape controllers attached to a MASSBUS controller, can in turn have units attached to them. These units are referred to as slave units. Thus, if this bit is set, the SCB describes a tape controller to which slave units can be attached.
S3.SPA	=	40	If this bit is set, port A on this unit is spinning up.
S3.SPB	=	100	If this bit is set, port B on this unit is spinning up.
S3.OPT	=	200	If this bit is set, seek optimization is enabled for this device. $GTPKT uses this bit to determine whether optimization is to be used. An MCR SET command can set and clear this bit. If you select seek optimization support for a DIGITAL-supplied device during system generation, SYSGEN sets this bit in that device SCB when it creates the device data base structures.

S.ST2 (controller status extension)

This status word defines certain status conditions for the controller-unit combination. Figure 4–11 shows the layout of this word.

Driver access: Initialized.

Figure 4–11: Controller Status Extension 2



DIGITAL has attempted to restrict bits in this word to those defining system-wide status. Specific bits for driver and Executive synchronization or driver internal synchronization are allocated from S.ST3. Table 4–8 describes the currently defined bits for this status byte.

Table 4–8: Controller Status Extension 2-Bit Settings

S2.EIP	=	1	This bit is reserved for DIGITAL error logging routines.
S2.ENB	=	2	This bit is reserved for DIGITAL error logging routines.
S2.LOG	=	4	This bit is reserved for DIGITAL error logging routines.
S2.MAD	=	10	This bit indicates the presence of the table of KRB addresses at the end of the Status Control Block. If this bit is set, the device is a multiaccess device and the SCB has a KRB table containing pointers to the KRBs of the controllers capable of accessing the device.
S2.LDS	=	40	This bit enables and disables load sharing for dual-access devices. If this bit is set, the Executive may dynamically switch ports and therefore alter controller assignment when establishing an access path for a driver. If this bit is not enabled, the Executive does not alter the current controller assignment. This feature permits status controller assignment; for example, for diagnostic operations. Devices (such as terminals) with S2.LDS clear have drivers that explicitly manage controller assignment.
S2.OPT	=	100	If this bit is set, this device supports queue optimization. This bit, used by $DRQRQ, determines whether to call the block check and convert the LBN routine in the driver.

Table 4–8 (Cont.): Controller Status Extension 2-Bit Settings

S2.CON	=	200	This bit indicates the continuous allocation of the controller request and status control blocks. Devices that do not support overlapped operation do not require a separate SCB for each unit. The KRB and SCB for such devices can be contiguous and some fields in the SCB overlap those in the KRB. Therefore, the SCB offsets S.CSR, S.PRI, S.VCT, and S.CON are valid only for such devices. For these devices, S2.CON is set. For the layout of the contiguous KRB and SCB, refer to Section 4.4.7.
S2.OP1 S2.OP2	= =	400 1000	These bits indicate the type of optimization selected for this device. An MCR command can set and clear these bits. These two bits give you three options of queue optimization. The options are as follows: S2.OP2,S2.OP1 = 0,0 (Nearest cylinder) S2.OP2,S2.OP1 = 0,1 (Elevator) S2.OP2,S2.OP1 = 1,0 (CSCAN) S2.OP2,S2.OP1 = 1,1 (Reserved)
S2.ACT	=	2000	If this bit is set, the driver has active I/O.

S.KRB (pointer to currently assigned KRB)

This word points to the currently assigned controller request block. For non-multiaccess devices, it is set during system generation and never altered. For multiaccess devices with load-sharing enabled, it may take on the value of one of the KRB pointers in the KRB table, S.KTB. If this word has a value of zero, then the device has no currently assigned KRB. It may, in fact, not have a KRB or CTB at all. Both the null driver and virtual terminal driver have no KRB.

Certain restrictions apply to drivers whose data bases do not include KRBs. They will receive powerfail, timeout, and cancel calls like any other driver, but the priority will always be zero, and the CSR address and controller index (where supplied) will be undefined.

Note

All code that checks S.KRB for a KRB pointer must check for a possible zero value and take appropriate action. A zero value in S.KRB does not necessarily mean that a KRB does not exist, but perhaps only that one is not currently assigned. A device which has no KRB will not have S2.CON set.

The first cell in the KRB (K.CSR) contains the control and status register (CSR) address for the controller. The offset K.CSR will always be zero so that the pointer (S.KRB) will always connect directly to the cell containing the CSR address.

Driver access: Initialized, referenced by driver to access the KRB.

S.ROFF

This byte is reserved for devices that support DIGITAL error logging software. This value is an offset from S.CSR/K.CSR to indicate the start of the device registers. It is typically zero.

S.RCNT

This byte is reserved for devices that support DIGITAL error logging software. It represents the minimum number of words of I/O page registers that this device has.

S.EMB

This word is reserved for devices that support DIGITAL error logging software.

S.KTB (KRB addresses)

This table appears only if the system has multiaccess device support (M$$ACD is defined) and the device is multiaccess (the S2.MAD bit set).

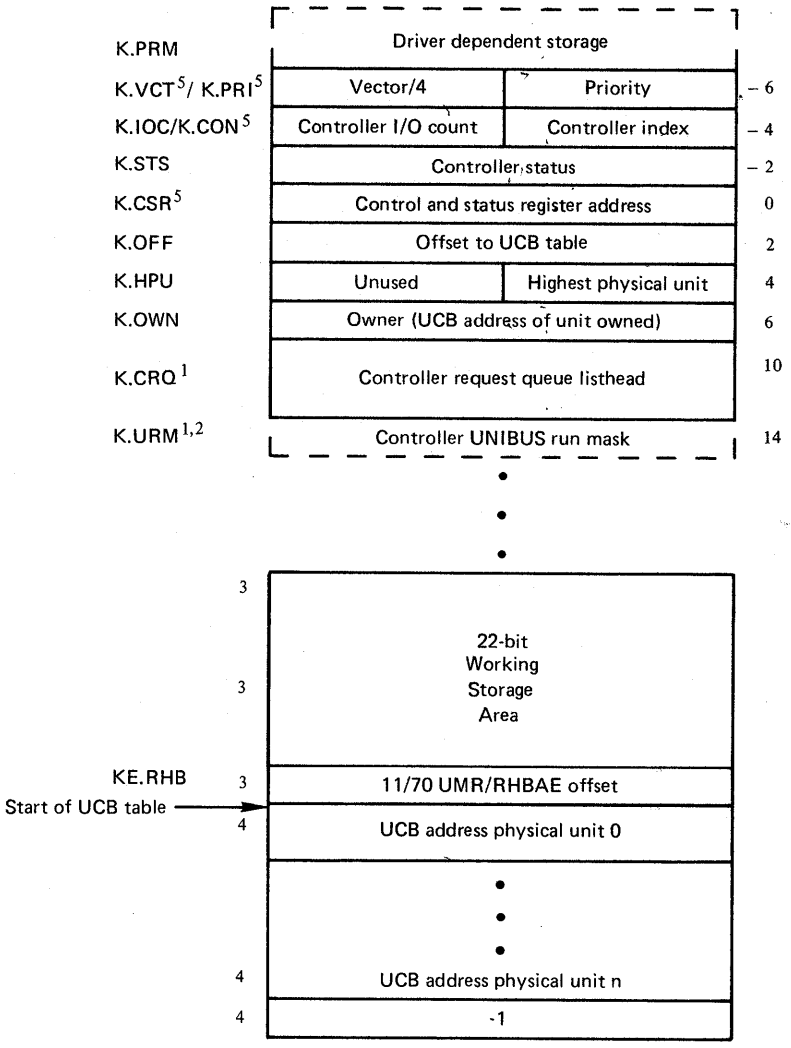
Every controller with which the unit (unit control block and status control block combination) can communicate is represented in this table by a controller request block address. The table contains at least two entries, with the list terminated by a zero word. For devices with executive load sharing supported (S2.LDS set), bit zero of each word is an on-line and off-line flag which, when set, indicates that KRB is off-line with respect to this SCB and should not be considered for controller assignment. Devices with S2.LDS clear have drivers that explicitly manage controller assignment. Only the driver may change S.KRB, and it may or may not use the low-order bit of the KRB addresses in S.KRB as an on-line and off-line flag. When drivers explicitly manage controller assignment, system software (other than the driver) must not modify S.KRB and must tolerate a 1 in the low-order bit of the values in S.KTB.

Driver access: Initialized, not referenced.

4.4.6 The Controller Request Block (KRB)

Figure 4–12 is a layout of the controller request block. One KRB exists for each controller. If a controller allows only a single operation on a single unit at a time, then the driver can allocate the controller request block and the status control block in continuous space. With such continuous allocation, all offsets commonly used by the driver are referenced by their S.xxx forms. The system will still use the offset S.KRB and the K.xxx forms for all references. Refer to Section 4.4.7 for the continuous SCB/KRB allocation.

Figure 4–12: Controller Request Block



1 If contiguous allocation of KRB and SCB is used (that is, if S2.CON is set), this field overlaps the I/O request queue.
2 This field is for multiprocessor support (M$$PRO is defined).
3 This area is for 11/70 extended memory support (M$$EXT is defined).
The area extends in a negative direction from the start of the UCB Table.
4 If KS.UCB is set, then this table appears (allows overlapped function interrupts).
5 The S.xxx forms of these offsets are valid only for devices that perform a single operation on a controller at a time. For such devices, S2.CON is set and the SCB and KRB are allocated in a contiguous area.
See figure 4–14 for the contiguous SCB/KRB structure.

ZK-264-81

The fields in the KRB are described as follows[1] :

K.PRM (device-dependent storage)

LOAD does not relocate any addresses in this area.

Driver access: Initialized, read/write.

K.PRI (device priority)

Contains the priority at which the device interrupts. Use symbolic values (for example, PR4) to initialize this field in the driver data source code. These symbolic values are defined by issuing the HWDDF$ macro (refer to the sample data base in Chapter 8 and to the listing of the HWDDF$ macro).

Driver access: Initialized, read-only.

K.VCT (interrupt vector divided by 4)

Interrupt vector address divided by 4. Because you can use the CON task to change the vector value, you need not be overly concerned with initializing K.VCT to the correct value. If K.VCT equals zero, then neither LOAD nor UNLOAD takes any vector action. In particular, LOAD does not create any interrupt control block linkage for this KRB.

Driver access: Initialized, not referenced.

K.CON (controller number times 2)

Controller number multiplied by 2. Drivers that support more than one controller use this field. A driver may use K.CON to index into a controller table created in the driver data base source code and maintained internally by the driver itself. By indexing the controller table, the driver can service the correct controller when a device interrupts.

Because this number is an index into the table of addresses in the CTB, its maximum value is limited by the value of L.NUM in that CTB.

Driver access: Initialized, read-only.

K.IOC (0)

This is an I/O count used by the system to keep track of how busy the controller is. The value is related to the number of outstanding requests queued for this controller. This is a weighted number to be used only by the system to judge the relative activity of one controller with respect to another.

Driver access: Initialized, not referenced.

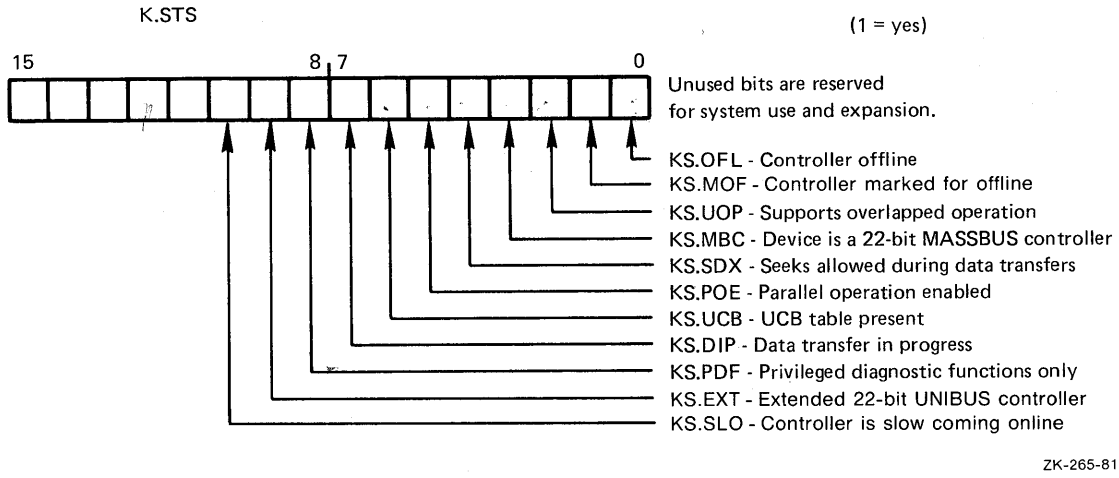
K.STS (controller-specific status)

This word is used as a status word that concerns the controller. Figure 4–13 shows the layout of the controller status word.

Driver access: Initialized, not referenced.

[1] Some fields require that you initialize a value in the data base source code. The descriptions indicate the required values in parentheses, following the symbolic offset.

Figure 4–13: Controller Status Word



All undefined bits are reserved for use by DIGITAL. Table 4–9 describes the currently defined bits for this word.

Table 4–9: Controller Status Word Bit Settings

KS.OFL =	1	The Executive reconfiguration routines set this bit to place the controller off-line and clear the bit to place the controller on-line. The bit is used in conjunction with KS.PDF to denote transition states. If a request is made to assign a unit to the controller and this KS.OFL is set (and no other on-line controller is found), the request terminates with the IE.OFL error and a return is made to the driver I/O initiation entry point to get a new packet. The driver data code should initialize this bit to 1.
KS.MOF =	2	If this bit is set, the unit/controller is in the process of becoming off-line.
KS.UOP =	4	This bit indicates whether the controller supports unit operation in parallel and requires synchronization. If this bit is set, each unit attached to the controller is capable of operating independently. Therefore, the KRB contains a UCB table holding the UCB addresses of each independent unit.

Table 4–9 (Cont.): Controller Status Word Bit Settings

KS.MBC	=	10	If this bit is set, the device is a 22-bit MASSBUS controller and does not use UNIBUS mapping registers (UMRs) but has 2 extra registers to describe a 22-bit address. If these registers exist, the offset to the first of them (RHBAE) is in the cell KE.RHB. These registers can be found by using the contents of KE.RHB in conjunction with the contents of S.RCNT. The Executive on-line reconfiguration code calls the common interrupt controller status change routine (in the module DVINT) which dynamically sets or clears this bit during controller processing.
KS.SDX	=	20	If this bit is set, the controller allows seek operations to be initiated while a data transfer is in progress. (Some types of disks, such as the RK06 and RK07, support overlapped seek operations but do not allow a seek to be initiated if a data transfer is in progress.) The Executive routines Request Controller for Control Function ($RQCNC) and Request Controller for Data Transfer ($RQCND) examine this bit to distinguish between the two types of controllers that support overlapped seeks.
KS.POE	=	40	If this bit is set, the driver may initiate an I/O operation on the controller in parallel with other I/O operations. A driver that supports overlapped seek operations checks this bit to decide whether it should attempt to perform an I/O operation as a seek phase and then a data transfer phase (that is, overlapped) or as an implied seek (that is, nonoverlapped). If this bit is set, the driver can then attempt the overlapped operation. An overlapped driver must check this bit once only for each I/O operation. Because this bit can be reset by system commands at any time, the driver must not rely on the bit value to decide whether, upon being interrupted, the driver was attempting a seek operation. The driver must use the S2.SIP bit to hold its internal state.
KS.UCB	=	100	This bit indicates the presence of the table of unit control block addresses associated with the KRB. If this bit is set, K.OFF gives the offset from the beginning of the KRB to the start of the UCB table. Devices that support unit operation in parallel (for example, overlapped seeks) require a mechanism for finding the UCB of the unit generating an interrupt. Therefore, if KS.UOP is set, a UCB table must exist. If KS.UOP is not set, however, a UCB table may still exist because some devices (for example, terminal multiplexers) support full unit operation in parallel but do not require synchronization. Therefore, KS.UCB may be used to determine whether the UCB table exists, regardless of whether KS.UOP is set.
KS.DIP	=	200	If this bit is set, a data transfer is in progress. A driver that supports overlapped seek operation sets or clears this bit to indicate to itself and to the Executive common interrupt module DVINT whether, after an interrupt, a data transfer is in progress. The driver must set or clear this bit. The use of this bit eliminates the need for the software to access the device registers to determine what type of operation was in progress.

Table 4–9 (Cont.): Controller Status Word Bit Settings

KS.PDF	=	400	This bit and one KS.OFL bit indicate the reconfiguration status of the controller. The Executive reconfiguration software accesses both bits to describe the off-line, on-line, and transition status of the controller.
KS.EXT	=	1000	If this bit is set, the device is a 22-bit UNIBUS controller and does not use UNIBUS mapping registers but has 2 extra registers to describe a 22-bit address. If these registers exist, the offset to the first of them, the bus address extension (BAE) is in the cell KE.RHB. These registers can be found by using the contents of KE.RHG in conjunction with the contents of S.RCNT.
KS.SLO	=	2000	If this bit is set, the controller requires the use of the extended time out feature of the reconfiguration subroutine. If this bit is not set, a controller will change to online or offline immediately.

K.CSR (controller status register address)

Contains the address of the Control and Status Register (CSR) for the device controller. Since you can use the CON task to change the CSR value, you need not be overly concerned with initializing K.CSR to the correct value. The driver uses K.CSR to initiate I/O operations and to access, by indexing, other registers that are related to the device and are located in the I/O page. This address need not be the CSR; it need only be a member of the device's register set. The Executive reconfiguration software probes K.CSR to bring a controller online. (If probing K.CSR yields a nonexistent memory trap, the controller will not be brought online.)

Driver access: Initialized, read-only.

Note

This word is guaranteed to be offset zero for the KRB. This assignment means that an RSX-11M-PLUS driver can access the CSR by the reference @S.KRB and need not use a separate register.

K.OFF (offset in bytes (from K.CSR) to start of UCB table)

This word contains the offset to the beginning of the unit control block table. When added to the starting address of the KRB, it yields the UCB table address. The UNIBUS mapping register work area extends in a negative direction from the start of the UCB table.

The status bit KS.UCB may be used to determine whether the UCB table exists. A UCB table may exist if KS.UOP is not set, since some devices (for example, terminal multiplexers) support full unit operation in parallel with no synchronization required. If KS.UOP is set, a UCB table must appear (and KS.UCB will also be set).

Driver access: Initialized, referenced by interrupt dispatch code.

K.HPU (highest physical unit number)

This byte contains the value of the highest physical unit number used on this controller.

Driver access: Initialized.

K.OWN (0)

This word has four slightly different uses, depending on the particular device.

- For controllers which always have only a single unit connected to them (for example, the line printer), K.OWN/S.OWN always points to the UCB of that unit. You can use the **suc** argument in the GTPKT$ macro to statically initialize this cell in the data base.
- For controllers that may have multiple units attached but do not support unit operation in parallel (for example, the RK05), K.OWN/S.OWN is set with the currently active unit by code generated with the GTPKT$ macro **suc** argument set to blank.
- For controllers that support unit operation in parallel and require synchronization (KS.UOP is set), this is a busy/nonbusy interlock for the controller. If the controller is busy for a data transfer, this word contains the UCB address of the currently active unit. This is true for RH disks such as the RP06. This word is set and cleared by the Request Controller for Control Access ($RQCNC), Request Controller for Data Access ($RQCND), and Release Controller ($RLCN) routines.
- For multiple SCB controllers that support unit operation in parallel but do not require synchronization, K.OWN must be set dynamically by the driver code. For this case you must specify the **suc** argument in the GTPKT$ macro.

Driver access: Initialized, referenced for actual unit.

K.CRQ (first word equals 0; second word points to first)

Two words that form the controller wait queue. Fork blocks are queued here for driver processes that have requested controller access. Driver processes that request access for control functions are queued on the front of the list, and those that request access for data transfer are queued on the end of the list.

Driver access: Initialized, not referenced.

K.URM (controller UNIBUS run mask)

This word appears only in a multiprocessor system (that is, one in which M$$PRO is defined).

It contains a UNIBUS run mask that defines the UNIBUS run to which the controller is attached. When controller assignment is made, the cell is moved into S.URM for the fork block there. This word should not be zero.

Driver access: Initialized, not referenced.

Table of UCB addresses (offset from K.CSR by K.OFF bytes)

This table contains the unit control block address for the units on this controller. Physical unit zero is in the first word, unit one is in the second word, and unit n is in word n +1. The table has a length of (K.HPU +1) words. A value of zero in this table indicates a physical unit number for which no actual physical unit exists. The table is terminated by a −1.

Driver access: Initialized, referenced by interrupt dispatch code.

Note

This table exists only for those devices that have KS.UCB set.

KE.RHB (reserve appropriate amount of storage)

The UNIBUS mapping register work area extends in a negative direction from the start of the unit control block table. This work area always appears if the device is an NPR device. For devices with either KS.MBC or KS.EXT set, the first word is used as the BAE offset for the controller. This word value is the offset that, when added to the CSR address contained in K.CSR, yields the address of the BAE register on the controller. If both KS.MBC and KS.EXT are clear, the device controller uses UMRs.

Use the mapping assignment block only if the device requires UNIBUS mapping registers (such as an RH11 or on a PDP-11/44 CPU).

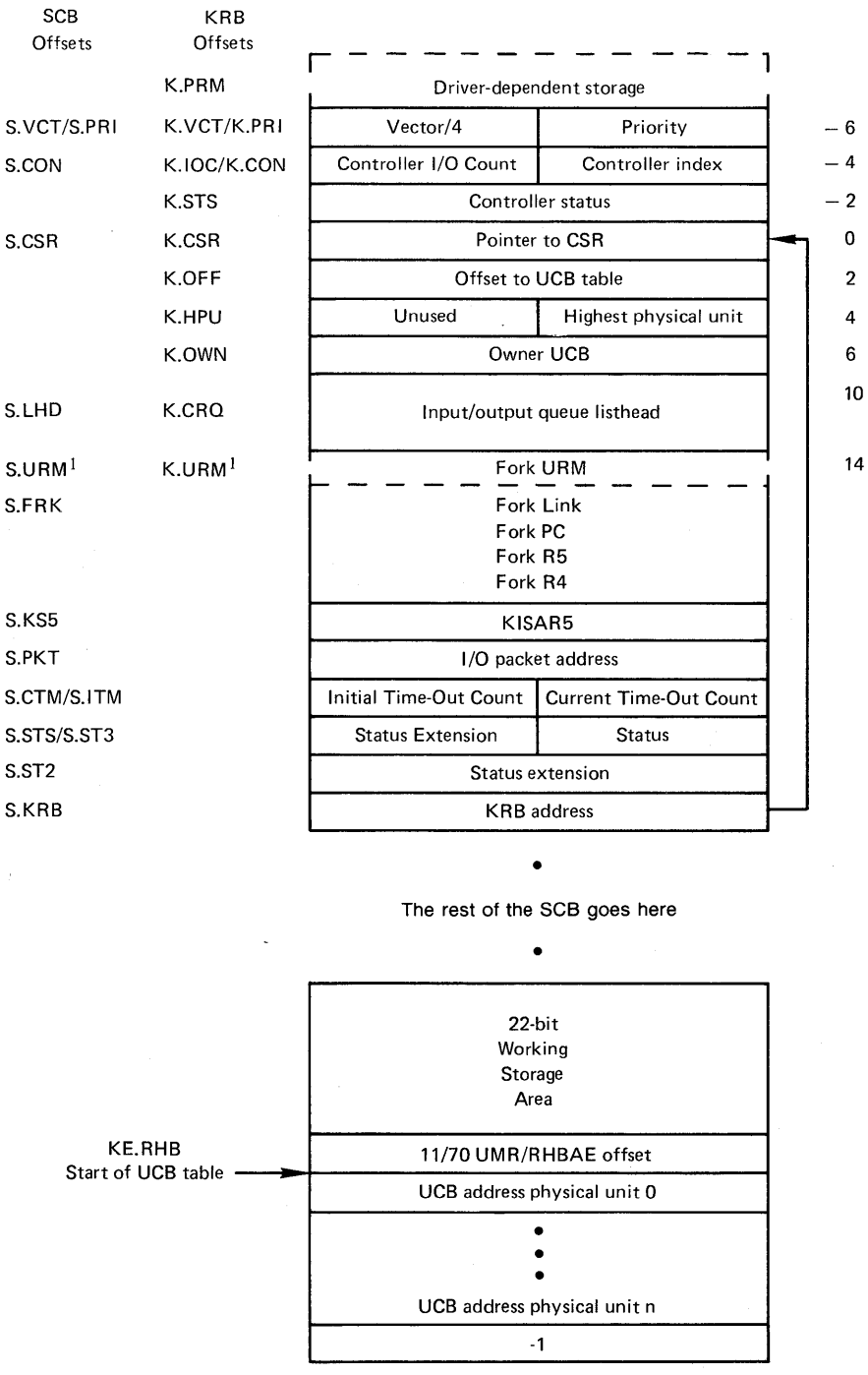
4.4.7 Continuous Allocation of the SCB and KRB

In a configuration where a controller and the Executive support only a single operation on a unit at one time, the driver can allocate space for the KRB and the SCB in a continuous area. Some fields of the KRB overlap those in the SCB. Although the KRB and SCB in this arrangement are contiguous, the system still considers the I/O data structure to contain a KRB. The system will still use the S.KRB offset and the K.xxx forms for all references. The driver can reference the fields by the S.xxx form of the symbolic offset definitions. In such a case, although the physical offsets may differ between RSX-11M and RSX-11M-PLUS systems, correct referencing of many locations on both systems is eased. Figure 4–14 shows the physical layout of the continuous KRB and SCB allocation.

4.4.8 Controller Table (CTB)

Figure 4–15 is a layout of the controller table. You ensure that the CTB is linked into the system list of controller tables by placing the CTB macro immediately before the allocation of the L.LNK word. The CTB macro generates a global symbol that links the user-written CTB into the system list.

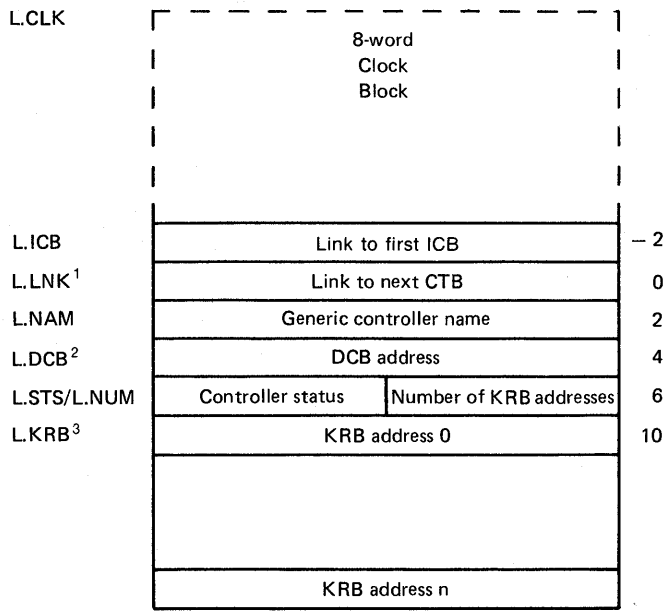
Figure 4–14: Continuous KRB/SCB Allocation



[1]This field is for multiprocessor support (M$$PRO is defined).

ZK-266-81

Figure 4–15: Controller Table



[1] The head of the list of controller tables is $CTLST in SYSCM.
[2] If LS.CIN is set, this cell points to the common interrupt address table rather than to the DCB.
[3] See Table 4-10 for label XXCTB.

ZK-267-81

The fields in the CTB are described as follows[1] :

L.CLK

This is the clock queue entry for those devices that need a single clock block per generic controller type. It only appears if LS.CLK is set.

Driver access: Initialized.

L.ICB (reserve one word of storage)

This word points to the first interrupt control block for the type of controller being supported. It is a link and not an address. In any system the ICBs must be in an executable pool area. In an I and D space multiprocessor system, they must be distinct for each processor, since each processor has its own local executable pool mapped by KISARO. Since the linkage

[1] Some fields require that you initialize a value in the data base source code. The descriptions indicate the required values in parentheses, following the symbolic offset.

must enter and leave in some way other than through the usual Executive kernel mapping, the upper four bits encode a processor number that may be used to enter $K6TAB, and the lower 12 bits form an address that has been shifted right once. On a multiprocessor system other than an I and D space system, the upper four bits are considered part of the address, which has also been shifted right once.

Driver access: Not initialized, not referenced.

L.LNK (0 or link to next CTB in list)

All of the controller tables in the system are linked together so they can be found, and they are threaded through this first word. A zero link terminates this list. A CTB must exist for every physical controller type in the system.

Driver access: Not initialized, not referenced.

L.NAM (two-character ASCII device name)

This two-character ASCII string is the controller mnemonic used to find this controller table from among all the others in the system. For the RH11/70 controller, it is RH instead of DB, DS, DR, or MM.

L.NAM must be unique throughout the system, unlike D.NAM in the device control block.

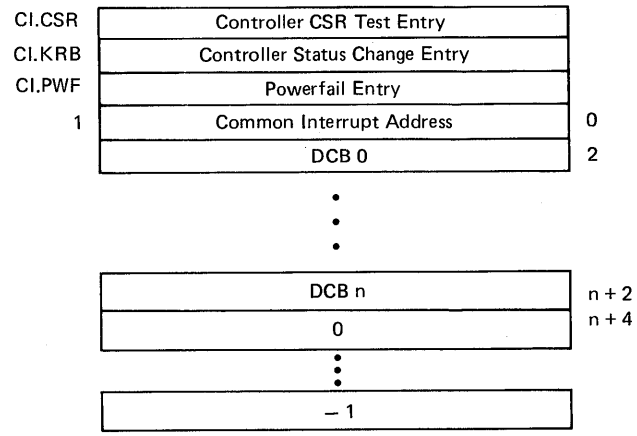
Driver access: Initialized, read-only.

L.DCB (DCB address or address of common interrupt table)

The DCB pointer is used to reach the device control block and, thereby, the unit control block and driver dispatch table for a driver. If LS.CIN is set, L.DCB is a pointer to a block that holds the common interrupt address (the address of the interrupt dispatch routine in the Executive), and the DCB addresses (the addresses of the DCBs for the devices that this controller interfaces). This block is called the common interrupt table and is shown in Figure 4–16.

Driver access: Initialized, not referenced.

Figure 4–16: Common Interrupt Table and Table of DCB Addresses



[1] If LS.CIN is set, L.DCB in CTB points to this structure instead of to the DCB.

ZK-268-81

The powerfail entry at offset CI.PWF and the controller status change entry at offset CI.KRB are addresses of routines built into the Executive and are used instead of the entries in a particular driver dispatch table. This allows devices that have no DCB (for example, the interprocessor interrupt and sanity timer) to nevertheless participate in reconfiguration.

At offset CI.KRB is the address of a routine built into the Executive for multidriver controllers such as the RH type. This routine should set or clear the KS.MBC bit to indicate whether the device is connected to an RH11 or an RH70. The driver checks the KS.MBC bit to determine which addressing format to use. If the value at CI.CSR is zero, the Executive on-line routines check the existence of a device attached to this controller by probing the address at K.CSR. If the value is nonzero, it is the address of a routine built into the Executive to check device presence. Instead of probing the address at K.CSR, the Executive on-line code calls this routine, which returns either with the C bit clear if the device is present or with the C bit set if the device is not present.

The common interrupt table may have only the common interrupt address in those cases in which a DCB does not exist (for example, the IIST). If LS.MDC is clear, then only one DCB address exists. (The zero termination is still necessary.) If LS.MDC is set, then more than one DCB address is possible; therefore, space should be left for all possible DCB addresses (for LOAD) and the table terminated by a zero, followed by a −1. Empty entries in this case are indicated by a zero word. LOAD will then enter the DCB addresses into the table when it loads data structures for drivers.

L.NUM (number of KRB addresses)

Used by programs that scan the controller tables to compute the number of KRB addresses. This value is never zero, since without controller request blocks there should be no controller table.

The maximum value for L.NUM depends on the type of device and on whether the driver is loadable. For common interrupt devices, resident drivers, and drivers loaded by MCR LOAD, the value must be less than 17_{10}. For drivers loaded by VMR LOAD, the value must be less than 17_{10} if the data base is loadable and less than 129 if the data base is resident.

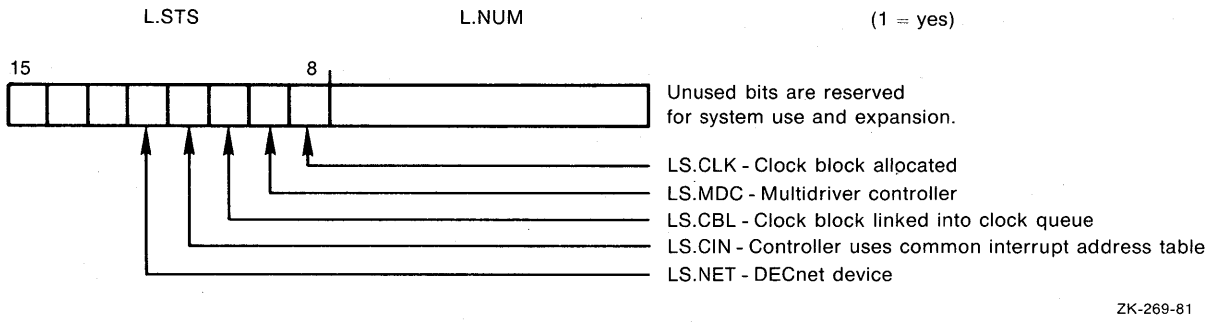
Driver access: Initialized, read-only.

L.STS (generic controller status)

The controller table status bits give information about the class of controllers. Figure 4–17 shows the layout of this byte.

Driver access: Initialized, read-only.

Figure 4–17: Controller Table Status Byte



These bits are described as follows:

LS.CLK	=	1	If this bit is set, the controller table has an 8-word clock block.
LS.MDC	=	2	If this bit is set, multiple drivers service units attached to the associated controller.
LS.CBL	=	4	If this bit is set, the clock block is linked into the clock queue.
LS.CIN	=	10	If this bit is set, the driver is associated with a common interrupt controller and must have exactly one interrupt vector. The driver is therefore called at the D.VPWF entry point only for unit power failure. The Executive uses the CI.PWF entry point in the common interrupt entry table for controller power failure recovery. In addition, the cell L.DCB does not point to the device control block but rather to the common interrupt entry table in the Executive.

L.KRB (KRB addresses of controllers)

A list of the controller request block addresses ordered by their respective system-wide controller numbers. This table is indexed by the controller index retrieved from the PS word immediately after an interrupt. The table is of length (L.NUM(R?)) words. While the interrupt routines will not have to scan the list in a linear fashion, the only way to find all

the controller request blocks in the system involves a linear scan of all the controller tables. The CTB is static.

The address of the start of the KRB address list in the CTB is the global symbol $xxCTB in the driver dispatch table, where xx are the characters comprising the controller mnemonic. Because LOAD supplies this address in the DDT when it loads the driver, a loadable driver should not specify this address in the DDT.

Note

A KRB address of zero indicates a controller that was specified during system generation, with no attached units. No controller request block for such a controller is generated.

The proper action for drivers to access their list of KRB addresses is to retrieve the address of the start of the KRB list in the CTB from the cell in the driver dispatch table set up by LOAD (both VMR and MCR).

Driver access: Initialized once for the controller, not referenced.

4.5 Driver Code Details

This section describes the specific requirements for driver code. The driver code must contain a driver dispatch table which allows the Executive to call the driver to perform discrete system functions. If the driver needs to access either system structures such as the partition and task control blocks or structures within its own data base, it should use the system-wide symbolic offsets rather than the real offsets. Because the driver is built with the Executive library EXELIB.OLB, the symbolic offsets are automatically defined for the driver code. If you want to see the definitions of the symbols in your driver listing, place in your driver source code the related macro name in a .MCALL directive and invoke the macro. (For your convenience, the source code of the macro calls that define the symbols of structures is in Appendix A.) The detailed descriptions of the driver data base structures are in Section 4.4.

4.5.1 Driver Dispatch Table Format

The driver dispatch table associates the entry points that the Executive expects to find in a device driver and the actual locations of the routines in the driver code. The DDT also provides a link from the driver code to the driver data base. Figure 4–18 shows the format of the DDT. Section 4.3.1 describes the DDT$ macro call, which automatically generates the DDT.

All device drivers require a driver dispatch table somewhere in the first 4K words of the driver code. Conventionally, the table is located at the beginning of the code.

Note

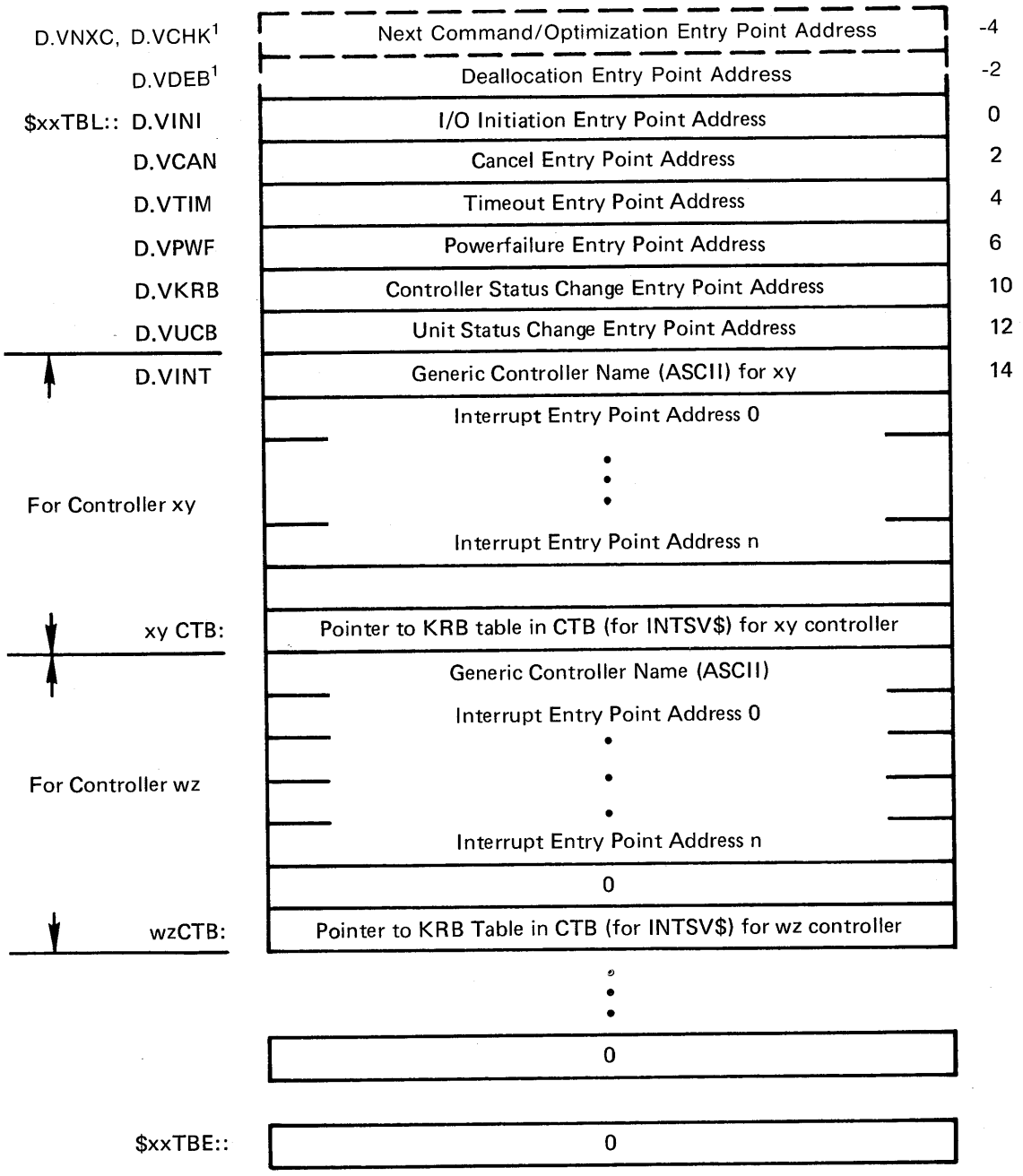
If the length of a driver must exceed 4K words (20000 octal bytes), then your driver must set up the mapping for the second 4K words whenever it is entered; and, of course, all entry points must be in the first 4K words of the driver.

The driver must define some labels that the Executive routines and the INTSV$ macro call use to access the DDT. Table 4–11 lists these labels, which are automatically generated by the DDT$ macro call. Because these labels do not appear in the DDT itself, their format is fixed and they must be specified in the format shown.

Table 4–10: Labels Required for the Driver Dispatch Table

Required Format	Meaning
$xxTBL::	Defines the start of the DDT. You specify this label in the D.DSP word of the DCB of resident drivers to link the DCB to the DDT. For loadable drivers, the LOAD routines use this label to fill in D.DSP.
xxCTB:	Defines the pointer to the table of KRB addresses in the CTB of the controller for device xx. Because a driver can support different types of controllers, there may be more than one of this form of label. (The DDT$ macro supports only one controller type.)
$xxTBE::	Defines the end of the DDT for Executive LOAD and UNLOAD routines that scan the DDT.

Figure 4–18: Driver Dispatch Table Format



1. These are optional advance driver features

ZK-270-81

Labels defining the addresses of the entry points in your driver appear at offsets D.VINI through D.VUCB in the DDT. As a standard procedure, you supply the labels described in Table 4–11 at the entry points in the driver code. The formats of the standard labels that appear in the DDT are not fixed. Because the Executive expects to find the entry point addresses at fixed offsets from the start of the DDT, and because the labels themselves appear in the DDT, you can change their format if you construct the DDT without using the DDT$ macro call. However, other labels that are required in the driver code but do not appear in the DDT have a certain, fixed format that you must not change. For reference, these fixed format labels are as follows:

```
$xxTBL::
xxCTB:
$xxTBE::
$xxLOA::
$xxUNL::
```

These fixed-format labels are described elsewhere in this chapter. The DDT$ macro uses the standard labels but allows you to alter the format of some of them.

The name of the controller type that the driver supports (the same name as in the CTB) is located at offset D.VINT in the DDT. If the driver has no controller (such as the virtual terminal driver VTDRV), this word is zero. The structure allows the driver to support multiple controller types. (The terminal driver supports different controller types.) Although the DDT$ macro supports only one controller type, there is no restriction on the number of controller types that a driver can support.

A block of interrupt entry addresses follows each controller name. The first interrupt address block, each word of which defines an address to be included in a vector for the driver, begins at location D.VINT+2. A zero terminates the block and indicates that there are no more interrupt entry points for the controller. There is no restriction on the number of vectors each controller may have. For a single interrupt device, location D.VINT+2 (interrupt entry address 0) is the interrupt address.

Table 4–11: Standard Labels for Driver Entry Points

Label[1]	Entry Point
xxINI:	I/O initiation
xxCAN:	Cancel I/O
xxCHK:	Block check and conversion
xxOUT:	Device timeout
xxPWF:	Power failure
xxKRB:	Controller status change
xxUCB:	Unit status change
$xxINT::	Interrupt entry point

[1]The characters xx are the two-character mnemonic.

The Executive reconfiguration software follows certain rules when it accesses the interrupt address block to calculate the vectors for a controller. To calculate the first vector address, a

reconfiguration routine accesses the cell K.VCT (or S.VCT) in the controller request block. If K.VCT is not equal to zero, it is multiplied by 4. The routine examines the value of entry point 0 in the driver dispatch table. This value is loaded into the vector address now pointed to by K.VCT. The next entry point in the DDT is examined, and if it is zero, there are no more vectors or interrupt entry points for the controller. If it is even, the entry point is loaded into the next vector address (the previous one plus 4).

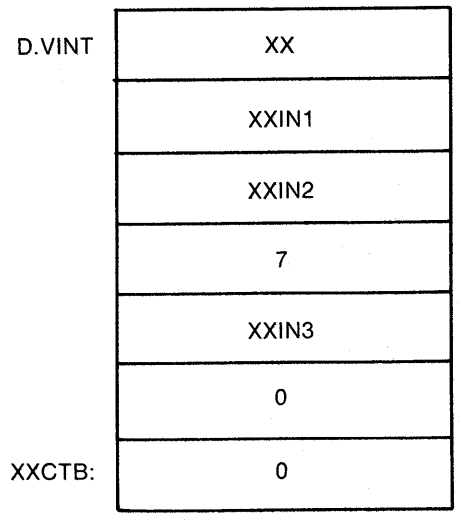
If an entry point value in the block is odd (bit zero is set), bit zero is cleared and the resulting number is an offset to the next vector address. To compute the next vector address, the offset is added to the last vector address. The next interrupt entry point is examined. If it is even, then its value is loaded into the last vector address computed. If it is odd, the result is an offset that is added to the vector address just computed, and the next entry point is examined. The computation of vector addresses terminates when the next entry point is zero.

The entries shown in Figure 4–19 can be used to calculate the interrupt vector addresses when K.VCT equals 300. The vectors at 300 and 304 are loaded with addresses xxIN1 and xxIN2. The odd value 7 yields the offset 6 that is added to the last vector computed to attain 312. The address xxin3 in the next interrupt entry point examined is loaded in the vector at 312. A zero word in the block shows there are no more vectors or interrupt entry points.

Following the interrupt entry address block for a controller type is a pointer to the KRB table in the CTB. Its label is in the form xxCTB, which is used by the INTSV$ macro. This pointer connects the driver code to the driver data base and is the last entry in a block for a specific controller.

A zero terminates the driver dispatch table. The global label in the form $xxTBE marks the terminating word in the DDT.

Figure 4–19: Sample Interrupt Address Block in the DDT



4.5.2 I/O Initiation Entry Point

The offset D.VINI in the driver dispatch table contains the address of this entry point. A driver is called at this entry point at priority 0 from the Executive routine $DRQRQ in the module DRQIO. A driver should call the Executive $GTPKT routine to get an I/O packet to process. This action dequeues an I/O request. The entry conditions when the Executive enters the driver are as follows:

R5 = Address of the UCB of the unit for which the Executive has queued an I/O packet

This entry condition holds true unless the driver wants to delay the queuing operation. Therefore, if the queue-to-driver bit UC.QUE in the unit status block offset U.CTL is set, the following are the register conventions:

R5 = UCB address of unit for which a packet has been created

R4 = SCB address of the related unit

R1 = Address of the I/O packet

You can find more information and coding requirements for the queue-to-driver operation in the description of the UC.QUE bit in Section 4.4.4 and an example of its use in Chapter 8.

The GTPKT$ macro call automatically generates the call to the $GTPKT routine and the code to process the return from $GTPKT. Upon return from $GTPKT, the C bit indicates whether there is a packet to process.

C = 1 If the C bit is set, the Executive found the controller busy, could not dequeue a request, or had to call $FORK to have the driver run on the correct processor.

C = 0 If the C bit is clear, the Executive successfully dequeued a packet for the driver and placed it in the device's input/output queue.

If a request was successfully dequeued, the following are the contents of the registers:

R5 = Address of unit control block

R4 = Address of status control block

R3 = Controller index

R2 = Physical unit number of device to process

R1 = Address of the I/O packet

If the C bit is set, the driver returns control to the caller (a RETURN instruction should be executed). If the C bit is clear, the generated code loads the location at offset K.OWN/S.OWN in the continuous KRB/SCB with the UCB address of the unit to process. The driver may then process the request and activate the device. All registers are available to the driver. The driver executes a RETURN instruction to transfer control to the system.

On a multiprocessor system, before returning a packet to the driver, $GTPKT calls the conditional fork routine $CFORK to ensure that the driver executes on the correct processor. If the current processor is the correct processor, $CFORK returns to $GTPKT, and $GTPKT dequeues an I/O packet, queues it to the driver, and returns to the driver with the C bit clear. Should the current processor not be the correct processor, $CFORK will call $FORK, which returns to the driver

with the C bit set. This action causes the driver to dismiss itself. Eventually the fork processor restarts the driver executing on the correct processor.

4.5.3 Cancel Entry Point

The offset D.VCAN in the driver dispatch table contains the address of this entry point. The Executive routine $IOKIL in the IOSUB module calls the driver at this entry point at device priority. When the Executive enters the driver, the following register conventions pertain:

R5 = UCB address

R4 = SCB address

R3 = Controller index (undefined if S.KRB equals zero)

R1 = Address of TCB of current task

R0 = Address of active I/O packet

The use of this entry point is explained in Section 2.2.2. All registers are available to the driver. The driver returns control to the Executive by executing a RETURN instruction.

4.5.4 Device Timeout Entry Point

The offset D.VTIM in the driver dispatch table contains the address of this entry point. Routines in the Executive module TDSCH call the driver at this entry point at device priority. When the Executive enters the driver, the entry conditions are as follows:

R5 = UCB address

R4 = SCB address

R3 = Controller index (undefined if S.KRB equals zero)

R2 = Address of device CSR

R0 = I/O status code IE.DNR (Device Not Ready)

The use of this entry point is explained in Section 2.2.3. All registers are available to the driver. The driver returns control to the Executive by executing a RETURN instruction.

4.5.5 Next Command Entry Point

The offset D.VNXC in the driver dispatch table is only applicable to the terminal driver. The offset D.VNXC contains the entry point address of a routine within the terminal driver which is called from the routine $SNCMD in the Executive module DRSUB. This entry point is entered when a task exists whose TI: is set to serial mode. The driver then passes the next CLI command to the MCR dispatcher. When the Executive enters the driver, the register entry conditions are as follows:

R0 = UCB address of the TI: of the exiting task

4.5.6 Queue Optimization Entry Point

The offset D.VCHK in the driver dispatch table contains the address of this entry point. The routine $DRQRQ in the Executive's module DRSUB calls the driver at this entry point at priority zero. When the Executive enters the driver, the register entry conditions are as follows:

R5 = UCB address

R1 = I/O packet address

If the I/O operation is a data transfer function, the I/O packet contains the starting LBN for the I/O request. The routine at this entry point must verify the request is a data transfer function, and if it is, the routine must replace the starting LBN with the starting cylinder, track, and sector number to perform queue optimization. See the routine DBCHK in the module DBDRV for an example of a driver that supports queue optimization.

4.5.7 Deallocation Entry Point

The offset D.VDEB in the driver dispatch table contains the address of this entry point. This entry point is called at priority zero from the routine $FINBF in the Executive module SYSXT after a buffered I/O request completes. The driver is expected to deallocate its buffers at this entry point. When the entry point is called, the registers are set up as follows:

R0 = address of the first buffer

All registers are available to the driver. The driver returns control to the Executive by executing a RETURN instruction.

4.5.8 Power Failure Entry Point

The offset D.VPWF in the driver dispatch table contains the address of this entry point. The routines in the Executive module POWER call the driver at this entry point at priority 0 for both unit and controller power failures. The Executive first calls the driver for controller power failure with the C bit set. The driver is called in this fashion once for each controller. The register entry conditions are as follows:

C bit set (controller power failure)

R3 = CTB address

R2 = KRB address

The driver may use all registers.

After the Executive has called the driver for all related controllers, it calls the driver once for each unit power failure at priority 0 with the C bit clear. The register entry conditions are as follows:

C bit clear (unit power failure)

R5 = UCB address

R4 = SCB address

R3 = Controller index

For both controller and unit power failures, the driver returns control to the calling routine by executing a RETURN instruction.

If the driver supports a common interrupt device (that is, the LS.CIN bit in the CTB is set), the driver is called at this entry point for unit power failures only. For controller power failures, the Executive calls the entry point at CI.PWF in the common interrupt entry table. See the description of the offset L.DCB in Section 4.2.5.

4.5.9 Controller Status Change Entry Point

The offset D.VKRB in the driver dispatch table contains the address of this entry point. The Executive routine $KRBSC in the OLRSR module calls the driver at this entry point at priority 0 to put a controller on-line or to take a controller off-line.

Note

If the controller is a common interrupt controller (LS.CIN is set), the Executive does not call the driver at this address (if any) specified in the DDT but at the address in the common interrupt table labelled CI.KRB. See Section 4.2.5.

The C bit indicates whether the request is for off-line or on-line. The register conditions upon entry to the driver are as follows:

R3 = CTB address for the controller

R2 = KRB address of controller changing status

0(SP) = Return address for completion

2(SP) = Return address for caller of the Executive routine

The C bit is set to indicate the requested status change as follows:

C = 1 On-line to off-line transition

C = 0 Off-line to on-line transition

The status change byte $SCERR is preset as follows:

$SCERR = 1

The driver indicates the return status in the $SCERR byte as follows:

$SCERR < 0 Operation is not successful and a negative value in $SCERR is the I/O error code. Thus, a negative value rejects the status change requested by the C bit.

$SCERR = 1 Operation is successful. The driver accepts the status change requested. This is the default condition.

All registers are available to the driver. The Executive does not change the status of the controller until and unless the driver shows successful completion of the on-line or off-line request.

The driver must return immediately by one of the following methods:

- The driver can indicate the return status immediately and can return to the first address on the stack in the normal fashion. If the driver accepts the status change, it merely executes a RETURN instruction. (The status change byte $SCERR has been preset with 1.) If the driver rejects the status change, it loads the relevant I/O error code into $SCERR and executes a RETURN instruction. (The I/O error code symbols are listed in an appendix of the *RSX-11M-PLUS and Micro/RSX I/O Operations Reference Manual.*)

- The driver need not indicate the status immediately but removes the first address from the stack, saves it, and returns immediately to the second address. The driver then has 60 seconds to perform its processing, to indicate the return status, and to return to the first address. The driver can use the offset S.CTM in the status control block to time out some operation (such as a protocol rundown) and then accept or reject the operation by using $SCERR.

If the driver does not return to the first address on the stack, the system can be considered to be in an indeterminate state and possibly corrupted. The driver must return immediately because status changes should not stall the system. The 60-second delay allows a driver time to overcome conditions over which it has little control (such as network connections). System disk and terminal drivers must indicate return status immediately. However, the terminal driver (TTDRV) rejects a controller on-line request for a DZ11 multiplexer if some of the status bits indicate that the device is not a DZ11 or that it is broken.

4.5.10 Unit Status Change Entry Point

The offset D.VUCB in the driver dispatch table contains the address of this entry point. The Executive routine $UCBSC in the OLRSR module calls the driver at this entry point at priority 0 to put a unit on-line or to take a unit off-line. This entry is called once for each unit whose status changes. The C bit indicates whether the request is for on-line or off-line. The following are the register conventions:

R5 = Address of UCB or unit changing status

R4 = Address of SCB of unit

R3 = Controller index (undefined if S.KRB equals zero)

0(SP) = Return address for driver completion

2(SP) = Return address for caller of the Executive routine

The C bit is set to indicate the requested status change as follows:

C = 1 On-line to off-line transition

C = 0 Off-line to on-line transition

The status change byte $SCERR is preset as follows:

$SCERR = 1

The driver indicates the return status in the $SCERR byte as follows:

$SCERR < 0 Operation is not successful and a negative value in $SCERR is the I/O error code. Thus, a negative value rejects the status change requested by the C bit.

$SCERR = 1 Operation is successful. The driver accepts the status change requested. This is the default condition.

All registers are available to the driver. The driver must return within 60 seconds. The Executive does not change the status of a unit until and unless the driver shows successful completion of the on-line or off-line request.

The driver must return immediately by one of the following methods:

1. The driver can indicate the return status immediately and can return to the first address on the stack in the normal fashion. If the driver accepts the status change, it merely executes a RETURN instruction. (The status change byte $SCERR has been preset with 1.) If the driver rejects the status change, it loads the relevant I/O error code into $SCERR and executes a RETURN instruction. (The I/O error code symbols are listed in an appendix of the *RSX-11M-PLUS and Micro/RSX I/O Operations Reference Manual.*)
2. The driver need not indicate the status immediately but removes the first address from the stack, saves it, and returns immediately to the second address. The driver then has 60 seconds to perform its processing, to indicate the return status, and to return to the first address. The driver can use the offset S.CTM in the status control block to time out some operation (such as a protocol rundown) and then accept or reject the operation by using $SCERR.

If the driver does not return to the first address on the stack, the system can be considered to be in an indeterminate state and possibly corrupted. The driver must return immediately because status changes should not stall the system. The 60-second delay allows a driver time to overcome conditions over which it has little control (such as network connections). System disk and terminal drivers must indicate return status immediately.

4.5.11 Interrupt Entry Point

Upon an interrupt, control is dispatched directly to the driver from an interrupt vector through an interrupt control block. A device may have more than one interrupt entry point. The entries in the DDT interrupt address block are used to initialize either the vectors or the interrupt control block with the addresses of the related interrupt entry points. (Refer to Section 4.5.1 for a discussion of the interrupt address block.) All drivers should observe the protocol for handling interrupts introduced in Section 1.3 and summarized in Section 4.1.

If the driver is loadable, it will be called from the interrupt dispatch coroutine $INTSI in the Executive. The register entry conditions when the driver gets control are as follows:

R4 = Controller index

Registers 4 and 5 are available to the driver. The driver runs at the priority set in the interrupt control block. To dismiss the interrupt, a driver executes a RETURN instruction.

If the driver is resident, it receives control directly from the interrupt vector. It runs at priority PR7 and the low-order four bits of the PSW have the controller number of the interrupting device. Because the low-order four bits are status bits and almost any instruction modifies them, the first operation that should be performed is to save the PSW. Then, the driver does its processing at priority PR7 (saving registers if necessary). After processing, it restores the registers (if necessary) and dismisses the interrupt by executing a Return to Interrupt (RTI) instruction.

However, your driver should use the INTSV$ macro call at an interrupt entry point. The INTSV$ macro resolves entry processing for both loadable and resident drivers. For loadable drivers, INTSV$ does not generate a call to $INTSV because LOAD establishes in the interrupt control block the call to the $INTSI coroutine. The $INTSI coroutine saves Registers 4 and 5, sets the priority to that in the interrupt control block, and forms the controller index from the

PSW, storing it in Register 4. (LOAD previously set the priority in the interrupt control block based on the value at offset K.PRI in the controller request block.)

For resident drivers, INTSV$ generates a call to the $INTSV coroutine, which sets the priority to that specified in the INTSV$ macro call, saves Registers 4 and 5, and forms the controller index from the PSW, storing it in Register 4.

For both loadable and resident drivers, INTSV$ generates code to load Register 5 with the UCB address of the interrupting unit. After the INTSV$ call in the driver code, the register entry conditions for both loadable and resident drivers are as follows:

R5 = UCB address of the interrupting unit

R4 = Controller index

The driver may then do the following operations:

1. Save extra registers if necessary
2. Do whatever processing is necessary
3. Become a fork process to access the data structures or to call Executive routines if necessary
4. Restore the explicitly saved extra registers
5. Execute a RETURN instruction to the coroutine, which dismisses the interrupt

In summary, the INTSV$ macro eliminates your having to consider the coding differences between a loadable and a resident driver in the interrupt service routine.

4.5.12 Volume Valid Processing

System-supplied drivers that service mountable devices (those that have the DV.MNT bit in the UCB U.CW1 word set) take advantage of special processing of volume valid for a device. For such devices the Executive directive processor DRQIO checks that either of the mounted status bits US.MNT or US.FOR in the UCB U.STS word is set. If a mounted status bit is not set, DRQIO requires that a device-specific bit called volume valid (US.VV) be set or else it rejects the directive. If a mounted status bit is set, DRQIO does not check the volume valid bit. (DRQIO assumes that the MOUNT command properly set the volume valid bit.)

To effectively service a mountable device on the system, a user-written driver should perform in one of two ways. First, it can take advantage of the volume valid capability in the same way that a system-supplied driver does. This processing involves calling the $VOLVD routine in the Executive module IOSUB and handling the spinning-up status bit (US.SPU) and the volume valid bit (US.VV) in the UCB status byte U.STS. (For details of this mechanism, refer to driver source code supplied on the system.) Second, a user-written driver can circumvent the volume valid processing with the following operations:

1. Enable the set characteristics function (IO.STC) for volume valid in the DCB legal function mask word.
2. Enable the same function in the DCB no-op function mask word.
3. Statically set the US.VV bit in the UCB in the driver data base source code.

The second method allows the device to be successfully mounted and associated with an ancillary control processor without your having to include code in the driver to handle US.VV.

Chapter 5

Incorporating a User-Supplied Driver Into RSX-11M-PLUS

This chapter describes how to incorporate a user-supplied driver into RSX-11M-PLUS and Micro/RSX systems. Most of the procedures in this chapter apply only to RSX-11M-PLUS, not to Micro/RSX. The procedure for incorporating a user-supplied driver into a Micro/RSX system is described in Section 5.4.3. The material in the chapter assumes that your driver source code adheres to the programming specifics in Chapter 4.

5.1 Guidelines for Incorporating a Driver

The procedures to incorporate a user-supplied driver into RSX-11M-PLUS depend on the type of driver you have. RSX-11M-PLUS supports the following kinds of drivers[1] :

- Loadable driver with a loadable data base
- Loadable driver with a resident data base
- Resident driver with a resident data base

If your driver is loadable with a loadable data base, you may perform a system generation to include your driver, or you may incorporate it directly into the system you are currently running. If you want to use a new version of a loadable driver with a loadable base and your driver is currently loaded, you must create a new system image file, load the new version of the driver into the file, and then bootstrap the new system. Refer to Section 5.1.1 if you want to incorporate your driver at system generation. Refer to Section 5.1.2 if you want to incorporate your driver after system generation.

If your driver is loadable with a resident data base, or is resident, you must perform a system generation because the resident driver and/or data base reside in the Executive and must be assembled and task-built as part of the Executive. Refer to Section 5.1.1 to incorporate your driver at system generation.

[1] Micro/RSX supports only loadable user-supplied drivers with loadable data bases. See Section 5.4.3.

Because loadable drivers and loadable data bases can be changed and reloaded without performing a system generation, loadable drivers with loadable data bases are easier to debug and maintain than resident drivers and/or resident data bases.

5.1.1 Incorporating a Driver at System Generation

If you want to build a loadable driver with a loadable or resident data base during system generation, proceed as follows:

1. Assemble and task-build your driver to eliminate any assembly or Task-Builder (TKB) errors.
2. Put the MACRO-11 source files containing your driver code and data base in the User File Directory (UFD) [11,10] on the target system disk. The driver source file should be named xxDRV.MAC and the data base source file should be named xxTAB.MAC, where xx is the two-character device mnemonic. Mnemonics for user-supplied devices should begin with the letters J or Q to avoid conflict with DIGITAL-supplied devices.
3. Perform a system generation and choose the full-functionality Executive. Answer the questions concerning user-supplied drivers printed during system generation. This procedure includes your driver data base in the Executive if it is resident and builds your driver task image. A loadable driver and data base are loaded into the system image file. Refer to Section 5.3 for a description of the system generation procedure.
4. Use CON from the Monitor Control Routine (MCR) to make your devices accessible. Refer to Section 5.2.5 for the CON command description.

If you want to build a resident driver, proceed as follows:

1. If your driver can run loadable with a loadable data base, first build and test it as loadable with a loadable data base.
2. Put the MACRO-11 source files containing your driver code and data base in the UFD [11,10] on your target system disk. The driver source file should be named xxDRV.MAC and the data base source file should be named xxTAB.MAC, where xx is the two-character device mnemonic. Mnemonics for user-supplied devices should begin with the letters J or Q to avoid conflict with DIGITAL-supplied devices.
3. Perform a system generation. If you want to include a resident driver, you must not choose the full-functionality Executive or Executive data space support. Answer the questions concerning user-supplied drivers printed during system generation. This procedure includes your driver and data base modules in the Executive. Refer to Section 5.3 for a description of the system generation procedure.
4. Use CON from the MCR to make your devices accessible. Refer to Section 5.2.5 for the CON command description.

5.1.2 Incorporating a Loadable Driver with a Loadable Data Base After System Generation

Incorporating a loadable driver with a loadable data base after system generation involves the following steps: (This procedure is for RSX-11M-PLUS systems only. To incorporate a loadable driver into a Micro/RSX system, see Section 5.4.3.)

1. Assemble and task-build your driver to eliminate any assembly or Task-Builder errors.
2. Put the MACRO-11 sources files containing your driver code and data base in UFD [11,10] on your target system disk. The driver source file should be named xxDRV.MAC and the data base source file should be named xxTAB.MAC, where xx is the two-character device mnemonic. Mnemonics for user-supplied devices should start with the letters J or Q to avoid conflict with DIGITAL-supplied devices.
3. Run the system generation procedure and follow the instructions in the "Adding a Device" (AD) section. (For information on invoking the system generation procedure [SYSGEN], refer to the *RSX-11M-PLUS System Generation and Installation Guide.*)

   These procedures require you to enter the two-character device mnemonic for your driver. Remember, this should be the same mnemonic used in the driver and data base source files names.
4. Use the MCR LOA command to link your driver data base into the system device tables and to load your driver data base and driver code. Refer to Section 5.2.4 for the LOA command descriptions.
5. Use CON from MCR to place the controllers and units on line. (CON can also alter vector assignments.) Refer to Section 5.2.5 for the CON command descriptions.

5.2 What the System Generation Procedure Does for You

The system generation procedure assembles your driver and data base, puts the resulting object modules in the Executive object library, and task-builds your driver. If your driver or its data base is resident, the driver or data base is included in the Executive. If your driver or its data base is loadable, the driver or data base is loaded into the system image file. You must then make the controllers and units accessible.

The commands that the system generation procedure uses to assemble your data base, insert your driver and data base modules in the library, and task-build your driver are the same commands that you may use to assemble, insert, and task-build your driver. The following subsections explain each of the procedures for incorporating your driver.

5.2.1 Assembling the Driver and Data Base

The following commands assemble your driver and its data base during the system generation procedure:

```
MAC>[11,24]xxDRV,[11,34]xxDRV/-SP=[1,1]EXEMC/ML,[11,10]RSXMC/PA:1,xxDRV
MAC>[11,24]xxTAB,[11,34]xxTAB/-SP=[1,1]EXEMC/ML,[11,10]RSXMC/PA:1,xxTAB
```

If your driver is resident, these commands are located in the file RSXASM.CMD. If your driver is loadable, these commands are located in the file xxDRVASM.CMD, where xx is the device mnemonic.

The commands to the assembler specify as input the Executive macro library EXEMC.MLB, the Executive assembly prefix file RSXMC.MAC, and either your driver code or driver data base source file (xxDRV.MAC or xxTAB.MAC). EXEMC.MLB contains the macro definitions of structures and symbolic offsets that your code may reference. (The source code for some of the macro definitions is given in Appendix A.) RSXMC.MAC contains symbols defined during system generation and definitions of some macros that your driver may invoke (such as DDT$, GTPKT$, and INTSV$). The assembler looks for the source file of your driver in UFD [11,10].

As output, the assembler creates object modules in UFD [11,24] and listing files in UFD [11,34]. The object modules xxDRV.OBJ and xxTAB.OBJ will later be put in the Executive library. You should retain the listing files xxDRV.LST and xxTAB.LST for documentation and maintenance purposes.

5.2.2 Inserting the Driver and Data Base Modules in the Library

After your driver and data base modules have been assembled, the driver and data base modules are added to the Executive object library. Commands to the Task-Builder (described in Section 5.2.3) require that the modules be in this library.

The following command during system generation adds both the driver and its data base to the same library:

```
LBR [1,24]RSX11M/RP=[11,24]xxDRV,xxTAB
```

The command to LBR adds the object modules of both your driver and its data base to the Executive object library RSX11M.OLB, which resides in UFD [1,24]. RSX11M.OLB is built from object modules assembled during system generation. The /RP switch ensures that any modules of the same name are replaced by the recently created modules. If this is not the first time you have performed this operation, LBR prints messages telling you that it replaced your modules in the library with the new versions.

5.2.3 Task-Building the Driver

After the modules have been added to the Executive object library, the system generation procedure task-builds your driver and data base. The commands for a resident driver are located in the file RSX11M.CMD. The commands for a resident data base are located in the file RSX11M.CMD on systems without Executive data space support and in the file DSP11M.CMD on systems with Executive data space support.

The commands for a loadable driver are located in xxDRVBLD.CMD, where xx is the device mnemonic. The following discussion explains each of the lines that are contained in the command file for a loadable driver.

1. When the system generation procedure builds your driver, a task-image file name and a symbol definition file name are specified as TKB output. The task image and symbol definition files are placed in the UFD corresponding to the system User Identification Code (UIC) that will be in effect when the LOA command is issued. The file names are both xxDRV, where xx is the device mnemonic. The Task-Builder produces the output files named xxDRV.TSK, xxDRV.MAP, and xxDRV.STB. For example, the input supplied to TKB to build the xx device would look like the following:

   ```
   [1,54]xxDRV/-HD/-MM,[1,34]xxDRV/-SP,[1,54]xxDRV=
   ```

2. No task header is included. The switch /-HD is used, as in the previous example. A driver is not really a task, but an extension of the Executive, and as such needs no task header.

3. The switch /-MM must be used in the command line.

4. A map file is produced and is useful for debugging. All driver map files are written to UFD [1,34]. The switch /-SP suppresses automatic spooling to the line printer.

5. The system generation procedure links your driver to the system symbol definition file that contains definitions of Executive global symbols. To illustrate, the following TKB input continues the example from item 1:

   ```
   [1,24]RSX11M/LB:xxDRV:xxTAB
   [1,54]RSX11M.STB/SS
   ```

   The first line above specifies the library file (/LB) in which the input driver object module and the object file for the loadable data base can be found. The object module specification for the driver always precedes the specification for the data base in the TKB command line.

   The second line in item 5, above, indicates that the symbol definition file RSX11M.STB is to be searched selectively (/SS) for definitions of Executive global symbols. Note that the /SS switch must appear in this context. It is never omitted.

6. The system generation procedure links your driver to the system library file that defines masks and offsets used in the Executive. The following input continues the example from item 1:

   ```
   [1,1]EXELIB/LB
   /
   ```

   The single slash begins the option phase of the Task-Builder.

7. The Task-Builder is directed not to allocate space for a stack within the driver. For example:

   ```
   STACK=0
   ```

8. A partition for the driver is specified. For example:

```
PAR=DRVPAR:120000:20000
//
```

The partition name DRVPAR is the typical name of a conventional partition reserved for drivers. A driver may be loaded into any system-controlled partition. The base virtual address of the partition is always 120000_8. That is, the loadable driver must be mapped through kernel APR 5. The length of the partition (the second parameter) should not exceed 8K words (40000 octal bytes).

The double slash ends the option phase of the Task-Builder.

5.2.4 Loading the Driver

After your driver is task-built, you are ready to load the driver on your system. This procedure is used when you are incorporating your loadable driver with a loadable data base after system generation. To load your driver, use the privileged MCR command Load, as follows:

```
>LOAD xx:[/PAR=GEN] [/HIGH]
>
```

The variable xx is the two-character device mnemonic. Specifying a partition is optional. If a partition is not specified, the partition input to the Task-Builder is used. The keyword /HIGH puts the driver as high as possible in the partition. The default condition is to put the driver as low as possible in the partition.

LOAD performs many diagnostic checks on your driver data base, relocates many addresses within the data base, and loads the data base and the driver code into memory. Because the LOAD diagnostic checks are complicated and LOAD supports another, infrequently used option (/CTB), a description of LOAD is given in Section 5.4. LOAD error messages and meanings are listed in the *RSX-11M-PLUS MCR Operations Manual*. After the driver is loaded, the controllers and units are off line and are not accessible. To allow access to the device, you must next place the controllers and units on line.

5.2.5 Making the Devices Accessible

After your driver has been successfully loaded, you must make the controllers and units accessible. You use the CON task to place controllers and units on line, to change vector and CSR assignments that you established in the driver data base, and to take units and controllers off line. Unless the vector and CSR values in the driver data base are not correct for the running system, you can place the controllers and units on line. You may change the vector and CSR assignments to match the hardware CSR and vector assignments only while the controllers and units are off line.

5.2.5.1 Setting Vector and CSR Assignments

If the values at the offsets S.VCT/K.VCT and S.CSR/K.CSR in the KRBs of your driver data base are incorrect for the running system, you must issue the privileged SET command in CON to establish the correct values.

Note

Because CON causes the Executive to access a driver data base when it changes a vector or CSR assignment, you must load the driver before you issue the SET commands in CON. If a driver data base is resident, you do not need to load the driver to establish correct vector and CSR assignments.

You must do this operation while the controllers and units are off line. (LOAD ensures that, for a loadable data base, the controllers and units are off line.) Typical commands to set a vector and CSR for a driver that supports a single controller are as follows:

```
>CON
CON>SET xxA VEC=300 CSR=160040
CON>^Z
>
```

The command first establishes 300 as the vector for controller A of type xx. The Executive accesses the offset S.VCT/K.VCT in the driver data base and writes the specified value divided by 4. The command secondly establishes the control and status register address as 160040 for controller A of type xx. The Executive accesses the offset S.CSR/K.CSR in the driver data base and writes the specified value. You type the CTRL/Z combination to exit from CON. After you set the vector or CSR assignment, you can attempt to place the controllers and units on line.

5.2.5.2 Placing a Controller and Units On Line

If the vector and CSR assignments in your driver data base are correct for the running system, you can place the controllers and units on line by issuing the privileged ONLINE command in CON.

Note

Because placing a controller or a unit on line causes the Executive to call the driver, you must have loaded the driver before you issue the ONLINE command in CON.

The following commands demonstrate a typical sequence to place a single controller and two attached units on line:

```
>CON
CON>ONLINE xxA
CON>ONLINE xx0:
CON>ONLINE xx1:
CON>^Z
>
```

The first command places controller A of type xx on line. The Executive accesses the KRB of the controller to read the S.VCT/K.VCT offset and initializes the vector to point to the related interrupt control block.

Note

If the driver is resident within the Executive, the vector points directly to the driver. For a common interrupt controller, the vector points to the interrupt entry address in the Executive rather than to an ICB.

The Executive then ensures that the address in S.CSR/K.CSR is valid; that is, some device responds at that address. (Refer to Section 5.2.5.3 for a discussion of CSR and vector assignment errors.) Next, the Executive calls the driver at its controller status change entry point. Only after the driver indicates success does the Executive change the status bit in the SCB/KRB of the controller from offline to online.

The second and third commands place logical units 0 and 1 on line. The Executive checks that the controller is on line (that is, an access path exists to the unit). If the controller is not on line, the Executive sets the UCB of the unit as marked for online. (The Executive automatically places on line a unit that is in the marked for online state only when its controller is placed on line.) If the controller is on line, the Executive calls the driver at its unit status change entry point. Only after the driver indicates success does the Executive change the status bits in the UCB of the unit from offline to online. (The driver is not required to take any special action to indicate success. Refer to the discussion of status change entry points in Section 4.5.)

After you have issued the ONLINE commands, you can issue the DISPLAY command in CON as follows to verify that the devices are in the state that you want them to be in:

```
>CON
CON>DISPLAY FULL FOR xx
          (The display appears at the terminal.)
CON>^Z
>
```

The command displays status of all controllers of type xx and of all units attached to the controllers.

5.2.5.3 CSR and Vector Assignment Errors

CSR and vector assignment errors are not always immediately detectable. When you issue the ONLINE command to CON to place a device on line, CON verifies that some device responds at the CSR address that you established at the S.CSR/K.CSR offset in your driver data base. CON can encounter one of the following three cases:

- Your device is at the established CSR address and it responds to the CON probe. This is the case you want. CON continues attempting to place your device on line.
- Your device is at some address in the I/O page other than the established CSR address, but some other device responds at the established CSR address. CON cannot distinguish your device from some other device and continues attempting to place your device on line, possibly resulting in a system hang or failure.
- Your device is at some address in the I/O page other than the established CSR address, and no device responds at the established CSR address. In this case, CON reports an error and does not place the device on line.

If CON places your device on line and the device does not respond, have a DIGITAL Field Service representative verify the CSR address jumpers and ensure that the CSR assignment in your driver data base matches the verified hardware CSR assignment.

When the vector address developed from the value that you established at the offset S.VCT/K.VCT in the driver data base differs from the hardware vector assignment, several outcomes are possible. Should the established vector already be in use (that is, pointing at other than the nonsense interrupt entry address in the Executive), CON reports the condition and does not place the device on line. If the vector is not in use, CON establishes it as the device vector and continues attempting to place the device on line. This action does not guarantee that the software and hardware vector assignments match.

When CON does place a device on line and the software and hardware vector assignments do not match, two results are possible:

- Your driver will time out waiting for an interrupt.
- The device will interrupt through an unused vector.

If error logging is active on your system, a nonsense interrupt is logged as an undefined interrupt error and the ERRSEQ count in the Executive is increased by 1. The RMDEMO task display, which includes the ERRSEQ count, reflects the occurrence of nonsense interrupts by an increasing number in ERRSEQ. Consult an error log report and look for undefined interrupt errors.

When (1) error logging is active, (2) nonsense interrupts do not occur, and (3) your driver times out, the interrupt could be going through some other driver vector. If the unexpected interrupt goes to a DIGITAL-supplied driver, one of the following two outcomes is possible:

- The interrupt will simply be dismissed. (Common interrupt routines dismiss unexpected interrupts, and some drivers keep track of when they expect interrupts and dismiss unexpected ones.)
- The driver will react in an unpredictable fashion (such as attempting to terminate the last I/O packet again), causing a system failure.

Thus, error logging and the ERRSEQ count in the RMDEMO display help indicate improper vector assignments.

5.3 User-Supplied Driver System Generation Dialogue Summary

If you are building either a loadable driver with a resident data base or a resident driver, you must perform a system generation to incorporate your driver into the system. This section summarizes the system generation dialogue only as it relates to user-supplied driver support and related features. For more information on the system generation procedure itself, refer to the *RSX-11M-PLUS System Generation and Installation Guide.*

Note

If you are building a loadable driver with a loadable data base, you need not perform a system generation to incorporate your driver. However, you can still build your driver during system generation. Section 5.1.2 describes the complete procedures to build a loadable driver with a loadable data base any time after you build the Executive under which the driver will run.

5.3.1 Choosing Executive Options

The system features are determined during the "Choosing Executive Options" section. To specify system features, you must answer questions related to including a user-supplied driver in your system. A question appears in the following form:

```
* CE020   Do you want the Full-functionality Executive? [Y/N D:Y]:
```

If you choose the Full-functionality Executive, your driver must be loadable with either a loadable or resident data base. If you want to incorporate a user-supplied resident driver, you must omit the Full-functionality Executive and omit Executive data space support. All DIGITAL-supplied drivers should be loadable with a loadable or resident data base.

If you do not choose the Full-functionality Executive, the system generation procedure asks the following two questions:

```
* CE050   Do you want Executive data space support? [Y/N D:N]:
```

If you have a loadable driver with either a loadable or resident data base, you should answer Yes to this question. If you have a resident driver, you must answer No to this question.

```
* CE110   What is the ICB pool size (in words)? [D R:16.-1024. D:128.]:
```

On systems with Executive data space, the Interrupt Control Block (ICB) pool must be large enough for all the drivers loaded into the virgin system image. One ICB (eight words) is needed for every 16_{10} controllers of the same type. If the device controlled by your driver has a large number of controllers, you should ensure that there is enough ICB pool space.

There are no further questions in this section concerning user-supplied driver support or related features.

5.3.2 Choosing Peripheral Configuration

In the Peripheral Configuration section of the system generation procedure, you answer questions about your driver and its data base configuration. A question in the following form asks you to supply your device mnemonics:

```
* CP9604   Enter device mnemonics for user-supplied drivers [S]:
```

You must enter the two-character device mnemonic for your driver. This should be the same mnemonic used in the driver and data base source file names.

If you did not select Executive data space support, a question in the following form appears:

```
* CP9612   Do you want the xx: driver to be loadable? [Y/N D:N]:
```

Answer Yes to this question if you want a loadable driver. Answer No if you want your driver to be resident.

If your driver is loadable, the next question in the system generation procedure asks you about your data base:

```
* CP9616   Do you want xx: driver's data base to be loadable? [Y/N D:N]:
```

Answer Yes to this question if you want a loadable data base. Answer No if you want your data base to be resident.

The system generation procedure always asks you to specify the highest interrupt vector address:

```
* CP9632   What is the highest interrupt vector address? [0 R:n-774 D:n]:
```

The system generation procedure calculates and displays the highest interrupt vector address needed for the DIGITAL-supplied devices. If the vector address for your device is higher than this, enter the highest vector address used by your device.

This ends the system generation portion of incorporating a user-supplied driver. If you are generating a new system, the system generation procedure includes your driver in the Executive if it is resident or loads your driver into the system image if it is loadable. After the newly built system is running, you must make the devices that your driver supports accessible, as directed in Section 5.2.5.

If you are adding your driver after system generation, you must load your driver and make the devices that it supports accessible, as described in Sections 5.2.4 and 5.2.5.

5.4 LOAD Processing

The Executive LOAD routines extensively check the driver data base; LOAD provides the /CTB switch to handle multidriver controllers. The following subsections describe the two aspects of LOAD.

5.4.1 LOAD Operations and Diagnostic Checks

The following two LOAD modules load a driver into memory:

- LDVLDB conditionally checks the validity of and loads the data base.
- LDVFIN finishes the operation by loading the driver.

  If there is no resident data base, the data base is loaded into the system pool.

The LOAD routines relocate and validate many of the pointers within the data base and, in the process, validate other data in the structures. (If the data base is resident, no validity checks on the driver data base are performed.) The driver itself is then loaded into its partition, and the interrupt control blocks are created.

To read the data base from the xxDRV.TSK file into the system pool, the global labels $xxDAT and $xxEND, defining the start and end of the data base, are needed.

To check the data base, the LOAD routines must know the starting address of the DCB. If the global label $xxDCB is not defined (that is, it is not in the symbol table file), the start of the DCB is assumed to be the first word of the data base. Many unusual error conditions result when LOAD assumes that the DCB is at the start of the data base, but in fact, the DCB is elsewhere in the data base and not labelled properly. Thus, to avoid this type of problem, you should always define the start of the DCB with the global label $xxDCB.

The LOAD routines perform the following operations to load and validate the data base:

- Each CTB is checked and relocated. The following offsets are both checked and relocated:

L.LNK — The link to the next CTB must be even. If it is not zero, it must point within the data base, and the CTB to which it points must lie within the data base. (Because it is highly unusual to have two controller types in one driver data base, this value is usually zero.)

L.DCB — The address of the related DCB must be even, point within the data base, and the DCB to which it points must lie within the data base. If L.DCB points to a common interrupt table, the common interrupt entry point address in the table must be even and lie within the Executive. The DCB addresses in the table must be even, and the DCBs to which each address points must lie within the data base.

L.KRB — Each pointer in the table of KRB addresses must be even and must point within the data base, and the KRB to which each cell points must lie within the data base.

- The following offsets in the CTB are checked:

  L.NAM — The controller name cannot duplicate other L.NAM entries in the resident or loadable data base.

  L.NUM — The number of controllers must be less than 17_{10}.

- Each KRB is checked and relocated. The following offsets in the KRB are both checked and relocated:

  K.OWN — The pointer to the owner UCB must be even and point within the data base, or it must be zero. If it is nonzero, the pointer is relocated.

  K.OFF — The start of the table of UCB addresses produced from K.OFF must be even and must point within the data base. The entries themselves must be even, point within the data base, and the UCB to which each cell points must lie within the data base.

  K.CRQ
K.CRQ+2 — This is the listhead for the controller request queue. It is initialized to an empty list with the first word zero and the second word, pointing to the first, relocated.

- The following offset in the KRB is checked:

  K.URM — In a multiprocessor system, the UNIBUS run mask for the controller must have exactly one bit set and that bit must correspond to an existing UNIBUS run (either primary or secondary).

- LOAD puts each controller in the offline state by setting the KS.OFL bit in the K.STS byte. Therefore, all controllers are off line until you use CON to place each one on line.

- Each DCB is checked and relocated. The following offsets are both checked and relocated:

  D.LNK — The link to the next DCB must be even. If it is nonzero, it must point within the data base, and the DCB to which it points must lie within the data base.

  D.UCB — The link to the first UCB must be even and must point within the data base, and the UCB to which it points must lie within the data base.

- The following offsets in the DCB are checked:

  D.NAM — The device name must be the same as that which you specified in the LOAD command line.

  D.UCBL — The length of the UCB must be even and nonzero.

  D.UNIT — The highest unit number (increased by 1) used with D.UCBL forms the last address of all UCBs. This address must lie within the data base.

- The pointer to the driver dispatch table (D.DSP) is set to zero to show that the driver is not loaded.

- Each UCB is checked and relocated. The following offsets are both checked and relocated:

  U.DCB — The pointer to the DCB must point to the DCB that points to this UCB.

  U.SCB — The pointer to the SCB must be even, must point within the data base, and the SCB to which it points must lie within the data base.

  U.RED — The unit redirect pointer must be nonzero and even if it is an Executive address. If it is not an Executive address, it must be nonzero, even, and point within the data base.

- LOAD places each unit in the offline state by setting the US.OFL bit in the U.ST2 byte. Therefore, all units are off line until you use CON to place each one on line.

- Each SCB is checked and relocated. The following offsets are both checked and relocated:

  S.KRB — The pointer to the KRB must be even, must point within the data base, and the KRB to which it points must lie within the data base. If S.KRB is nonzero, there must be a CTB in the loadable data base.

  S.KTB — If the table of KRB addresses is present, each entry must point within the data base. (LOAD preserves bit zero in each entry.) Each entry in the table must also have a matching entry in the table of KRB addresses of a CTB in the loadable data base.

- The following offsets in each SCB are initialized as described:

  S.LHD — The head of the I/O queue is set to zero and the pointer to the end of the queue (S.LHD+2) is set to point at S.LHD.

  S.PKT — The pointer to the current I/O packet is set to 1.

After the data base is loaded and validated and no error is found, the driver itself is loaded into memory. When the driver is loaded, the driver dispatch table is validated, each interrupt entry in the driver dispatch table is inspected, and the vectors are checked. If a vector address is higher than the highest vector address allowed on the system (as specified at system generation) or does not point to a nonsense interrupt entry point, LOAD prints a warning message. You can use CON to set the correct vector address before you place the controller on line. Interrupt control blocks are created and linked into the list starting at L.ICB in the CTB.

The format of the DDT must be consistent with that described in Section 4.5.1. If the device that the data base describes does not have any physical controllers (that is, the value at offset S.VCT/K.VCT equals zero), the DDT is not checked. If S.VCT/K.VCT is nonzero, the device has at least one interrupt vector and therefore at least one interrupt entry point. The DDT is then checked. The two global labels $xxTBL and $xxTBE must define the start and end of the

DDT. The generic controller names must be nonzero and the interrupt entry values must be valid. Interrupt entry point 0 must be nonzero, even, and lie in the range 117777 to 140000. If the format of DDT is inconsistent, LOAD prints an error message, restores the system device tables, and exits.

When the driver is loaded, all links are established. The DCB of the loadable data base is put in the list of DCBs just in front of the DCB for the first pseudo device. The CTBs are linked to the end of the CTB list. The DDT address D.DSP, the driver PCB address D.PCB, and the driver mapping S.KS5 (the block number of the first word of the driver) in the fork block are initialized. The address of the start of the KRB table in the CTB, denoted in the driver data base by the global label $xxCTB, is loaded into the DDT.

5.4.2 Use of /CTB in LOAD

Some controllers, such as the RH70, can support more than one device type. The CTB for such a controller differs in two ways from the standard CTB. First, the table of KRB addresses at the end of the CTB contains pointers to KRBs of controllers for different device types. Second, instead of a pointer to one DCB in the CTB, there is a pointer to a table of DCB addresses because each different device must have a separate DCB to describe each separate device type. Moreover, more than one driver supports the different types of devices capable of being attached to the controller.

The data base for such a controller must be split. Because only one CTB is needed to describe the type of controller, only one driver that supports a device on that controller type can define that CTB. The remaining drivers cannot define a CTB but must reference the CTB defined for the first driver. Because all drivers and their data bases can be loadable, the remaining drivers and the syntax in the LOAD command must indicate to the LOAD routines which resident CTB to use. (Of course, the driver data base that defines the CTB of the multidriver controller must be loaded or already resident before the other drivers can be loaded.)

The driver data base that defines the CTB for a multidriver controller allows for structures to define the data base of drivers that are not resident. In particular, for each device controller there must be a slot in the CTB table of KRB addresses to hold the pointer to the KRB. (A KRB must be defined to describe each occurrence of a controller.) A zero is in the pointer for a device whose data base (and, therefore, whose KRB) is not resident. Moreover, the table of DCB addresses in the common interrupt table must have sufficient slots to point to the DCBs of all device types that the controller supports. A zero in the DCB table indicates no DCB exists (that is, the data base for a device type is not resident).

To load the data base of a device attached to the multidriver controller, the LOAD routines must know the controller name, the location of the device on the MASSBUS controller, and the KRBs of the devices whose driver is to be loaded. The /CTB syntax in the LOAD command supplies the first two pieces of information; for example,

```
>LOA DR:/CTB=RHB,C
>
```

The letters RH are the name in the CTB already resident in the system. LOAD routines search the system list of CTBs to locate the correct one. The letters B and C are the slots in the table of KRB addresses that will be used to link the resident CTB with the KRB in the data base being loaded.

The name of the device DR reflects the name in the DCB that is being loaded. An empty slot in the table of DCB addresses in the resident data base will be made to point to this DCB.

The LOAD routines need to find the correct KRB in the data base being loaded. A global label of the form $cca (where cc is the controller name and a is the slot, or controller number) must define the start of the KRBs being loaded. Thus, the loadable data base for the example above must contain the labels $RHB and $RHC, which are the KRB names. The address of the label is loaded into the appropriate slot in the CTB table of KRB addresses.

In summary, then, the /CTB syntax on the LOAD command, combined with the global labels, allows the LOAD routines to link a driver data base being loaded with a currently resident driver data base. The KRBs being loaded are incorporated in the resident data base and the DCB being loaded is connected to the common interrupt table.

5.4.3 Incorporating a Driver into a Micro/RSX System

Since there is no system generation procedure under Micro/RSX, you can incorporate neither a resident driver with a resident data base nor a loadable driver with a resident data base into Micro/RSX. Your system will support only a loadable driver with a loadable data base.

The Micro/RSX startup procedure allows you to incorporate a user-supplied driver by altering the startup parameters of your system, as follows:

1. Assemble your driver and data base using the following commands:

```
MAC>xxDRV,xxDRV=[11,10]RSXMC/PA:1,[directory]xxDRV
MAC>xxTAB,xxTAB=[11,10]RSXMC/PA:1,[directory]xxTAB
```

   These commands for a loadable driver are located in xxDRVASM.CMD, where xx is the device mnemonic.

   The commands to the assembler specify as input the Executive assembly prefix file RSXMC.MAC and your driver code and driver data base source files (xxDRV.MAC and xxTAB.MAC). The [directory] specification is not optional; each source code file specification should include the directory in which you have placed the driver or data base.

2. Task-build your driver using the following commands:

```
[1,54]xxDRV/-MM/-HD/-PI,[directory]xxDRV,[1,54]xxDRV=xxDRV,xxTAB
[1,54]RSX11M.STB/SS
/
UNITS=0
PAR=GEN:120000:20000
STACK=0
//
```

   These commands for a loadable driver are located in xxDRVBLD.CMD, where xx is the device mnemonic.

3. When you are sure your driver is assembled and built correctly, edit your configuration file, [1,2]SYSPARAM.DAT, to specify your driver mnemonic in the DRIVER statement. The driver statement should read as follows (where xx is your driver mnemonic):

```
DRIVER=xx:
```

This statement causes the startup procedure to load your driver. You may include this statement in the configuration file as many times as necessary to load the drivers you require.

After the startup procedure loads your driver, it will interpret the default CON_ONLINE_ALL=yes statement in the configuration file and bring your driver on line.

4. Run the startup procedure to incorporate your driver.

Chapter 6
Debugging a User-Supplied Driver

Adding a user-supplied driver carries with it the risk of introducing obscure bugs into an RSX-11M-PLUS system. Because the driver runs as part of the Executive, special debugging tools are both desirable and necessary. RSX-11M-PLUS provides the following aids, which can be incorporated into your system during system generation:

- Crash dump analysis support routine (CDA)
- Executive debugging tool (XDT)

You need not select any of this software during system generation. However, if you do require the facilities they offer, you can select them for incorporation in your system. The following sections describe the features and use of each debugging aid.

6.1 Crash Dump Analysis Support Routine

The crash dump analysis (CDA) support routine prints the following message on a notification device specified at system generation:

```
CRASH - CONT WITH SCRATCH MEDIA ON device mnemonic
```

You can then be sure that the secondary crash dump device is ready and depress the CONT switch on the operator's console. The Executive Crash Dump routine will dump memory to the crash dump device and halt the processor upon completion.

The procedure for subsequently invoking CDA, which reads and formats the memory dump, is documented in the *RSX-11M-PLUS and Micro/RSX Crash Dump Analyzer Reference Manual.*

6.2 The Executive Debugging Tool

An interactive debugging tool aids in debugging Executive modules, I/O drivers, and interrupt service routines. This debugging aid, called XDT, is a version of RSX-11 ODT. Including XDT in a system with Executive data space support does not reduce the size of pool space that the system can have. XDT occupies physical address space but does not take up any Executive virtual data address space. Also, XDT does not interfere with user-level RSX-11 ODT, which can be used with any number of tasks while you are debugging your driver with XDT.

You can include XDT in a system during the "Choosing Executive Options" section of system generation when the following question is asked:

```
Do you want to include XDT? [Y/N D:N]
```

If you answer Y, XDT is linked into the Executive image when you build the Executive.

6.2.1 XDT Commands

The *RSX-11M-PLUS and Micro/RSX XDT Reference Manual* contains a guide to debugging with XDT. While XDT commands are generally compatible with RSX-11 ODT commands (described in the *RSX-11M-PLUS and Micro/RSX Debugging Reference Manual*), XDT does not contain the following commands available in ODT:

No $M	-	(Mask) register
No $X	-	(Entry Flag) registers
No $V	-	(SST vector) registers
No $D	-	(I/O LUN) registers
No $E	-	(SST data) registers
No $W	-	(Directive status word) $DSW word
No E	-	(Effective Address Search) command
No F	-	(Fill Memory) command
No N	-	(Not word search) command
No V	-	(Restore SST vectors) command
No W	-	(Memory word search) command

In addition, the X (Exit) ODT command has been changed for XDT. The X command causes a jump to the crash dump routine.

Except for these omitted features and the change in the X command, XDT commands are compatible with RSX-11 ODT; consequently, the *RSX-11M-PLUS and Micro/RSX XDT Reference Manual*, together with the discussion in Section 6.2 in this manual, can be used as a guide to XDT operation on RSX-11M-PLUS.

XDT includes both Instruction space and Data space address referencing. The following commands control the current address referencing:

I Sets address references to Instruction space

D Sets address references to Data space

Address references set to Data space is the default condition when XDT starts up.

6.2.2 XDT Startup

When you bootstrap a system that includes XDT, the normal system startup transfers control to XDT, which identifies itself at the system console terminal with the following message:

```
XDT: <system name and version>
```

If no errors were encountered, the identification message is followed by the prompt:

```
XDT>
```

The following are the register conditions when XDT starts:

R0 = CSR address of the bootstrap device

R1 = LBN of the system image

R2 = LBN of the system image

R3 = physical unit number of the load device

R4 = ASCII name of the load device

R5 = total number of blocks read from the system image

XDT runs entirely at priority level 7.

You can set breakpoints at this time and then give a G command, passing control to the Executive initialization module INITL. Whenever control reaches a breakpoint, a printout similar to that of RSX-11 ODT occurs.

If INITL encounters an error condition, it prints an error message preceded by a prefix telling whether the condition is a warning or fatal. If the condition is a warning, XDT has control. You can set breakpoints to establish control or type the P command to proceed. If the condition is fatal, the processor halts. You must correct the condition before you can rebootstrap your system.

6.2.3 XDT Restrictions

On some types of systems, the following restrictions on the use of XDT exist when it is first entered:

- On all systems:

  Some data structures are not yet initialized. The secondary pool is not set up and the console terminal and the bootstrapped device are not on-line.)

- On systems with Kernel data space support:

  Data space mapping is not yet set up. Certain Executive locations that the Task-Builder could not resolve are not initialized. (The RH common interrupt table address [$RHTBX] does not contain the RH common interrupt routine address [$RHALT].)

To proceed when you encounter such restrictions on your system, at the initial XDT prompt you should first set a breakpoint near the end of the INITL module (after the routine that sets up the data structures). Then, after you proceed and XDT encounters the breakpoint near the end of INITL, use XDT to examine locations in the Executive and to set more Executive breakpoints.

On a multiprocessor system, you should be aware of the following conditions:

- When you initially place a processor on-line, XDT does not prompt from that processor unless you have set the processor's bit in the $XDTPR word.
- XDT does not handle multiprocessor-specific conditions. You cannot set processor-specific breakpoints nor can you easily examine other processors' low memory context.
- Under certain circumstances (such as when Data space is not yet set up), setting a breakpoint in shared Instruction space may eventually cause a trap on a processor other than the one on which you set the breakpoint. Consequently, because the processor encountering the breakpoint does not have that breakpoint in its XDT tables, XDT generates a breakpoint error message (BE:) rather than a breakpoint message.
- All processors are locked out of the Executive while XDT is being executed by one of the other processors.

6.2.4 XDT General Operation

A forced entry to XDT can be executed at any time from a privileged terminal by means of the MCR Breakpoint (BRK) command. Thus, if your system has no XDT restrictions as described in Section 6.2.3, the normal procedure is to type G when the system is bootstrapped without setting any breakpoints. When it becomes necessary to use XDT, the MCR Breakpoint command is executed, causing a forced breakpoint. XDT then prints on the console terminal:

```
BE:xxxxxx
```

This message is followed by the prompt:

```
XDT>
```

You can then set breakpoints and issue other XDT commands. Continue system operation by typing the Proceed (P) command to XDT.

All XDT command I/O goes to and from the console terminal, and the List Memory (L) command can list on either the console terminal or the line printer.

6.2.5 XDT and Debugging a User-Supplied Driver

Using XDT to debug a loadable driver has special pitfalls. One problem that can arise, a T-bit error, generates the following error message:

```
TE:xxxxxx
XDT>
```

This error results when control reaches a breakpoint that you have set, using XDT, in a loaded driver. The T-bit error, rather than the expected BE: error, occurs unless register APR 5 is mapped to the driver at the time XDT sets the breakpoint.

To avoid this T-bit error, assemble the driver with an embedded BPT instruction, or use either the ZAP utility or the MCR OPEN function to set the breakpoint by replacing a word of code with the BPT instruction. You can use the OPEN command to access the driver as follows:

```
>OPE nnnn/DRV=xx:
```

where nnnn matches the address in the driver map listing and xx is the device mnemonic. (Write down the instruction that you replace with the BPT instruction.)

When control reaches a breakpoint set in the driver, XDT prints the following message:

```
BE:xxxxxx
XDT>
```

Recover as follows:

1. Using XDT, replace the BPT instruction with the desired instruction.
2. Decrement the PC by subtracting 2 from the contents of Register 7.
3. Then proceed by using the P or S commands.

Note

You should not set breakpoints in more than one module that maps into the Executive through APR 5 or APR 6. In particular, do not set breakpoints in more than one loadable driver at a time or XDT will overwrite words of main memory when it attempts to restore what it considers to be the contents of breakpoints.

6.3 Fault Isolation

One or more of the following causes may be identified when the system faults:

- A user-state task has faulted in such a way that it causes the system to fault.
- The user-supplied driver has faulted in such a way that it causes the system to fault.
- The system software itself has faulted.
- The host hardware has faulted.

When the system faults, you must first decide which of these four causes is responsible. This section presents some procedures that can help you isolate the source of the fault. Correcting the fault itself is your responsibility.

6.3.1 Immediate Servicing

Faults manifest themselves in four ways (listed here in order of increasing difficulty of isolation):

1. If XDT is included, an unintended trap to XDT occurs.
2. The system displays text indicating a failure has occurred and halts.
3. The system halts but displays nothing.
4. The system is in an unintended loop.

The following discussions assume the existence of a system built with at least one debugging aid.

The immediate aim, regardless of the fault manifestation, is to get to the point where you can obtain pertinent fault isolation data.

6.3.1.1 The System Traps to XDT

The trap may or may not be intended. If it is not intended, type the X command, causing XDT to exit to location 40_8, from which the CDA support routine will be invoked. If, however, you have some idea of the source of the problem (for example, a recent coding change), then you may use XDT to examine pertinent data structures and code.

6.3.1.2 The System Reports a Crash

If the text displayed on the console terminal consists of output from the CDA support routine, follow the procedure for obtaining and formatting a memory dump as outlined in the *RSX-11M-PLUS and Micro/RSX Crash Dump Analyzer Reference Manual.*

6.3.1.3 The System Halts but Displays No Information

Before taking any action, preserve the current PSW and PC and the pertinent device registers (that is, examine and record the information these registers contain). The procedure depends on the particular PDP-11 processor. Consult the appropriate PDP-11 processor handbook for details.

After preserving the PSW and PC, invoke your resident debugging aid by entering 40_8 in the switch register, pressing LOAD ADDR, and then pressing START. The contents of 40_8 cause the invocation of the CDA support routine.

6.3.1.4 The System Is in an Unintended Loop

Proceed as follows:

1. Halt the processor.
2. Record the PC, the PSW, and any pertinent device registers, as in Section 6.3.1.3.

You may then want to step through a number of instructions in an attempt to locate the loop. For this attempt to be meaningful you must first disable the system clock. Proceed as follows:

1. Examine the contents of word 777546 (if your system has a line-frequency clock) or word 772540 (if your system has a programmable clock).
2. Clear bit 6 in this word and redeposit the word.

Note

Until you reenable the clock, some system operations do not work because they are waiting for time. You can type and the system echoes typed characters. You can input MCR commands.

After trying to locate the loop and reenabling the clock, transfer to location 40_8 as in Section 6.3.1.3.

6.3.2 Pertinent Fault Isolation Data

Before you attempt to locate the fault, you should dump system common (SYSCM), which contains a number of critical pointers and listheads. CDA always dumps the SYSCM area. In addition, you should dump the dynamic storage region (system pool and, if it exists, the ICB pool) and the device tables. The device tables are in the module SYSTB.

At this point, you have the following data:

- PSW
- PC
- The stack
- Registers 0 through 6
- Pertinent device registers
- The dynamic storage region
- The device tables
- System common

These data represent a minimal requirement for effectively tracing the fault.

6.4 Tracing Faults

Three pointers in SYSCM are critical in fault tracing. These pointers are described below:

$STKDP—Stack Depth Indicator

This data item indicates which stack was being used at the time of the failure. $STKDP plays an important role in determining the origin of a fault. The following values apply:

+1	User (task-state) stack or a privileged task at user state
0 or less	System stack

If the stack depth is +1, then the user has caused the system to fail.

$TKTCB—Pointer to the Current Task Header (TCB)

This is the TCB of the user-level task in control of the CPU.

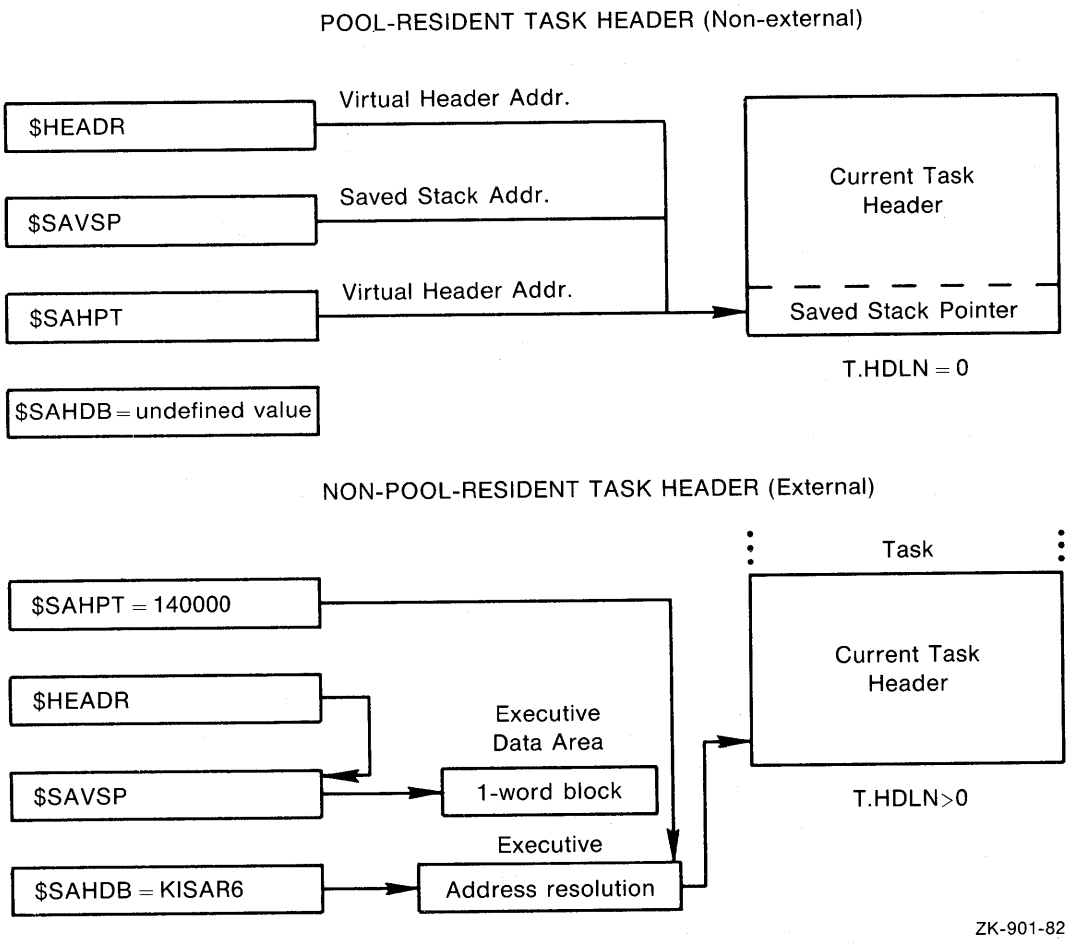
$HEADR—Pointer to the Current Task Header (Pool-Resident)

The location of the task header and the contents of its associated pointers vary according to whether the task has an external header. A task with an external header has its header attached in a physically contiguous and numerically lower location in memory. A task with a nonexternal header has its header located in Executive pool space. Therefore, a header in Executive pool is a pool-resident header, and a header adjacent to the task is a non-pool-resident header.

Figure 6–1 shows the interaction of header pointers for both pool-resident and non-pool-resident headers. For a pool-resident task header, $HEADR, $SAHPT, and $SAVSP all point to the first word of the task header. This word also contains the user task's stack pointer (SP) from the last time it was saved. Figure 6–2 shows a brief description of the

task header. The task header is fully described in the *RSX-11M-PLUS and Micro/RSX Task Builder Manual.*

Figure 6–1: Interaction of Task Header Pointers

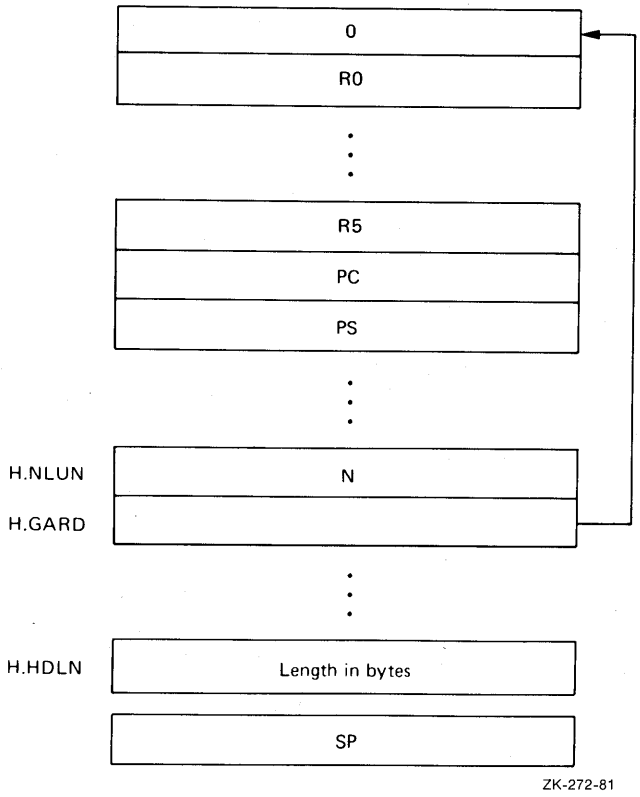


The header (as pointed to by $HEADR) also contains the last-saved register set, just before the header guard word (the last word in the header—pointed to by H.GARD).

The following pointers are associated with the header:

- $HEADR
- $SAVSP
- $SAHPT
- $SAHDB

Figure 6–2: Task Header



The pointers associated with a pool-resident header are described as follows:

$HEADR – Points to the current task header.

The $HEADR word points to the pool-resident task of the task currently running. (The value in $HEADR is a kernel virtual address in primary pool.)

$SAVSP – Points to the first word of the current task header, which contains the saved stack pointer.

$SAHPT – Points to the current task header in pool. ($SAHPT contains the virtual address of the header. $SAHPT and $HEADR contain the same virtual address for a pool-resident header.)

$SAHDB – Contains an unknown value.

The pointers associated with a non-pool-resident header are described as follows:

$HEADR – Points to the pointer for the saved stack pointer, $SAVSP.

$SAVSP – Points to a 4-word block in the Executive data area.

$SAHPT – Contains the octal value of 140000 that is to be used with $SAHDB to resolve the address of the task's header. ($SAHPT always contains 140000 in this case.)

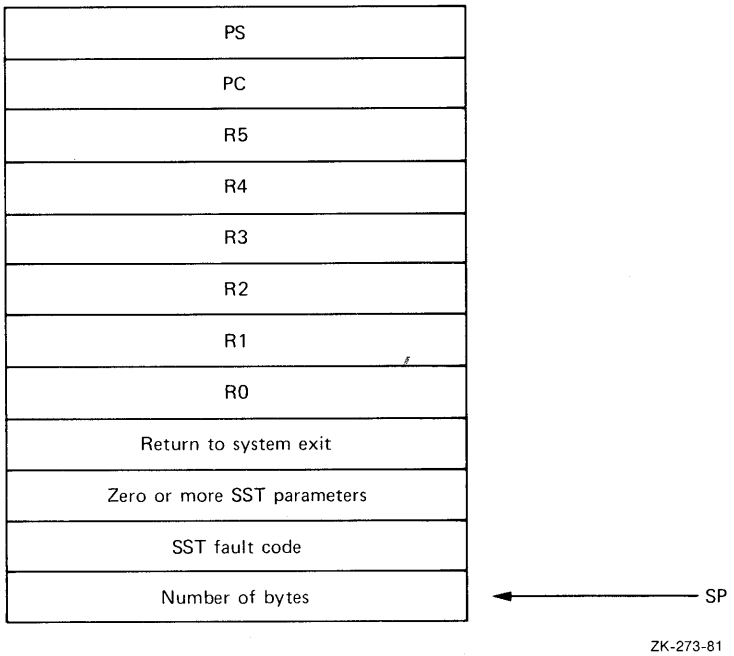
$SAHDB – Contains the value in KISAR6, which is a 32-word block-offset to be used with the value in $SAHPT to resolve the address of the task's header.

6.4.1 Tracing Faults Using the Executive Stack and Register Dump

To trace a fault after a display of the Executive stack and register contents, first examine the system stack pointer. Usually an Executive failure is the result of an SST-type trap within the Executive. If a synchronous system trap (SST) does occur within the Executive, then the origin of the call on the crash-reporting routine is in the SST service module. (The crash call is initiated by issuing an IOT at a stack depth of zero or less.)

A call to crash also occurs in the Directive Dispatcher when an EMT is issued at a stack depth of zero or less, or a trap instruction is executed at a stack depth of less than zero. The stack structure in the case of an internal SST fault is shown in Figure 6–3.

Figure 6–3: Stack Structure: Internal SST Fault



The fault codes are as follows:

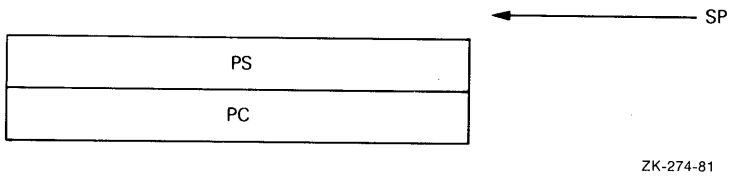
```
0       ; ODD ADDRESS AND TRAPS TO 4
2       ; MEMORY PROTECT VIOLATION
4       ; BREAK POINT OR TRACE TRAP
6       ; IOT INSTRUCTION
10      ; ILLEGAL OR RESERVED INSTRUCTION
12      ; NON RSX EMT INSTRUCTION
14      ; TRAP INSTRUCTION
16      ; 11/40 FLOATING POINT EXCEPTION
20      ; SST ABORT-BAD STACK
22      ; AST ABORT-BAD STACK
24      ; ABORT VIA DIRECTIVE
26      ; TASK LOAD READ FAILURE
30      ; TASK CHECKPOINT READ FAILURE
32      ; TASK EXIT WITH OUTSTANDING I/O
34      ; TASK MEMORY PARITY ERROR
```

The PC points to the instruction following the one that caused the SST failure. The number of bytes is the number normally transferred to the user stack when the particular type of SST occurs. If the number is 4, then a non-normal SST fault occurred, and only the PSW and PC are transferred. There are no SST parameters.

If the failure is detected in $DRDSP, the stack is the same as that shown in Figure 6–3, except that the number of bytes, the SST fault code (the fault codes are listed above), and the SST parameters are not present.

One SST-type failure, stack underflow, does not result in the stack structure of Figure 6–3. To determine where the failure occurred, first establish the stack structure; this can be deduced by the value of the SP and the contents of the top word on the stack. If the stack structure is that of Figure 6–3, then the failure occurred in $DRDSP, or was a normal SST failure. If the stack structure is that of Figure 6–4, then an abnormal SST failure has occurred.

Figure 6–4: Stack Structure: Abnormal SST Fault



Abnormal SST failures occur when it is not possible to push information on the stack without forcing another SST fault. When this situation occurs, a direct jump to the crash-reporting routine is made rather than an IOT failure. The PSW and PC on the stack are those of the actual failure, and the address printed out by the crash-reporting routine is the address of the fault rather than the address of the IOT that causes the system to fail. Note that the crash-reporting routine removes the PC and PSW of the IOT instruction from the stack, which in this case is incorrect. Thus, the SP appears to be four bytes greater than it really is (as in Figure 6–4).

The material in this section can help you isolate the cause of a failure. However, the best teachers are personal experience and a knowledge of the interaction between the driver and the services provided by the Executive.

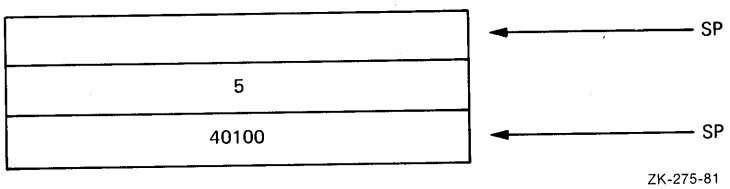
6.4.2 Tracing Faults When the Processor Halts Without Display

To trace a fault when the processor halts but displays no information, first examine $STKDP, $TKTCB, $HEADR, $SAVSP, $SAHPT, and $SAHDB. The difficulty in tracing failures in this case is that the system stack is not directly associated with the cause of a failure.

By examining $STKDP, you can determine the system state at the time of failure. If it was in user state, the next step is to examine the user's stack. The examination focuses on scanning the stack for addresses that may be subroutine links, which can ultimately lead to a thread of events isolating the fault. This is essentially the aim of looking at the system stack if $STKDP is zero or less.

Frequently, a fault can occur that causes the SP to point to Top of Stack (TOS) +4. This fault results from issuing an RTI when the top two items on the stack are data. The result is a wild branch and then, most probably, a halt. Figure 6–5 shows a case in which two data items are on the stack when the program executes an RTI. TOS points to a word containing 40100. Suppose that location 40100 contains a halt. This indicates that the original SP was four bytes below the final SP and fault tracing should begin from the original SP.

Figure 6–5: Stack Structure: Data Items on Stack



This type of fault also occurs when an RTS instruction is executed with an inconsistent stack. However, in that case, SP points to TOS +2.

A scan of the contents of the general registers may give some hint as to the neighborhood in which a fault (or the sequence of events leading up to the fault) occurred.

If the fault occurred in a new driver, a frequent source of clues is the buffer address and count words in the UCB (U.BUF, U.BUF +2, U.CNT), as are the activity flags (US.BSY and S.STS). Other locations in both the UCB and SCB may also provide information that may help locate the source of the fault.

6.4.3 Tracing Faults After an Unintended Loop

To trace a fault when an unintended loop has occurred, first halt the processor.

After you halt the processor, the same state exists as was discussed in Section 6.4.2. Follow the same tracing procedure described there. A specific suggestion is to check for a stack overflow loop. Patterns of data successively duplicated on the stack indicate a stack looping failure.

6.4.4 Additional Hints for Tracing Faults

Another item to check is the current (or last) I/O packet, whose address is found in S.PKT of the SCB. The packet function (I.FCN) defines the last activity performed on the unit.

If trouble occurred in terminating an I/O request, a scan of the system dynamic memory region may provide some insight. This region starts at the address contained in $CRAVL, a cell in SYSCM. Because all I/O packets are built in system dynamic memory, their memory is returned to the dynamic memory region when they are successfully terminated. Following the link pointers in this region may reveal whether I/O completion proceeded to that point. In systems with QIO optimization, $PKAVL (SYSCM) points to a list of I/O packet-sized blocks of dynamic memory that are not linked into the $CRAVL chain.

A frequent error for an interrupt-driven device is to terminate an I/O packet twice when the device is not properly disabled on I/O completion and an unexpected interrupt occurs. This action ultimately produces a double deallocation of the same packet of dynamic memory. Double deallocation of a dynamic buffer causes a loop in the module $DEACB on the next deallocation (of a block of higher address) after the second deallocation of the same block. At that time, Registers 2 and 3 both contain the address of the I/O packet memory that has been doubly deallocated. If XDT has been included in the system, the deallocation routine checks for bad deallocation and causes the system to fail if it occurs.

6.5 Rebuilding and Reincorporating a Loadable Driver

After correcting and assembling the driver source and updating the Executive object library, simply unload the old version using the MCR command UNLOAD, task-build the new one, and load it using the LOAD command. The commands for the Assembler, librarian, and Task-Builder are shown in Section 5.2.

Once loaded, the data base is not removed by the UNLOAD command. If the data base is in error and cannot be patched, correct its source, reassemble it, update the Executive object library RSX11M.OLB, and build the new driver task. Then bootstrap the system before loading the driver task image containing the corrected data base.

Chapter 7

Executive Services Available to an I/O Driver

Because a driver is mapped within the Executive address space, it can call Executive routines on the same basis as that of any other module in the Executive. The driver must observe the protocol and conventions established on the system. The following sections summarize the conventions, describe the address double word, explain what special processing is required for NPR devices attached to a PDP-11 processor with extended memory support (22-bit addressing), and summarize some of the typical Executive services available.

7.1 System-State Register Conventions

In system state, Registers 5 and 4 are nonvolatile registers, by convention. This means that an internally called routine is required to save and restore these two registers if the routine destroys their contents. Registers 3, 2, 1, and 0 are volatile registers and may be used by a called routine without save and restore responsibilities.

When a driver is entered directly from an interrupt, it is operating at interrupt level, not at system state. At interrupt level, any register the driver uses must be saved and restored. INTSV$ generates code to preserve Registers 5 and 4 for the driver's use. All drivers must follow these conventions.

See the description of the driver dispatch table in Section 4.5 for the contents of registers when a driver is entered.

7.2 The Address Doubleword

RSX-11M-PLUS can accommodate configurations whose maximum physical memory is 2048K words. Individual tasks, however, are limited to 32K words. The addressing is accomplished by using virtual addresses and memory mapping hardware. I/O transfers, however, use physical addresses 18 bits in length. Since the PDP-11 word size is 16 bits, some scheme is necessary to represent an address internally until it is actually used in an I/O operation. DIGITAL decided to encode two words as the internal representation of a physical address and to transform virtual addresses for I/O operations into the internal doubleword format.

On receipt of a QIO directive, the buffer address in the Directive Parameter Block, which contains a task virtual address, is converted to address doubleword format.

The virtual address in the DPB is structured as follows:

Bits 0 through 5	Displacement in terms of 32-word blocks
Bits 6 through 12	Block number
Bits 13 through 15	Page Address Register Number (PAR#)

The internal RSX-11M-PLUS translation restructures this virtual address into an address doubleword as described in the following paragraphs.

The relocation base contained in the PAR (specified by the PAR number in the virtual address in the DPB) is added to the block number in the virtual address. The result becomes the first word of the address doubleword. It represents the nth 32-word block in a memory viewed as a collection of 32-word blocks. Note that at the time the address doubleword is computed, the user task issuing the QIO directive is mapped by the processor's memory management registers.

The second word is formed by placing the block displacement (bits 0 through 5 of virtual address) into bits 0 through 5. The block number field was accommodated in the first word and bits 6 through 12 are cleared. Finally, a 6 is placed in bits 13 through 15 to enable use of PAR #6, which the Executive uses to service I/O for program transfer devices.

For non-processor request (NPR) devices, the driver requirements for manipulating the address doubleword are direct and are discussed with the description of U.BUF in Section 4.4.4.

7.3 Drivers for NPR Devices Using Extended Memory

Special features must be built into drivers for non-MASSBUS NPR (non-processor request) devices attached to a PDP-11 processor with extended-memory support (22-bit addressing).

Non-extended memory NPR devices on the PDP-11 processor must perform I/O transfers using UNIBUS Mapping Registers (UMRs), as described in the *PDP-11 Processor Handbook*. One UMR is required for each 4K words involved in the transfer—as specified by the contents of U.CNT in the UCB. When multiple UMRs are required for a transfer, they must be contiguous.

A driver can be assigned UMRs through any one of the following three procedures:

- Dynamically allocating UMRs for the duration of the data transfer
- Dynamically allocating UMRs for longer periods of time
- Statically allocating UMRs during system generation

Note

In large systems, using the procedures above to hold UMRs for longer periods than necessary can result in the blocking of other drivers and a reduction in system throughput.

7.3.1 Calling $STMAP and $MPUBM or $STMP1 and $MPUB1

To obtain UMRs through use of the $STMAP and $MPUBM or the $STMP1 and $MPUB1 routines, a driver must observe the following conventions:

- Have available six words for a mapping register assignment block in the 22-bit working storage area of the device's controller request block (KRB). The end of this area is accessed by adding the contents of K.OFF to the address at K.CSR. If the driver uses $STMP1 and $MPUB1, it must also have available a 10-word block.
- Call the routine $STMAP or $STMP1 (Set Up UNIBUS Mapping Address) after getting the I/O packet.
- Call the routine $MPUBM or $MPUB1 (Map UNIBUS to Memory) before initiating a transfer.

These requirements are detailed in the following three subsections. Note that these routines are only required when the driver is performing a data transfer.

7.3.1.1 Allocating a Mapping Register Assignment Block

The controller request block (KRB) of an NPR device requires a 6-word mapping register assignment block located in the 22-bit working storage area. It does not have to be initialized. Any initial contents are simply overwritten.

The following example shows the allocation of a mapping register assignment block.

```
.BLKW   M.LGTH  ;UMR WORK AREA
```

If the driver does not support parallel NPR operations requiring UMR mapping, it calls $STMAP and $MPUBM. If the driver supports parallel NPR operations requiring UMR mapping, it must call $STMP1 and $MPUB1. In the latter situation, the six additional words in the 22-bit working storage area are not used but must still be present. In addition, the driver data base must provide a 10-word mapping register assignment block for each data transfer to be mapped, as specified in the description of $STMP1 later in Section 7.4.31.

7.3.1.2 Calling $STMAP or $STMP1

In the coding at the initiator entry point, after the call to $GTPKT, the NPR-device driver must call the routine $STMAP or $STMP1. These routines dynamically allocate required UMRs. If UMRs are not available immediately, the driver is blocked. Such blocking, if it occurs, is completely transparent to the driver. The driver resumes processing at fork level when the UMRs have been allocated. The register returns are absolutely identical whether or not blocking has occurred.

$STMAP or $STMP1 stores into U.BUF and U.BUF+2 (in the UCB), a UNIBUS address that causes the appropriate UMR to be selected for mapping the transfer. The call to $STMAP or $STMP1 must be conditionalized on M$$EXT.

7.3.1.3 Calling $MPUBM or $MPUB1

Before executing the transfer, the driver must call $MPUBM or $MPUB1. These routines get the buffer's 22-bit physical address and load the UNIBUS mapping registers so that transfers are mapped directly to the task's space. The call to $MPUBM or $MPUB1 must be conditionalized on M$$EXT.

If the driver calls $STMAP and $MPUBM, the UMRs allocated to it are deallocated during the call to $IODON or $IOALT. If the driver calls $STMP1 and $MPUB1, it must call $DEUMR to deallocate any allocated UMRs before calling $IODON or $IOALT.

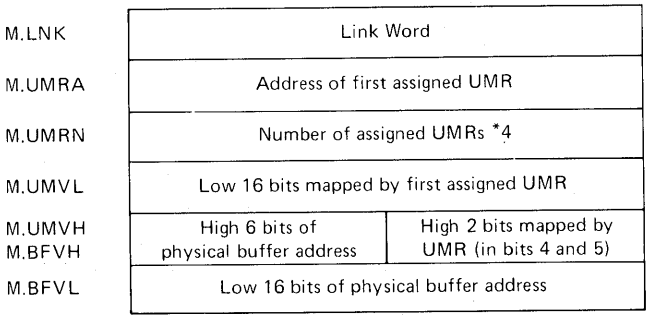
7.3.2 Calling $ASUMR and $DEUMR

Use of the procedure described in Section 7.3.3 ensures that UMRs are always allocated. However, a driver may not require UMRs to be allocated all of the time, and yet require UMRs for periods of time longer than the normal time-frame between $GTPKT and $IODON (or $IOALT). In such cases, there is a third procedure for allocating UMRs.

By using the Executive routines $ASUMR and $DEUMR, a driver can dynamically allocate, retain over a desired time-frame, and deallocate URMs. Refer to Section 7.4 for descriptions of the $ASUMR and $DEUMR routines.

Similar to the $STMAP/$MPUBM procedure, the use of $ASUMR and $DEUMR also requires the allocation of a 6-word mapping register assignment block. In this instance, however, the block must not be located in the 22-bit working storage area. $IODON or $IOALT, when called, will attempt to deallocate the UMRs of a block found in the 22-bit working storage area. To avoid this deallocation, the mapping register assignment block can be dynamically allocated from the pool. Figure 7–1 details the format of the 6-word block.

Figure 7–1: Mapping Register Assignment Block



ZK-276-81

7.3.3 Statically Allocating UMRs During System Generation

UMRs can be statically assigned during system generation. The system generation procedure defines the symbol N$$UMR equal to a fixed number of UMRs, multiplied by 4, that are statically assigned to the system. Before assembling the Executive, you can cause the static allocation of an additional number of UMRs by editing the Executive assembly prefix file RSXMC.MAC.

The value of the symbol N$$UMR can then be increased to represent the additional number of desired UMRs multiplied by 4.

The Executive assembly prefix file RSXMC.MAC further defines the following three symbols, which describe the first UMR statically allocated during system generation:

U$MRN The I/O page address of the first UMR register available for allocation to the user

U$$MLO The low-order 16 bits of the 18-bit UNIBUS address mapped by this UMR

U$$MHI The high-order two bits of the 18-bit UNIBUS address; these two bits are in bit positions 4 and 5

These three symbols are not used by the system itself. They are available for the user's information.

7.4 Service Calls

This section contains general commentary on the Executive routines typically used by I/O drivers. The descriptions of the routines are taken from the source code of modules linked to form the Executive. Table 7–1 summarizes the routines described in this section. Only the most widely used routines are described; however, many other Executive services are available. The source code for the related routines is in the MACRO-11 source files for the Executive modules.

Table 7–1: Summary of Executive Service Calls for Drivers

Routine Name	Location in UFD [11,10]	Function
$ACHKB	EXSUB	Address check for byte-aligned buffers
$ACHCK	EXSUB	Address check for word-aligned buffers
$ALOCB	CORAL	Allocate core buffer
$ASUMR	MEMAP	Assign UNIBUS mapping registers
$BLKCK	MDSUB	Check logical block number
$BLKC1	MDSUB	Check logical block number
$BLKC2	MDSUB	Check logical block number
$BLXIO	BFCTL	Move block of data
$CKBFI	EXESB	Check I/O buffer
$CKBFR	EXESB	Check I/O buffer
$CKBFW	EXESB	Check I/O buffer
$CKBFB	EXESB	Check I/O buffer
$CLINS	QUEUE	Clock queue insertion
$CVLBN	MDSUB	Convert logical block number

Table 7–1 (Cont.): Summary of Executive Service Calls for Drivers

Routine Name	Location in UFD [11,10]	Function
$DEACB	CORAL	Deallocate core buffer
$DEUMR	MEMAP	Deassign UNIBUS mapping registers
$DQUMR	MSCPDRV	Dequeue from UMR wait
$DVMSG	IOSUB	Device message output
$FORK	SYSXT	Create a fork process
$FORK1	SYSTXT	Fork but bypass clearing timeout count
$GTBYT	BFCTL	Get byte
$GTPKT	IOSUB	Get an I/O packet
$GSPKT	IOSUB	Get a special I/O packet
$GTWRD	BFCTL	Get word
$INIBF	IOSUB	Initiate I/O buffering
$INTSV	SYSXT	Interrupt save and restore
$INTXT	SYSXT	Interrupt exit
$IOALT	IOSUB	Alternate entry to $IODON
$IODON	IOSUB	I/O done for completing an I/O request
$IOFIN	IOSUB	I/O finish for special I/O completion
$MPUBM	MEMAP	Map UNIBUS memory
$MPUB1	MEMAP	Alternate $MPUBM entry for parallel operations
$PTBYT	BFCTL	Put byte
$PTWRD	BFCTL	Put word
$QINSP	QUEUE	Queue insertion by priority
$RELOC	MEMAP	Relocate address
$RELOP	MEMAP	Relocate UNIBUS physical address
$REQUE	IOSUB	Queue kernel AST to task
$REQU1	IOSUB	Queue kernel AST to task
$STMAP	MEMAP	Set up UNIBUS mapping address
$STMP1	MEMAP	Alternate $STMAP entry for parallel operations
$TSPAR	REQSB	Test if partition memory resident for kernel AST
$TSTBF	IOSUB	Test for I/O buffering

7.4.1 Address Check Routines ($ACHKB, $ACHCK)

The Address Check Byte Aligned ($ACHKB) and Address Check Word Aligned ($ACHCK) routines address check a block of memory to see whether it lies within the address space of the current task.

Format

```
CALL $ACHKB
CALL $ACHCK
```

Input

In Register 0:

Starting address of the block to be checked

In Register 1:

Length of the block to be checked, in bytes

Output

Condition Code

C = 1 If address check failed

C = 0 If address check succeeded

In Register 2:

Address of window block mapping buffer

Registers 0 and 3 preserved

Description

The Address Check routines are in the file IOSUB. A driver can call either routine to address check a task buffer while the task is the current task. The Address Check routines are normally used only by drivers setting UC.QUE in U.CTL. See Section 8.3 for an example.

Since privileged task I/O buffers are not address checked, Register 2 always returns a pointer to the first window block. Checkpointing and shuffling of commons will still work properly, provided that a privileged task never specifies an I/O into a common that it allows to remain checkpointable and shuffleable.

In RSX-11M-PLUS, almost all drivers will tend to use the alternate routines $CKBFB and $CKBFW. In addition to address checking the user buffer, $CKBFB and $CKBFW correctly maintain the attachment and partition I/O count mechanism. If the driver completes all references to the buffer in the initiation routine (that is, fills the buffer and calls $IOFIN, rather than queuing the packet or starting a transfer that is completed via interrupt service), then the driver can use $ACHKB or $ACHCK. See Section 7.4.6 for a description of $CKBFB and $CKBFW, and Section 8.3 for an example.

7.4.2 Allocate Core Buffer Routines ($ALOCB, $ALOC1)

The Allocate Core Buffer ($ALOCB) and the alternate entry Allocate Core Buffer ($ALOC1) routines are called to allocate an executive core buffer.

Format

```
CALL $ALOCB
CALL $ALOC1
```

Input

In Register 0:

One of the following:

- For the $ALOCB routine, no expected input in R0
- For the $ALOC1 routine, the address of core allocation listhead −2

In Register 1:

Size, in bytes, of the core buffer to allocate

Output

Condition Code

C = 1	If insufficient core is available to allocate the block
C = 0	If the block is allocated

In Register 0:

Address of the allocated block

In Register 1:

Length of the allocated block

Description

The Allocate buffer routines are in the file CORAL. The allocation algorithm is first fit, and blocks are allocated in multiples of four bytes.

The sample driver in Chapter 8 contains an instance of the use of $ALOCB.

7.4.3 Assign UNIBUS Mapping Registers Routine ($ASUMR)

The $ASUMR routine is called to assign a contiguous set of UMRs.

Format

```
CALL $ASUMR
```

Input

In Register 0:

Pointer to a mapping register assignment block

In M.UMRN(R0):

The number of UMRs required *4

In M.BFVL(RO):

The low address of transfer (for odd/even byte determination)

Output

Condition Code

C = 1 If the UMRs could not be assigned

C = 0 If the UMRs were successfully assigned

All registers preserved

Description

The $ASUMR routine is in the file MEMAP. It is used only for PDP-11/70 NPR devices requiring UNIBUS Mapping Registers when 22-bit memory addressing is enabled. Normally, it is not called directly by an I/O driver; it is called from within the $STMAP routine. Refer to Section 7.3 for a discussion.

For the sake of speed, the link word of each mapping assignment block points to the UMR address (second) word of the block, not the first word. The current state of UMR assignment is represented by a linked list of mapping assignment blocks, each block containing the address of the first UMR assigned and the number of UMRs assigned times 4. The blocks are linked in the order of increasing first UMR address.

7.4.4 Check Logical Block Routines ($BLKCK, $BLKC1, $BLKC2)

The following Check Logical Block routines are used by I/O device drivers to handle logical block numbers in data transfers:

- Logical Block Check ($BLKCK)
- Logical Block Check–Alternate Entry ($BLKC1)
- Logical Block Check–Alternate Entry for Queue Opt ($BLKC2)

Format

```
CALL $BLKCK
CALL $BLKC1
CALL $BLKC2
```

Input

In Register 1:
Address of the I/O packet

In Register 5:
Address of the UCB

Output

If the check fails:
$IODON is entered with a final status of "IE.BLK" and a return to the driver's initiator entry point is executed.

If the check succeeds:
The following registers are returned:

R0	=	Low part of logical block number
R1	=	Points to I.PRM +12 (low part of user LBN)
R2	=	High part of logical block number
R3	=	Address of I/O packet

Description

The $BLKCK routine is in the file MDSUB.

The $BLKCK routine is called by I/O device drivers to check the starting and ending logical block numbers of an I/O transfer to a file structured device. If the range of blocks is not legal, then $IODON is entered with a final status of "IE.BLK" and a return to the driver's initiator entry point is executed; otherwise a return to the driver is executed.

$BLKC2 returns to $QOPDN in $DRQRQ if there is an error instead of to the driver's initiator entry point. This allows the queue optimization code to use $BLKCK.

The output from these routines is used by disk drivers as input to the $CVLBN routine to handle logical block numbers in data transfers.

The sample driver in Chapter 8 contains an instance of the use of $BLKCK and $BLKC1.

7.4.5 Move Block of Data Routine ($BLXIO)

The $BLXIO routine is called to move memory data in a mapped system.

Format

```
CALL $BLXIO
```

Input

In Register 0:
The number of bytes to move

In Register 1:
The source APR 5 bias

In Register 2:
The source displacement

In Register 3:
The destination APR 6 bias

In Register 4:
The destination displacement

Note

The count input in Register 0 must not be zero, and it must not be large enough to cross APR boundaries (this typically means a maximum of 4 to 63K bytes).

Output

In the specified destination:
The block of data is moved.

If the move succeeds:
The following registers are returned:

R0	–	Contents destroyed
R1, R3	–	Preserved
R2, R4	–	Point to the last byte of source and destination +1

Description

The $BLXIO routine is in the file BFCTL.

The sample driver in Chapter 8 contains instances of the use of $BLXIO.

7.4.6 Check I/O Buffer Routines ($CKBFI, $CKBFR, $CKBFW, $CKBFB)

The following Check I/O Buffer routines are called to address check an I/O buffer associated with an I/O packet that is under construction:

- Check I/O buffer for I-space (overlay) access ($CKBFI)
- Check I/O buffer for read-only (byte) access ($CKBFR)
- Check I/O buffer for read-write (word) access ($CKBFW)
- Check I/O buffer for read-write (byte) access ($CKBFB)

Format

```
CALL $CKBFI
CALL $CKBFR
CALL $CKBFW
CALL $CKBFB
```

Input

In Register 0:
Starting address of block to be checked

In Register 1:
Length of buffer to be checked

In buffer $ATTPT:
Address of I.AADA in current I/O packet

In KISAR6:
The header of the subject task

Output

Condition Code

C = 1	If check is unsuccessful or packet could not be filled in
C = 0	If check and packet update are successful

If the check succeeds
The following occurs:

I.AADA or I.AADA +2 points to the ADB

A.IOC and P.IOC are incremented

Description

The Check I/O Buffer routines are in the file EXESB. These routines are called to address check an I/O buffer associated with the current (under construction) I/O packet. If the address check passes, then an attempt is made to point one of the attachment descriptor pointers at the associated ADB. This will have one of the following results:

- There is currently no attachment pointer in the packet to this ADB, and the pointers are not full. A pointer is filled in and the A.IOC, P.IOC fields for this I/O are incremented. This is the "normal" (successful) case.
- There is already one pointer to this ADB. The packet is untouched, as are the A.IOC and P.IOC fields, and the check is considered successful. The implication of not incrementing A.IOC and P.IOC is that drivers and ACPs may not release buffers for an I/O request one at a time; that is, the driver should not call $DECIO directly but should call $IODON or $DECAL after all buffer access has completed.

- There are already two pointers, none of them to this attachment descriptor. This is considered a check failure and return is made with carry set.

The sample driver in Chapter 8 contains an instance of the use of $CKBFR.

7.4.7 Clock Queue Insertion Routine ($CLINS)

The $CLINS routine is called to make an entry in the clock queue.

Format

```
CALL $CLINS
```

Input

In Register 0:
Address of the clock queue entry core block

In Register 1:
High-order half of Delta time

In Register 2:
Low-order half of Delta time

In Register 4:
Request type

In Register 5:
Address of requesting Task Control Block (TCB) or request identifier

Output

Result:
The clock queue entry is inserted in the clock queue according to the time that it will come due.

Description

The $CLINS routine is in the file QUEUE.

The entry is inserted such that the clock queue is ordered in ascending time; thus the front entries are most imminent and the back least imminent.

Note

On multiprocessor systems, a request with type C.SYST!100000 will be executed on a particular UNIBUS run, with URM specified in C.URM. Type C.CYST requests on MP systems are defaulted to run on any UNIBUS run, which in practice will result in the request executing on the CPU which owns the clock ($CKURM).

The sample driver in Chapter 8 contains an instance of the use of $CLINS.

7.4.8 Convert Logical Block Number Routine ($CVLBN)

The $CVLBN routine converts a logical block number to a physical disk address.

Format

```
CALL $CVLBN
```

Input (same as $BLKCK output)

In Register 0:
Low part of logical block number

In Register 1:
Points to I.PRM +12 (low part of user LBN)

In Register 2:
High part of logical block number

In Register 3:
Address of I/O packet

In Register 5:
The UCB address

Output

In Register 0:
The sector number

In Register 1:
The track number

In Register 2:
The cylinder number

Description

The $CVLBN routine is in the file MDSUB. The input to this routine is the same as the output from the $BLKCK routine. Typically, a disk driver calls this routine to convert a logical block number to a physical disk address. The routine accesses the U.PRM fields in the driver data base unit control block. These fields contain the sector, track, and cylinder parameters for the type of disk supported. Refer to the description of the U.PRM fields in Section 4.4.4.

The sample driver in Chapter 8 contains instances of the use of $CVLBN.

7.4.9 Deallocate Core Buffer Routines ($DEACB, $DEAC1)

The following routines are called to deallocate an Executive core buffer:

- Deallocate Core Buffer ($DEACB)
- Deallocate Core Buffer–Alternate Entry ($DEAC1)

Format

```
CALL $DEACB
CALL $DEAC1
```

Input

In Register 0:
Address of the core buffer to be deallocated

In Register 1:
Size (in bytes) of the core buffer to be deallocated

In Register 3:
One of the following:

- For the $DEACB routine, no expected input in R3
- For the $DEAC1 routine, the address of core allocation listhead −2

Output

Result:
The core block is merged into the free core chain at the core address.

Description

The $DEACB routine is in the file CORAL. The block is inserted into the free block chain by core address. If an adjacent block is currently free, then the two blocks are merged and inserted in the free block chain.

The sample driver in Chapter 8 contains instances of the use of $DEACB.

7.4.10 Deassign UNIBUS Mapping Registers Routine ($DEUMR)

The $DEUMR routine deassigns a contiguous block of UMRs.

Format

```
CALL $DEUMR
```

Input

In Register 2:
Pointer to the assignment block

Output

Registers 0 and 1 are preserved

Description

The $DEUMR routine is called to deassign a block of UMRs. $DEUMR is in the file MEMAP. It is used only for NPR devices requiring UNIBUS Mapping Registers when 22-bit addressing is enabled.

Normally this routine is not called directly by an I/O driver but from within the $IODON routine. When you call the $DEUMR routine directly, your driver must also contain a call to the $DQUMR routine before it returns to the system. If your driver contains a call to $DEUMR but none to $DQUMR (or to $IODON, which would call $DQUMR), the system may be left with drivers waiting for available UMRs. This can cause problems at a later date.

Refer to Section 7.3, Drivers for NPR Devices Using Extended Memory, for a discussion of assigning UMRs for I/O transfers.

7.4.11 Dequeue From UMR Wait Routine ($DQUMR)

The $DQUMR routine checks to see if a driver is waiting for UMR assignment.

Format

```
CALL $DQUMR
```

Input

In (SP)
The return address to the driver's caller

Output

Result:
The context of the waiting driver restored and the allocation routine called back

Description

The $DQUMR routine is in the file MEMAP.

First the calling driver is called back as a coroutine. When the calling driver issues a return back to the $DQUMR routine, $DQUMR checks to see if any drivers are waiting for UMRs. If so, the waiting driver's context is restored without actually dequeuing the mapping assignment block; control is passed back to the original UMR assignment routine.

7.4.12 Device Message Output Routine ($DVMSG)

The $DVMSG routine is called to submit a message to the task termination notification task.

Format

```
CALL $DVMSG
```

Input

In Register 0:
Message number

In Register 5:
Address of the UCB or TCB that the message applies to

Output

Result:
A four-word packet is allocated and threaded into the task termination notification task's message queue.

In the second word of the packet:
The contents of Register 0

In the third word of the packet:
The contents of Register 5

Description

The $DVMSG routine is in the file IOSUB.

The $DVMSG routine submits a message to the task termination notification task. A message is either device related or a checkpoint write failure from the loader. If the task termination notification task is not installed or no storage can be obtained, then the message request is ignored. $DVMSG can be set up and called as follows:

```
MOV   #T.NDNR,R0
 or
MOV   #T.NDSE,R0
CALL   $DVMSG
```

7.4.13 Fork Routine ($FORK)

The $FORK routine is called from an I/O driver to create a system process that will return to the driver at stack depth zero to finish processing.

Format

```
CALL $FORK
```

Input

In Register 5:
Address of the UCB for the unit being processed

In 0(SP):
Return address to the caller

7.4.16 Get Packet Routines ($GTPKT, $GSPKT)

The following Get Packet routines are called by device drivers to dequeue the next I/O request to process:

- Get I/O Packet From Request Queue routine ($GTPKT)
- Get Selective I/O Packet From Request Queue ($GSPKT)

Format

```
CALL $GTPKT
CALL $GSPKT
```

Input

In Register 2:
Address of driver's acceptance routine (if call is at $GSPKT)

In Register 5:
Address of the UCB of the controller for which the packet is called

Output

Condition Code

C = 1	If controller is busy or no request can be dequeued
C = 0	If a request was successfully dequeued

In Register 1:
Address of the I/O packet

In Register 2:
Physical unit number

In Register 3:
Controller index

In Register 4:
Address of the status control block

In Register 5:
Address of the unit control block

Registers destroyed:
Registers 4 and 5

Description

Get Packet and Get Special Packet are in the file IOSUB. The recommended way to use $GTPKT is to use the GTPKT$ macro call defined in Section 4.3. The use of $GSPKT is described briefly in Section 1.4.7.

If the device controller is busy when the driver calls $GTPKT, then a carry set indication is returned to the caller; otherwise an attempt is made to dequeue the next request from the controller queue. If no request can be dequeued, then a carry set indication is returned to the caller; otherwise the controller is set busy and a carry clear indication is returned to the caller.

If queue optimization is supported and enabled for the device, the appropriate packet for the current optimization algorithm is returned. Three algorithms are supported: nearest cylinder, elevator, and c scan. All three algorithms incorporate a fairness count. If the first packet on the list is passed over more than "fcount" times, it is done immediately.

The alternate entry point $GSPKT is intended for use by drivers that support parallel operations on a single unit, a common example being full duplex. Such drivers are expected to look to the system as if they are always free, while maintaining the status of all parallel operations internally within their own device data structures. Parallelism is accomplished by coordinating the handling of driver-defined classes of I/O function codes. For example, a full-duplex driver would handle input requests in parallel with output requests. A driver calls $GSPKT when it wants to dequeue a packet whose I/O function code belongs to a certain class. Which functions qualify is determined by an acceptance routine in the driver whose address is passed to $GSPKT in Register 2. The acceptance routine is called by $GSPKT each time a packet eligible to be dequeued is found in the queue. The acceptance routine is then expected to take one of the following three actions:

- Return with carry clear if the packet should be dequeued (in this case $GSPKT proceeds as $GTPKT normally would on dequeuing the packet)
- Return with carry set if the packet should not be dequeued (in this case $GSPKT will continue the scan of the I/O queue)
- Add the constant G$$SPSA to the stack pointer to abort the scan with no further action

The acceptance routine must save and restore any registers that it intends to modify. When a packet is dequeued via $GSPKT, the following normal $GTPKT actions do not occur:

- Filling in of U.BUF, U.BUF+2, and U.CNT (these fields are available for driver-specific use)
- Busying of UCB and SCB
- Execution of $CFORK to get to proper processor (multiprocessor systems)

$GSPKT may not be used by a driver that supports queue optimization

7.4.17 Get Word Routine ($GTWRD)

The $GTWRD routine is called to fetch the next word from the user buffer.

Format

```
CALL $GTWRD
```

Input

In Register 5:

Address of the UCB that contains the buffer pointers

Output

Result:

The next word is fetched from the user buffer

All registers preserved across call

Description

The $GTWRD routine is in the file BFCTL. It manipulates words U.BUF and U.BUF+2 in the UCB. $GTWRD fetches the next word and returns it to the caller on the stack. After the word has been fetched, the next word address is calculated.

7.4.18 Initiate I/O Buffering Routine ($INIBF)

Format

```
CALL $INIBF
```

The $INIBF routine is called to initiate I/O buffering.

Input

In Register 3:

Address of packet for I/O request

Output

Register 3 preserved

Description

The $INIBF routine is in the file IOSUB. $INIBF initiates I/O buffering using the following operations:

- Decrements the task's I/O count
- Increments the task's buffered I/O count
- Initiates checkpointing if a request is pending

The sample driver in Chapter 8 contains an instance of the use of $INIBF.

7.4.19 Interrupt Save Routines ($INTSV, $INTSE)

The following Interrupt Save routines are called from an interrupt service routine when an interrupt is not going to be immediately dismissed:

- Interrupt Save routine ($INTSV)
- Interrupt Save routine for error logging devices ($INTSE)

Format

```
CALL $INTSV,PRn
CALL $INTSE,PRn
```

The value for n has a range of 0 through 7.

Input

In 4(SP):
PSW pushed by interrupt

In 2(SP):
PC word pushed by interrupt

In 0(SP):
Saved Register 5 pushed by JSR R5,$INTSV instruction

In 0(R5):
New processor priority

Output

Result:
Interrupt service routine executes switch to system stack

In Register 4:
The controller index *2

New processor status is set

Description

The $INTSV routine is in the file SYSXT. The recommended way to use $INTSV is with the INTSV$ macro call described in Section 4.3.

The $INTSV routine is called from an interrupt service routine when an interrupt is not going to be immediately dismissed. A switch to the system stack is executed if the current stack depth is +1. When the interrupt service routine finishes its processing, it either forks, jumps to $INTXT, or executes a return.

Register 4 is pushed onto the current stack and the current stack depth is decremented. If the result is zero, then a switch to the system stack is executed. The new processor status is set and a coroutine call to the caller is executed. Register 4 is set with the controller index *2, which is determined from the PSW at entry.

Note

A system macro, INTSV$, is provided to simplify the coding of standard interrupt entry processing. See Section 4.3.

7.4.20 Interrupt Exit Routine ($INTXT)

The $INTXT routine is called to exit from an interrupt.

Format

```
JMP $INTXT
```

Input

In 0(SP):
Interrupt save return address

Output

Result:
A return to interrupt save is executed

Description

The $INTXT routine is in the file SYSXT. This routine may be called via a JMP instruction to exit from an interrupt.

7.4.21 I/O Done Routines ($IOALT, $IODON, $IODSA)

The following $I/O Done routines are called by device drivers at the completion of an I/O request to do final processing:

- I/O Done ($IODON)
- I/O Done–Alternate Entry–($IOALT)
- I/O Done–For DSA Drivers–($IODSA)

Format

```
CALL $IOALT
CALL $IODON
CALL $IODSA
```

Input

In Register 0:
First I/O status word

In Register 1:
One of the following:

If entry is at $IODON or $IODSA, R1 is the second I/O status word

- If entry is at $IOALT, then Register 1 is clear to signify that the second status word is zero

In Register 2:
Starting and final error retry counts if error logging device

In Register 5:
Address of the unit control block of the unit being completed

In (SP):
Return address to driver's caller

Output

The unit and controller are set idle

In Register 3:
Address of the current I/O packet

Contents of Register 4 are destroyed

Description

The $IODON, $IOALT, and $IODSA routines are in the file IOSUB.

The I/O Done routines are called by device drivers at the completion of an I/O request to do final processing. The unit and controller are set idle and $IOFIN is entered to finish the processing.

The $IODON and $IOALT routines push the address of routine $DQUMR onto the stack before returning to the driver. If your driver calls these routines, do not use the stack for temporary data storage.

If entry is at $IODSA, the following operations are not performed:

- UMR deallocation
- device busy status
- error logging finish code

The sample driver in Chapter 8 contains instances of the use of $IOALT and $IODON.

7.4.22 I/O Finish Routine ($IOFIN)

The $IOFIN routine is called to finish I/O processing.

Format

```
CALL $IOFIN
```

7.4.25 Put Byte Routine ($PTBYT)

Format

```
CALL $PTBYT
```

Input

In Register 5:
Address of the UCB that contains the buffer pointers

In 2(SP)
Byte to be stored in the next location of the user buffer

Output

Result:
The byte is stored in the user buffer and removed from the stack

All registers are preserved across the call

Description

The $PTBYT routine is in the file BFCTL and manipulates words U.BUF and U.BUF+2 in the UCB. It is called to put a byte in the next location in user buffer. The byte is stored in the user buffer and removed from the stack. The next byte address is then incremented.

7.4.26 Put Word Routine ($PTWRD)

The $PTWRD routine puts a word in the next available location in the user buffer.

Format

```
CALL $PTWRD
```

Input

In Register 5:
Address of the UCB that contains the buffer pointers

In 2(SP):
Word to be stored in the next location of the buffer

Output

Result:
The word is stored in the user buffer and removed from the stack

All registers are preserved across the call

Description

The $PTWRD routine is in the file BFCTL. It manipulates words U.BUF and U.BUF+2 in the UCB. After the word has been stored in the user buffer and removed from the stack, the next word address is calculated.

7.4.27 Queue Insertion by Priority Routine ($QINSP)

The $QINSP routine is called to insert an entry in a priority ordered list.

Format

```
CALL $QINSP
```

Input

In Register 0:
Address of the two-word listhead

In Register 1:
Address of the entry to be inserted

Output

Result:
The entry is linked into the list by priority

Registers 0 and 1 are preserved

Description

The $QINSP routine is in the file QUEUE. The Queue Insertion by Priority routine is used only by drivers setting UC.QUE in U.CTL. See Section 8.3 for an example.

A driver may call the $QINSP routine to insert into the I/O queue an I/O packet that the Executive has not already placed in the queue. The list is searched until an entry is found that has a lower priority or until the end of the list is reached. The new entry is then linked into the list at the appropriate point.

The sample driver in Chapter 8 contains an instance of the use of $QINSP.

7.4.28 Relocate Routine ($RELOC)

The $RELOC routine is called to transform a 16-bit user virtual address into a relocation bias and displacement in block relative to APR 6.

Format

```
CALL $RELOC
```

Input

In Register 0:
User virtual address to relocate

Output

In Register 1:
Relocation bias to be loaded into PAR6

In Register 2:
Displacement in block +140000 (PAR6 bias)

Registers 0 and 3 are preserved

Description

The $RELOC routine is in the file MEMAP. A driver may call $RELOC to relocate a task virtual address while the task is the current task. Relocate is normally used only by drivers setting UC.QUE in U.CTL. See Section 8.3 for an example.

The sample driver in Chapter 8 contains an instance of the use of $RELOC.

7.4.29 Relocate UNIBUS Physical Address Routine ($RELOP)

The $RELOP routine relocates a UNIBUS physical address to a KISAR6 bias and displacement.

Format

```
CALL $RELOP
```

Input

In Register 0:
Byte offset from address in U.BUF +1 and U.BUF +2

In Register 4:
SCB address

In Register 5:
UCB address

In U.BUF +1:
High-order bits of physical address

In U.BUF +2:
Low-order bits of physical address

Output

In KISAR6:

Calculated bias (mapped system)

In Register 1:

Real address or displacement

Description

The $RELOP routine is in the file MEMAP.

7.4.30 Queue Kernel AST to Task Routines ($REQUE, $REQU1)

The following routines are used to requeue a task kernel AST that has been previously used as a region load AST.

- Requeue a Region Load AST to a Task AST ($REQUE)
- Requeue a Region Load AST to a Task AST–Alternate Entry ($REQU1)

Format

```
CALL $REQUE
CALL $REQU1
```

Input

In Register 0:

TCB address of associated task

In Register 3:

Address of packet to be queued

Output

None

Description

The $REQUE routine is in module IOSUB. The buffered I/O count of the task is decremented if the entry is at $REQUE.

The sample driver in Chapter 8 contains instances of the use of $REQUE.

7.4.31 Set Up UNIBUS Mapping Address Routine ($STMAP)

The $STMAP routine is called by UNIBUS NPR device drivers to assign the UMRs and set up the UNIBUS mapping address.

Format

```
CALL $STMAP
```

Input

In Register 4:
Address of device SCB

In Register 5:
Address of device UCB

In (SP):
Return to driver's caller

Output

In the device UCB:
UNIBUS map addresses

In the SCB:
The actual physical address

Registers 1 through 3 are preserved across the call

Description

The $STMAP routine is in the file MEMAP. It is used only for NPR devices requiring UNIBUS Mapping Registers when 22-bit memory addressing is enabled. See Section 7.3 for a discussion.

The $STMAP routine is called by UNIBUS NPR device drivers to set up the UNIBUS mapping address, first assigning the UMRs. If the UMRs cannot be allocated, the driver's mapping assignment block is placed in a wait queue and a return to the driver's caller is executed. The assignment block will eventually be dequeued when the UMRs are available. The driver will be remapped and returned to with the contents of Registers 1 through 5 preserved and with the normal outputs of this routine. The driver's context is stored in the assignment block and fork block while it is blocked and in the wait queue. Once a driver's mapping assignment block is placed in the UMR wait queue, it is not removed from the queue until the UMRs are successfully assigned. This strategy ensures that waiting drivers will be serviced on a FIFO basis and that drivers with large requests for UMRs will not wait indefinitely.

The $STMAP routine pushes the address of routine $DQUMR+2 onto the stack before returning to the caller. If your driver calls this routine, do not use the stack for temporary data storage.

The sample driver in Chapter 8 contains an instance of the use of $STMAP.

7.4.32 Set Up UNIBUS Mapping Address Alternate Entry Routine ($STMP1)

Format

```
CALL $STMP1
```

Input

In Register 0:
Address of the alternate data structure (see Description)

In Register 4:
Address of device SCB

In Register 5:
Address of device UCB

Output

Result:
Data structure pointers set up for entry to $STMP2 in $STMAP

Description

The $STMP1 routine is in the file MEMAP. It is used only for NPR devices that require UNIBUS Mapping Registers when 22-bit memory addressing is enabled and for support parallel operations.

This entry code sets up an alternate data structure used as a UMR mapping assignment block and context storage block, in the same manner as $STMAP uses the fork block and mapping block in the SCB, KRB. The format of the structure is as follows:

1. Four words used for saving driver's context in case UMRs cannot be mapped immediately
2. Six words used as a UMR mapping assignment block

The $STMAP routine pushes the address of routine $DQUMR+2 onto the stack before returning to the caller. If your driver calls this routine, do not use the stack for temporary data storage.

7.4.33 Test if Partition Memory Resident for Kernel AST Routine ($TSPAR)

The $TSPAR routine tests whether a partition is in memory for a kernel AST.

Format

```
CALL $TSPAR
```

Input

In Register 0:
Address of packet being processed

In Register 1:
PCB address of region

In Register 5:
TCB address of associated task

Output

Condition Code

C = 0 If region is memory-resident

C = 1 If region is non-resident (In this case the region AST has been queued)

Description

The $TSPAR routine is in the file REQSB. It is called to check a region for memory residence to determine if it is safe to service a kernel AST (for example, to copy a buffer) into the region. If the region is checkpointed or currently being checkpointed, then a region load AST is queued and the region is accessed on the task's behalf.

The sample driver in Chapter 8 contains an instance of the use of $TSPAR.

7.4.34 Test for I/O Buffering Routine ($TSTBF)

The $TSTBF routine tests whether I/O buffering can be initiated.

Format

```
CALL $TSTBF
```

Input

In Register 3:

Address of packet for I/O request

Output

Condition Code

C = 0 If I/O buffering can be initiated

C = 1 If I/O buffering cannot be initiated

Register 3 preserved

Description

The $TSTBF routine is in the file IOSUB. It determines whether a given I/O request is eligible for I/O buffering; if so, it finds the Partition Control Block address (PCB) that indicates the region in memory where the transfer will occur. It stores this address in I.PRM+16 of the I/O packet.

The sample driver in Chapter 8 contains an instance of the use of $TSTBF.

Chapter 8
Sample Driver Code

This chapter presents three sections of code. Sections 8.1 and 8.2 show the driver data base and driver code for a conventional driver. Section 8.3 gives a coding example from a driver that inhibits the automatic packet queuing in QIO processing so that it might address-check and relocate a special user buffer. Both of the sample drivers are in directory [USER] on the distribution kit.

In addition to the examples shown in this chapter, you should review the source code for one or more, standard, DIGITAL-supplied drivers. You should also examine the files SYSTB.MAC and xxTAB.MAC (where xx is the device mnemonic), which contain data structures created at system generation.

8.1 Sample Driver Data Base

The following example shows the source code to create the data base for the driver that supports the DL device. The data base allows for one controller and one unit.

```
        .TITLE DLTAB
        .IDENT /09.0/
;
; SYSTEM TABLES
;
; MACRO LIBRARY CALLS
;
        .MCALL  CLKDF$
        .MCALL  HWDDF$
        .MCALL  SCBDF$
        .MCALL  UCBDF$

        CLKDF$                  ; DEFINE CLOCK BLOCK OFFSETS
        HWDDF$                  ; DEFINE HARDWARE REGISTERS
        SCBDF$                  ; DEFINE SCB OFFSETS
                                ; USERS OF THIS OR ANY CONTROL BLOCK SHOULD
                                ; NEVER SPECIFY THE SYSDEF ARGUMENT BUT SHOULD
                                ; ALLOW THE DEFINITION TO BE RESOLVED FROM
                                ; THE EXECUTIVE .STB FILE
        UCBDF$
```

```
;
;
;
$DLDAT::
;                                                   DL CTB
;
        .WORD   0               ; L.ICB
$CTBO:
        .WORD   $CTB1           ; L.LNK
        .ASCII  /DL/            ; L.NAM
        .WORD   .DCO            ; L.DCB
        .BYTE   1               ; L.NUM
        .BYTE   0               ; L.STS
$DLCTB::                        ; L.KRB
        .WORD   $DLA
;
;                                                   DL DCB
;
$DLTBL=0                        ;LOADABLE DLDRV
$DLDCB::
.DCO:
        .WORD   .DC1            ; D.LNK
        .WORD   .DCO            ; D.UCB
        .ASCII  /DL/            ; D.NAM
        .BYTE   0,0             ; D.UNIT
        .WORD   DLND-DLST       ; D.UCBL
        .WORD   $DLTBL          ; D.DSP
        .WORD   177477,70,0,177200,377,0,0,377 ; D.MSK
        .WORD   0               ; D.PCB

;
;                                                   DL UCB'S
;
;DLST=.
        .WORD   0
.DLO::
        .WORD   .DCO
        .WORD   .-2
        .BYTE   UC.ALG!UC.NPR!UC.PWF!1,US.MNT
        .BYTE   0,US.OFL
        .WORD   DV.DIR!DV.MSD!DV.UMD!DV.F11!DV.MNT
        .WORD   0
        .WORD   50000
        .WORD   512.
        .WORD   $DLO
        .WORD   0,0,0,0,0,0,0
        .BYTE   40.,2.
        .WORD   512.
```

```
DLND=.
;
;                                                    DLA  KRB
;
        .BYTE   PR5             ; K.PRI
        .BYTE   160/4           ; K.VCT
        .BYTE   0*2,0           ; K.CON, K.IOC
        .WORD   0!KS.OFL        ; K.STS
$DLA::  .WORD   174400          ; K.CSR
        .WORD   DLA-$DLA        ; K.OFF
        .BYTE   0,0             ; K.HPU
        .WORD   0               ; K.OWN
;
;       CONTIGUOUS   S C B  HERE FOR    DL
;
$DLO::  .WORD   0,.-2           ; S.LHD AND K.CRQ
        .WORD   0,0,0,0         ; S.FRK
        .WORD   0               ; S.KS5
        .WORD   0               ; S.PKT
        .BYTE   0               ; S.CTM
        .BYTE   4               ; S.ITM
        .BYTE   0               ; S.STS
        .BYTE   0               ; S.ST3
        .WORD   S2.LOG!S2.CON   ; S.ST2
        .WORD   $DLA            ; S.KRB
        .BYTE   7               ; S.RCNT
        .BYTE   0               ; S.ROFF
        .WORD   0               ; S.EMB
        .BLKW   6               ; MAPPING ASSIGNMENT BLOCK
        .WORD   0               ; KE.RHB
DLA:
;
;
$DLEND::

.DC1 = 0                        ; END OF DCB LIST FOR DL:

$CTB1 = 0                       ; END OF CTB LIST FOR DL:

        .END
```

8.2 Sample Driver Code

The following example shows the source code for the DL driver. Comments beginning with ';;;' indicate that the instruction is being executed at a priority level greater than or equal to 5.

```
        .TITLE  DLDRV
        .IDENT  /01/

;
; RL11-RL01/02 DISK DRIVER
;

.MCALL  HWDDF$,PKTDF$
        HWDDF$                  ;DEFINE HARDWARE REGISTERS
        PKTDF$                  ;DEFINE I/O PACKET OFFSETS

;
; EQUATED SYMBOLS
;
```

```
RETRY=  8.                      ;CONTROLLER ERROR RETRY COUNT
RLBPT=  512.*20.                ;BYTES PER SURFACE
RLSPU=  15.                     ;TIME TO SPIN UP

;
; RL11 DEVICE REGISTER OFFSETS
;
RLCS=   0                       ;CONTROL STATUS REGISTER
RLBA=   2                       ;BUS ADDRESS REGISTER
RLDA=   4                       ;DISK ADDRESS REGISTER
RLMP=   6                       ;MULTIPURPOSE REGISTER

;
; RLCS BIT ASSIGNMENTS
;
ERR=    100000                  ;COMPOSITE ERROR
DE=     040000                  ;DRIVE ERROR
NXM=    020000                  ;NONEXISTENT MEMORY
DLT=    010000                  ;DATA LATE
HNF=    010000                  ;HEADER NOT FOUND
DCK=    004000                  ;DATA CHECK ERROR
HCRC=   004000                  ;HEADER CRC ERROR
OPI=    002000                  ;OPERATION INCOMPLETE
DRDY=   1                       ;DRIVE READY
WCHK=   2                       ;WRITE CHECK FUNCTION
WRITE=  2                       ;WRITE OFFSET
GSTS=   4                       ;GET DRIVE STATUS FUNCTION
SEEK=   6                       ;SEEK FUNCTION
RDH=    10                      ;READ HEADERS FUNCTION
READ=   14                      ;READ DATA FUNCTION
IE=     100                     ;INTERRUPT ENABLE
CRDY=   200                     ;CONTROLLER READY

;
; RLDA STATUS CODES
;
MRK=    1                       ;MARKER BIT
STS=    2                       ;GET STATUS BIT
SN=     4                       ;SIGN BIT FOR SEEK
RST=    10                      ;DRIVE RESET BIT
HS=     20                      ;HEAD SELECT BIT FOR DIFFERENCE
REV=    200!MRK                 ;REVERSE SEEK DIFFERENCE WORD

;
; RLMP GET STATUS BIT ASSIGNMENTS
;
```

```
WDE=    100000                  ;WRITE DATA ERROR
CHE=    040000                  ;CURRENT HEAD ERROR
WLS=    020000                  ;WRITE LOCK STATUS
SKTO=   010000                  ;SEEK TIMEOUT ERROR
SPD=    004000                  ;SPEED ERROR
WGE=    002000                  ;WRITE GATE ERROR
VC=     001000                  ;VOLUME CHECK
DSE=    000400                  ;DRIVE SELECT ERROR
DT=     000200                  ;DRIVE TYPE
HSS=    000100                  ;HEAD SELECT STATUS
CO=     000040                  ;COVER OPEN
HH=     000020                  ;HEADS HOME
BH=     000010                  ;BRUSHES HOME
SLM=    000005                  ;DRIVE IN SEEK-LINEAR MODE STATE

;
; LOCAL DATA
;
; CONTROLLER IMPURE DATA TABLES
; THESE ARE INDEXED BY THE CONTROLLER NUMBER
;
RTTBL:  .BLKW  R$$L11           ;RETRY COUNT FOR CURRENT OPERATION
PRMSV:  .BLKW  R$L11*5          ;PARAMETER SAVE AREA FOR WRITE CHECK

;
;DRIVER DISPATCH TABLE
;
        DDT$   DL,R$$L1,NEW=Y   ;GENERATE DISPATCH TABLE

;+
; **-DLINI-RL11-RL01/02 DISK CONTROLLER INITIATOR
;
; THIS ROUTINE IS ENTERED FROM THE QUEUE I/O DIRECTIVE WHEN AN I/O
; REQUEST IS QUEUED AND AT THE END OF A PREVIOUS I/O OPERATION TO
; PROPAGATE THE EXECUTION OF THE DRIVER.  IF THE SPECIFIED CONTROLLER
; IS NOT BUSY, THEN AN ATTEMPT IS MADE TO DEQUEUE THE NEXT I/O REQUEST.
; ELSE A RETURN TO THE CALLER IS EXECUTED.  IF THE DEQUEUE ATTEMPT
; IS SUCCESSFUL, THEN THE NEXT I/O OPERATION IS INITIATED.  A RETURN
; TO THE CALLER IS THEN EXECUTED.
;
; INPUTS:
;       R5=ADDRESS OF THE UCB OF THE CONTROLLER TO BE INITIATED.
;
; OUTPUTS:
;       IF THE SPECIFIED CONTROLLER IS NOT BUSY AND AN I/O REQUEST IS
;       WAITING TO BE PROCESSED, THEN THE REQUEST IS DEQUEUED AND THE
;       DRIVER INITIATES THE REQUESTED I/O FUNCTION
;-

DLINI:  GTPKT$  DL,R$$L11       ;GET NEXT I/O PACKET TO PROCESS
```

```
;
; THE FOLLOWING ARGUMENTS ARE RETURNED BY $GTPKT:
;
;       R1=ADDRESS OF THE I/O REQUEST PACKET
;       R2=PHYSICAL UNIT NUMBER OF THE REQUESTED DRIVE
;       R3=CONTROLLER INDEX
;       R4=ADDRESS OF THE STATUS CONTROL BLOCK
;       R5=ADDRESS OF THE UCB OF THE DRIVE TO BE INITIATED
;
; RL11-RL01/02 DISK CONTROLLER I/O REQUEST PACKET FORMAT:
;
;       WD. 00 -- I/O QUEUE THREAD WORD
;       WD. 01 -- REQUEST PRIORITY, EVENT FLAG NUMBER
;       WD. 02 -- ADDRESS OF THE TCB OF THE REQUESTOR TASK
;       WD. 03 -- POINTER TO 2ND LUN WORD IN REQUESTOR TASK HEADER
;       WD. 04 -- CONTENTS OF FIRST LUN WORD
;       WD. 05 -- I/O FUNCTION CODE
;       WD. 06 -- VIRTUAL ADDRESS OF I/O STATUS BLOCK
;       WD. 07 -- RELOCATION BIAS OF I/O STATUS BLOCK
;       WD. 10 -- I/O STATUS BLOCK ADDRESS (DISPLACEMENT + 140000)
;       WD. 11 -- VIRTUAL ADDRESS OF AST SERVICE ROUTINE
;       WD. 12 -- MEMORY EXTENSION BITS OF I/O TRANSFER
;       WD. 13 -- BUFFER ADDRESS OF I/O TRANSFER
;       WD. 14 -- NUMBER OF BYTES TO BE TRANSFERRED
;       WD. 15 -- NOT USED.
;       WD. 16 -- LOW BYTE MUST BE ZERO AND HIGH BYTE IS NOT USED
;       WD. 17 -- LOW PART OF LOGICAL BLOCK NUMBER OF I/O REQUEST
;       WD. 20 -- RELOCATION BIAS OF REGISTER BUFFER ELSE NOT USED
;       WD. 21 -- REGISTER BUFFER ADDRESS (DISPLACEMENT + 140000) ELSE NOT USED
;
; DRIVER USE OF WORDS IN I/O PACKET:
;       I.PRM+6  (WD. 15) - SEEK DIFFERENCE WORD
;       I.PRM+10 (WD. 16) - STARTING DISK ADDRESS FOR THIS TRANSFER
;       I.PRM+12 (WD. 17) - BYTE COUNT FOR THIS TRANSFER
;
        MOV     #RETRY&377,RTTBL(R3)  ;SET INITIAL RETRY COUNT
        CALL    $VOLVD                ;VALIDATE VOLUME VALID
        BCS     5$                    ;IF CS WE FAILED
        TST     R0                    ;TRANSFER FUNCTION?
        BMI     10$                   ;IF MI YES
        TST     I.PRM+2(R1)           ;SIZE THE DISK?
        BPL     5$                    ;IF PL NO, ERROR
        MOV     S.CSR(R4),R2          ;RETRIEVE CSR ADDRESS
        CALL    DLRST                 ;RESET DRIVE AND GET STATUS
        MOV     S.PKT(R4),R3          ;RETRIEVE I/O PACKET ADDRESS
        MOV     I.PRM+14(R3),KISAR6   ;SET BUFFER RELOCATION BIAS
        MOV                         CALL  ;MOVE REGISTERS INTO BUFFER
5$:     JMP     DLFIN                 ;FINISH UP
```

```
10$:    CALL    $STMAP          ;SET UP UNIBUS MAPPING ADDRESS
        MOVB    R2,U.BUF+1(R5)  ;SET CURRENT UNIT NUMBER
        MOV     #IE.IFC&377,R0  ;ASSUME ILLEGAL FUNCTION
        BIS     #READ!IE,U.BUF(R5)  ;ASSUME READ LOGICAL FUNCTION
        CMPB    #IO.RLB/256.,I.FCN+1(R1)  ;REALLY?
        BEQ     20$             ;IF EQ YES
        CMPB    #IO.WLB/256.,I.FCN+1(R1)  ;WRITE LOGICAL FUNCTION
        BNE     5$              ;IF NE NO, EXIT WITH ERROR
        SUB     #WRITE,U.BUF(R5)  ;CONVERT TO WRITE LOGICAL FUNCTION
20$:    MOV     #RETRY,RTTBL(R3)  ;SET INITIAL RETRY COUNT
        MOV     I.PRM+12(R1),R0 ;RETRIEVE BLOCK NUMBER
        CLR     R2              ;CLEAR HIGH ORDER BLOCK NUMBER
        BITB    $IO.WPB&377,I.FCN(R1)  ;PHYSICAL BLOCK FUNCTION?
        BNE     35$             ;IF NE YES
        CALL    $BLKCK          ;CHECK LOGICAL BLOCK NUMBER
        CMPB    #IO.WLB/256.,I.FCN+1(R3)  ;WRITE FUNCTION?
        BNE     30$             ;IF NE NO
        BITB    #IO.WLT&377,I.FCN(R3)  ;OK TO WRITE ON LAST TRACK?
        BNE     30$             ;IF NE YES
        MOV     R0,I.PRM+6(R3)  ;YES, SAVE STARTING BLOCK NUMBER
        ADD     #^D<20>,I.PRM+12(R3)  ;ADD 1 TRACK'S WORTH OF BLOCKS
        CALL    $BLKC1          ;CHECK IF WRITE ON LAST TRACK OF DISK
        MOV     I.PRM+6(R3),R0  ;RESTORE ORIGINAL STARTING BLOCK NUMBER

30$:    ASL     R0              ;CONVERT BLOCKS TO SECTORS
35$:    MOV     S.PKT(R4),R3    ;RESET I/O PACKET ADDRESS
        CALL    $CVLBN          ;CONVERT BLOCK NUMBER TO DISK ADDRESS
        ROR     R1              ;PUT SURFACE BIT IN CARRY
        ROL     R2              ;MERGE IT WITH THE CYLINDER NUMBER
        ASH     #6,R2           ;POSITION CYLINDER AND SURFACE
        BIS     R0,R2           ;MERGE SECTOR WITH CYLINDER AND SURFACE
        MOV     R2,I.PRM+10(R3) ;SAVE STARTING DISK ADDRESS
        MOV     U.CNT(R5),I.PRM+12(R3)  ;ASSUME ONLY ONE XFER NEEDED
        MOV     #^D<40>,R1      ;SET SECTORS/SURFACE
        SUB     R0,R1           ;CALCULATE SECTORS LEFT ON SURFACE
        SWAB    R1              ;GET BYTES LEFT ON SURFACE
        CMP     U.CNT(R5),R1    ;ARE ADDITIONAL TRANSFERS REQUIRED?
        BLOS    40$             ;IF LOS NO
        MOV     R1,I.PRM+12(R3) ;SET BYTE COUNT FOR FIRST TRANSFER
40$:    MOVB    S.CON(R4),R1    ;RETRIEVE CONTROLLER INDEX
        MUL     #5,R1           ;FORM INDEX INTO PARAMETER SAVE AREA
        ADD     $PRMSV,R1       ;POINT TO THIS ENTRY
        MOV     U.BUF(R5),(R1)+ ;SAVE INITIAL PARAMETERS
        MOV     U.BUF+2(R5),(R1)+  ;...
        MOV     U.CNT(R5),(R1)+ ;...
        MOV     I.PRM+10(R3),(R1)+  ;...
        MOV     I.PRM+12(R3),(R1)+  ;...

;+
; THIS SECTION WILL INITIATE THE OPERATION
;-
```

```
DLINIO: CALL    $MPUBM              ;MAP UNIBUS TO TRANSFER
        MOV     S.CSR(R4),R2        ;GET ADDRESS OF CSR
        MOV     S.PKT(R4),R3        ;GET ADDRESS OF I/O PACKET
        CLRB    U.CW2+1(R5)         ;RESET DRIVE SETTLE DOWN FLAG
        CLR     I.PRM+6(R3)         ;RESET ERROR DIFFERENCE WORD
        MOVB    S.ITM(R4),S.CTM(R4)  ;SET DEVICE TIMEOUT COUNTER
        CALL    DLRST               ;RESET DRIVE AND GET STATUS
        MOV     RLMP(R2),R1         ;GET THE STATUS INFO
        BIC     #WLS!DT!HSS,R1      ;REMOVE IRRELEVANT BITS
        BIT     #DRDY,(R2)          ;IS THE DRIVE READY?
        BEQ     10$                 ;IF EQ NO
        CMP     #HH!BH!SLM,R1       ;HEADS, BRUSHES AND STATE OK?
        BEQ     20$                 ;IF EQ YES

10$:    BITB    #US.SPU,U.STS(R5)   ;IS DRIVE SPINNING UP?
        BNE     DLPWF1              ;IF NE YES, WAIT FOR IT TO SPIN UP
        MOV     #IE.DNR&377,R0      ;SET RETURN ERROR CODE
        JMP     DLFIN               ;EXIT WITH FATAL ERROR
20$:    BICB    #US.SPU,U.STS(R5)   ;RESET DRIVE SPINNING UP
        MOV     I.PRM+10(R3),R0     ;RETRIEVE STARTING DISK ADDRESS
        CALL    DLDIFF              ;CALCULATE DIFFERENCE WORD
DLGO:   BEQ     30$                 ;IF EQ NO SEEK IS NECESSARY
        MOV     #SEEK,R1            ;GET CODE FOR SEEK FUNCTION
        CALL    DLXCT               ;EXECUTE THE SEEK
        BMI     DLEROR              ;IF MI ERROR DURING SEEK FUNCTION

30$:    ADD     #RLMP,R2            ;POINT TO RLMP
        MOV     I.PRM+12(R3),R1     ;GET BYTE COUNT
        ROR     R1                  ;MAKE IT A WORD COUNT
        NEG     R1                  ;ALSO NEGATIVE
        MOV     R1,(R2)             ;LOAD WORD COUNT
        MOV     I.PRM+10(R3),-(R2)   ;LOAD STARTING DISK ADDRESS
        MOV     U.BUF+2(R5),-(R2)    ;LOAD BUS ADDRESS
        CALL    $BMSET              ;SET I/O ACTIVE BIT IN MAP
        MOV     U.BUF(R5),-(R2)     ;;;LOAD FUNCTION AND GO

;+
; CANCEL I/O OPERATION IS A NOP FOR FILE STRUCTURED DEVICES.
;-

DLCAN:  RETURN                      ;;;NOP FOR RL11

;+
; POWERFAIL IS HANDLED VIA THE DEVICE TIMEOUT FACILITY AND
; CAUSES NO IMMEDIATE ACTION ON THE UNIT.  THE CURRENT TIMEOUT
; COUNT IS EXTENDED, THUS IF A UNIT WAS BUSY IT WILL HAVE
; SUFFICIENT TIME TO SPIN BACK UP.  THE NEXT I/O REQUEST TO ANY
; UNIT WILL BE SUSPENDED FOR AT LEAST THE EXTENDED TIMEOUT UNLESS
; THE UNIT IS CURRENTLY READY.
;-

DLPWF:  TSTB    S.STS(R4)           ;IS DRIVE CURRENTLY BUSY?
        BEQ     DLPWF2              ;IF EQ NO
        MOVB    #4,2.STS(R4)        ;ALLOW FOR A FULL MINUTE TO SPIN UP
DLPWF1: MOVB    #RLSPU,S.CTM(R4)    ;EXTEND TIMEOUT IN CASE UNIT WAS BUSY
DLPWF2: BISB    #US.SPU,U.STS(R5)   ;SET UNIT SPINNING UP
        RETURN                      ;
```

```
;+
; **-$DLINT-RL11-RL01/02 DISK CONTROLLER
;       INTERRUPT AND ERROR SERVICE ROUTINES
;
;-
        .ENABL  LSB

$DLINT::INTSV$  DL,PR5,R$$L11   ;;;SAVE REGISTERS AND SET PRIORITY
        CALL    $FORK           ;;;CREATE A SYSTEM PROCESS
        MOV     R4,R3           ;COPY CONTROLLER INDEX
        ASRB    RTTBL+1(R3)      ;HOME SEEK IN PROGRESS?
        BCS     DLINIO          ;IF CS YES
        MOV     U.SCB(R5),R4    ;GET ADDRESS OF SCB
        MOV     S.CSR(R4),R2    ;GET ADDRESS OF CSR
        MOV     #IS.SUC&377,R0  ;ASSUME SUCCESSFUL OPERATION
        MOV     S.PKT(R4),R3    ;RETRIEVE I/O PACKET ADDRESS
        MOV     (R2),R1         ;GET CONTENTS OF RLCS
        BMI     20$             ;IF MI AN ERROR OCCURRED
        SUB     I.PRM+12(R3),U.CNT(R5)  ;CALCULATE BYTES LEFT TO XFER
        BEQ     70$             ;IF EQ NONE LEFT
        MOV     U.CNT(R5),I.PRM+12(R3)  ;ASSUME LAST XFER COMING
        CMP     U.CNT(R5),#RLBPT  ;IS THIS THE LAST TRANSFER?
        BLOS    10$             ;IF LOS YES
        MOV     #RLBPT,I.PRM+12(R3)  ;TRANSFER A WHOLE TRACKS WORTH
10$:    BIC     #CRDY,R1        ;CLEAR CRDY TO START FUNCTION
        MOV     R1,U.BUF(R5)    ;SAVE CURRENT FUNCTION AND ADDRESS BITS
        MOV     RLBA(R2),U.BUF+2(R5)  ;SAVE CURRENT BUS ADDRESS
        MOV     I.PRM+10(R3),R0 ;GET INITIAL DISK ADDRESS
        MOV     R0,R1           ;COPY DISK ADDRESS
        BIS     #77,R0          ;UPDATE CYLINDER AND SURFACE ...
        INC     R0              ;... LEAVING SECTOR BITS ZERO
        MOV     R0,I.PRM+10(R3) ;SAVE NEW DISK ADDRESS
        CALL    DLDIFO          ;CALCULATE MID-TRANSFER DIFFERENCE
        JMP     DLGO            ;GO DO THE OPERATION

20$:    BIT     #DRDY,R1        ;IS THE DRIVE READY?
        BNE     DLEROR          ;IF NE YES, GO CHECK FOR ERRORS
25$:    MOVB    #3,S.CTM(R4)    ;WAIT 3 SECONDS FOR THE DRIVE TO SETTLE
        INCB    U.CW2+1(R5)     ;FLAG SETTLE DOWN IN PROGRESS
        RETURN
```

```
DLEROR: MOV     (R2),R1          ;RETRIEVE CONTENTS OF RLCS
        MOV     #IE.VER&377,R0   ;ASSUME UNRECOVERABLE ERROR
        BIT     #NXM,R1          ;NON-EXISTENT MEMORY?
        BNE     90$              ;IF NE YES
        BIT     #DE,R1           ;DRIVE PROBLEMS?
        BEQ     40$              ;IF EQ NO
        CALL    DLGST            ;EXECUTE GET DRIVE STATUS FUNCTION
        BIT     #WGE,RLMP(R2)    ;WRITE GATE ERROR?
        BEQ     90$              ;IF EQ NO
        BIT     #WLS,RLMP(R2)    ;IS THE DRIVE WRITE LOCKED?
        BEQ     DLRTRY           ;IF EQ NO
        MOV     #IE.WKK&377,R0   ;SET WRITE LOCK ERROR CODE
        BR      DLFIN            ;
40$:    BIT     #10,U.BUF(R5)    ;WRITE CHECK FUNCTION?
        BNE     DLRTRY           ;IF NE NO
        BIT     #OPI,R1          ;OPERATION INCOMPLETE?
        BNE     DLRTRY           ;IF NE YES
        BIT     #DCK,R1          ;WRITE CHECK ERROR
        BEQ     DLRTRY           ;IF EQ NO
        MOV     #IE.WCK&377,R0   ;YES, SET WRITE CHECK ERROR CODE
        BR      DLRTRY           ;GO RETRY OPERATION IF REQUIRED

70$:    BITB    #IO.WLC&377,I.FCN(R3)  ;WRITE WITH WRITE CHECK?
        BNE     80$              ;IF NE YES
        BITB    #US.WCK,U.STS(R5)  ;WRITE CHECK ENABLED?
        BEQ     DLFIN            ;IF EQ NO
80$:    MOV     U.BUF(R5),R1     ;GET CURRENT FUNCTION CODE
        BIT     #WCHK,R1         ;WRITE OR WRITE CHECK FUNCTION?
        BEQ     DLFIN            ;IF EQ NO
        BIT     #10,R1           ;WAS FUNCTION WRITE CHECK?
        BEQ     DLFIN            ;IF EQ YES
        MOVB    S.CON(R4),R1     ;RETRIEVE CONTROLLER INDEX
        MOV     #RETRY,RTTBL(R1);RESET RETRY COUNT
        MUL     #5,R1            ;FORM AN INDEX INTO SAVE AREA
        ADD     #PRMSV,R1        ;...
        MOV     (R1)+,U.BUF(R5) ;RESTORE STARTING PARAMETERS
        MOV     (R1)+,U.BUF+2(R5)  ;...
        MOV     (R1)+,U.CNT(R5) ;...
        MOV     (R1)+,I.PRM+10(R3)  ;...
        MOV     (R1)+,I.PRM+12(R3)  ;...
        BIC     #10,U.BUF(R5)    ;CONVERT TO WRITE CHECK FUNCTION
        JMP     DLINIO           ;START THE WRITE CHECK

;+
; FINISH I/O OPERATION
;-

90$:    MOV     #IE.VER&377,R0   ;SET UNSUCCESSFUL OPERATION
DLFIN:  MOV     S.PKT(R4),R2     ;GET ADDRESS OF I/O PACKET
        MOV     I.PRM+4(R2),R1   ;GET TOTAL TRANSFER SIZE
        SUB     U.CNT(R5),R1     ;CALCULATE BYTES TRANSFERRED
        MOVB    S.CON(R4),R3     ;RETRIEVE CONTROLLER INDEX
        MOVB    RTTBL(R3),R2     ;GET FINAL RETRY COUNT
        BIS     #RETRY*^D<256>,R2  ;MERGE STARTING RETRY COUNT
        CALL    $IODON           ;FINISH I/O OPERATION
        JMP     DLINI            ;PROCESS NEXT REQUEST
```

```
;+
; **-DLOUT-RL11-RL01/02 DISK CONTROLLER
;       DEVICE TIMEOUT ROUTINE
;
; DEVICE TIMEOUT RESULTS IN THE OPERATION BEING REPEATED.
; TIMEOUTS ARE USUALLY CAUSED BY A POWER FAILURE BUT MAY ALSO
; BE THE RESULT OF A HARDWARE MALFUNCTION.
;-

DLOUT:  MOV     S.PKT(R4),R3      ;;;RETRIEVE I/O PACKET ADDRESS
        BITB    #US.SPU,U.STS(R5)  ;;;IS DRIVE SPINNING UP?
        BEQ     20$               ;;;IF EQ NO
        DECB    S.STS(R4)         ;;;HAVE WE WAITED A MINUTE YET?
        BNE     10$               ;IF NE NO
        INCB    S.STS(R4)         ;;;LEAVE CONTROLLER BUSY
        BR      30$               ;;;LOG DEVICE TIMEOUT
10$:    MTPS    #0                ;;;ALLOW INTERRUPTS
        JMP     DLINIO            ;RETRY ENTIRE OPERATION
20$:    TSTB    U.CW2+1(R5)       ;;;IS DRIVE SETTLING DOWN?
        BEQ     30$               ;;;IF EQ NO
        MTPS    #0                ;;;YES, ALLOW INTERRUPTS
        JMP     DLEROR            ;PROCESS THE ERROR
30$:    MTPS    #0                ;;;ALLOW INTERRUPTS
        CALL    DLRST             ;RESET DRIVE
        MOV     #IE.DNR&377,R0    ;SET DEVICE NOT READY
DLRTRY: MOV     S.PKT(R4),R1      ;GET I/O PACKET ADDRESS
        BITB    #IQ.X,I.FCN(R1)   ;INHIBIT RETRIES?
        BNE     DLFIN             ;IF NE YES
        DECB    RTTBL(R3)         ;ANY MORE RETRIES LEFT?
        BLE     DLFIN             ;IF LE NO
        JMP     DLINIO            ;YES, RETRY ENTIRE OPERATION

;+
; **-DLXCT,DLGST,DLRST-RL11-RL01/02 DISK CONTROLLER
;       FUNCTION EXECUTION ROUTINES
;
; THIS ROUTINE WILL EXECUTE A GET DRIVE STATUS OR ANY
; NON-INTERRUPTABLE FUNCTION AND WAIT FOR ITS COMPLETION.
;
; INPUTS:
;       R1 = FUNCTION CODE
;       R2 = CSR ADDRESS
;       R5 = UCB ADDRESS
;
; OUTPUTS:
;       R1 = CONTENTS OF RLCS (TESTED)
;       FUNCTION EXECUTED
;-
```

```
        .ENABL  LSB
DLRST:  MOV     #RST!STS!MRK,RLDA(R2)  ;SET MESSAGE CODES IN RLDA
        CALL    10$             ;DO THE DRIVE RESET FIRST
DLGST:  MOV     #STS!MRK,RLDA(R2)  ;SET MESSAGE CODES IN RLDA
10$:    MOV     #GSTS,R1        ;SET GET STATUS FUNCTION
DLXCT:  MOV     R1,-(SP)        ;SAVE FUNCTION CODE
        MOVB    U.UNIT(R5),1(SP);MERGE CURRENT DRIVE BITS
        MOV     (SP)+,(R2)      ;LOAD RLCS
20$:    BIT     #ERR!CRDY,(R2)  ;READY OR ERROR?
        BEQ     20$             ;IF EQ NEITHER
        MOV     (R2),R1         ;SAVE RLCS AND TEST FOR ERRORS
        RETURN
        .DSABL  LSB

;+
; **-DLDIFF-RL11-RL01/02 DISK CONTROLLER
;       CYLINDER ADDRESS DIFFERENCE CALCULATOR
;
; THIS SUBROUTINE CALCULATES THE DIFFERENCE WORD USED IN THE
; SEEK OPERATION.  IF A HEADER CANNOT BE READ AFTER 16. RETRIES,
; AN ERROR WILL BE LOGGED AND A ONE CYLINDER REVERSE SEEK WILL BE
; ISSUED.  THE SEEK IS FOLLOWED BY A READ HEADERS TO CAUSE AN
; INTERRUPT.
;
; INPUTS:
;       R0 = DESIRED DISK ADDRESS
;       R3 = I/O PACKET ADDRESS
;
; OUTPUTS:
;       R1 = DIFFERENCE WORD
;       RLDA = LOADED WITH DIFFERENCE WORD
;       I.PRM+6 = LOADED WITH DIFFERENCE WORD
;       IF EQ NO SEEK IS NECESSARY
;-
```

```
DLDIFF: MOV     #RETRY*2,-(SP)  ;SET READ HEADER RETRY COUNT
10$:    MOV     #RDH,R1         ;SET CODE FOR READ HEADERS FUNCTION
        CALL    DLXCT           ;EXECUTE THE FUNCTION
        BPL     20$             ;IF PL FUNCTION EXECUTED OK
        DEC     (SP)            ;ANY RETRIES LEFT?
        BGT     10$             ;IF GT YES
        CMP     (SP)+,(SP)+     ;REMOVE RETRY COUNT AND CALLERS ADDRESS
        CALL    DLRST           ;RESET DRIVE
        MOV     #REV,RLDA(R2)   ;LOAD REVERSE SEEK DIFFERENCE WORD
        MOV     #SEEK,R1        ;GET CODE FOR SEEK FUNCTION
        MOV     #IE.VER&377,R0  ;ASSUME WE WILL FAIL
        CALL    DLXCT           ;EXECUTE THE SEEK
        BMI     DLFIN           ;IF MI WE FAILED
        BIC     #377,R1         ;CLEAR OUT FUNCTION BITS
        BIS     #IE!RDH,R1      ;LOAD CODES FOR READ HEADER
        MOVB    #1,RTTBL+1(R3)  ;INDICATE REVERSE SEEK IN PROGRESS
        MOVB    S.ITM(R4),S.CTM(R4)  ;SET DEVICE TIMEOUT COUNTER
        MOV     R1,(R2)         ;LOAD FUNCTION AND GO
        RETURN                  ;WAIT FOR THE INTERRUPT
20$:    TST     (SP)+           ;REMOVE RETRY COUNT
        MOV     RLMP(R2),R1     ;RETRIEVE HEADER WORD
DLDIFO: CLR     I.PRM+6(R3)     ;RESET DIFFERENCE WORD
        BIC     #77,R0          ;MASK OUT SECTOR BITS
        BIC     #77,R1          ;...
        CMP     R0,R1           ;DO WE NEED TO DO A SEEK?
        BEQ     40$             ;IF EQ NO
        MOV     R0,-(SP)        ;SAVE DESIRED DISK ADDRESS
        BIC     #^C<100>,(SP)   ;ISOLATE SURFACE BIT
        ASR     (SP)            ;PUT INTO THE PROPER POSITION
        ASR     (SP)            ;...
        BIC     #100,R0         ;REMOVE SURFACE BIT
        BIC     #100,R1         ;...
        SUB     R0,R1           ;SUBTRACT DESIRED FROM ACTUAL
        BCC     30$             ;IF CC ACTUAL >= DESIRED
        NEG     R1              ;ACTUAL < DESIRED, MAKE POSITIVE DIFFERENCE
        BIS     #SN,R1          ;SET SIGN FOR MOVE TO CENTER OF DISK
30$:    INC     R1              ;SET MARKER BIT
        BIS     (SP)+,R1        ;MERGE IN SURFACE BIT
        MOV     R1,RLDA(R2)     ;LOAD DIFFERENCE WORD
        MOV     R1,I.PRM+6(R3)  ;SAVE DIFFERENCE WORD
40$:    RETURN                  ;

;+
; MOVE THE CONTROLLER/DRIVE REGISTERS INTO THE SPECIFIED BUFFER.
;
; INPUTS:
;       R2 = CSR ADDRESS
;       R3 = BUFFER ADDRESS
;
;-
```

```
REGPAS: MOV     (R2),(R2)+      ;MOVE RLCS
        MOV     RLBA(R2),(R3)+  ;MOVE RLBA
        MOV     RLDA(R2),(R3)+  ;MOVE RLDA
        MOV     RLMP(R2),(R3)+  ;MOVE RLMP
        CLR     (R3)+           ;CLEAR PLACE HOLDERS...
        CLR     (R3)+           ;...SO HRC/CON WILL WORK
        CALL    DLGST           ;EXECUTE GET DRIVE STATUS FUNCTION
        MOV     RLMP(R2),(R3)   ;SAVE DRIVE STATUS
        RETURN                  ;

;+
; **-DLKRB-CONTROLLER ON-LINE/OFF-LINE ROUTINE
;
; THIS ROUTINE WILL HANDLE RECONFIGURATION CALLS FOR ON-LINE
; CONTROLLER AND OFF-LINE CONTROLLER FOR THE RLF11.
;
; INPUTS:
;       R2 = KRB ADDRESS
;       R3 = CTB ADDRESS
;       C=1 IF OFF-LINE REQUEST
;       C=0 IF ON-LINE REQUEST
;
; OUTPUTS:
;       NONE
;-

DLKRB:  BCS     DLOFL           ;HANDLE OFF-LINE REQUEST

;
; CODE SPECIFIC TO HANDLE THE CONTROLLER COMING ON-LINE.
;
        RETURN                  ;EXIT

;
; CODE SPECIFIC TO HANDLE THE CONTROLLER GOING OFF-LINE
;
DLOFL:                          ;
        RETURN                  ;

;+
; **-DLUCB-UNIT ON-LINE/OFF-LINE ROUTINES
;
; THIS ROUTINE WILL HANDLE RECONFIGURATION CALLS FOR ON-LINE
; UNIT AND OFF-LINE UNIT FOR RLO1 AND RLO2 DRIVES.
;
; INPUTS:
;       R3 = CONTROLLER INDEX
;       R4 = SCB ADDRESS
;       R5 = UCB ADDRESS
;       C=1 IF OFF-LINE REQUEST
;       C=0 IF ON-LINE REQUEST
;
; OUTPUTS:
;       NONE
;-
DLUCB:  BCS     DLOFLU          ;IF CS OFF-LINE REQUEST
```

```
;
; CODE SPECIFIC TO BRINGING UNIT ON-LINE.
;
        RETURN                          ;
;
; CODE SPECIFIC TO TAKING UNIT OFF-LINE.
;
DLOFLU:                                 ;
        RETURN                          ;
        .END
```

8.3 Handling Special User Buffers

Some drivers need to handle user buffers in addition to the buffer that the Executive address-checks and relocates in a normal transfer request. Address-checking and relocation operations must take place in the context of the task issuing the I/O request because the mapping registers are set for the issuing task. However, in the normal driver interface, the task context after the call to $GTPKT is not, in general, that of the issuing task.

Thus, drivers that need to handle special buffers must be able to refer to the I/O packet before it is queued, while the context of the issuing task is still intact.

The coding shown in this section is an excerpt from a driver and illustrates the handling of a special user buffer. The key points are:

1. The UC.QUE bit has been set in the control byte (U.CTL) of the UCB for each device/unit.
2. The routine ZZINI, which is defined as the I/O initiation entry point in the driver dispatch table (DDT$) macro call, performs the following actions:
   a. Retrieves the user virtual address and address-checks it
   b. Relocates the virtual address and stores the result back into the packet
   c. Inserts the packet into the I/O queue and continues execution inline to the entry point BMINI, which calls $GTPKT
3. The driver propagates its own execution by branching back to BMINI to call $GTPKT.

```
        .TITLE  BMTAB - DATA BASE FOR BLOCK MOVE DRIVER
        .IDENT  /01/
;
;           COPYRIGHT (c) 1981, 1982 BY
;       DIGITAL EQUIPMENT CORPORATION, MAYNARD
;        MASSACHUSETTS.  ALL RIGHTS RESERVED.
;
; THIS  SOFTWARE IS  FURNISHED  UNDER A  LICENSE  AND  MAY BE USED
; AND COPIED ONLY IN  ACCORDANCE  WITH  THE  TERMS OF SUCH LICENSE
; AND  WITH THE  INCLUSION  OF  THE ABOVE  COPYRIGHT NOTICE.  THIS
; SOFTWARE OR  ANY  OTHER  COPIES  THEREOF, MAY NOT BE PROVIDED OR
; OTHERWISE MADE AVAILABLE TO  ANY OTHER  PERSON.  NO TITLE TO AND
; OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.
;
; THE INFORMATION IN THIS DOCUMENT IS  SUBJECT  TO CHANGE  WITHOUT
; NOTEICE AND SHOULD NOT BE CONSTRUED  AS A COMMITMENT BY  DIGITAL
; EQUIPMENT CORPORATION.
;
; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABLILITY OF
; ITS SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.

;
;
;
; LOADABLE DATA BASE FOR EXAMPLE BUFFERED I/O DRIVER
;
; MACRO LIBRARY CALLS
;
        .MCALL  CLKDF$
        .MCALL  HWDDF$
        .MCALL  SCBDF$
        .MCALL  UCBDF$

        CLKDF$                  ;DEFINE CLOCK BLOCK OFFSETS
        HWDDF$                  ;DEFINE HARDWARE REGISTERS
        SCBDF$  ,,SYSDEF        ;DEFINE SCB OFFSETS
        UCBDF$                  ;DEFINE UCB OFFSETS

$BMDAT::
;
;                                               BM DCB
;
        $BMTBL=0                            ;LOADABLE BMDRV

$BMDCB::
```

```
        .WORD   0               ; D.LNK
        .WORD   .BMO            ; D.UCB
        .ASCII  /BM/            ; D.NAM
        .BYTE   0,1-1           ; D.UNIT,D.UNIT+1
        .WORD   DMND-BMST       ; D.UCBL
        .WORD   $BMTBL          ; D.DSP
                                ; D.MSK - FUNCTION MASKS
        .WORD   33              ; LEGAL    0-17 IO.KIL,IO.WLB,IO.ATT
                                                IO.DET
        .WORD   31              ; CONTROL  0-17 IO.KIL,IO.ATT,IO.DET
        .WORD   0               ; NOOP     0-17
        .WORD   0               ; ACP      0-17
        .WORD   4               ; LEGAL   20-37 IO.WVB
        .WORD   0               ; CONTROL 20-37
        .WORD   0               ; NOOP    20-37
        .WORD   0               ; ACP     20-37
        .WORD   0               ; D.PCB
;
;                                                       BM UCB'S
;
        PRO=0

BMST=.
        .IF     DF M$$MUP
        .WORD   0
        .ENDC

.BMO::
        .WORD   $BMDCB          ; U.DCB
        .WORD   .-2             ; U.RED
        .BYTE   UC.QUE,0        ; U.CTL,U.STS
        .BYTE   0,US.OFL        ; U.UNIT,U.ST2
        .WORD   DV.REC          ; U.CW1
        .WORD   0               ; U.CW2
        .WORD   0               ; U.CW3
        .WORD   72.             ; U.CW4
        .WORD   $BMO            ; U.SCB
        .WORD   0               ; U.ATT
        .WORD   0,0             ; U.BUF,U.BUF+2
        .WORD   0               ; U.CNT

BMND=.

;
;                                                               BM SCB'S
;

$BMO::  .WORD   0,.-2           ; S.LHD
        .WORD   0,0,0,0         ; S.FRK
        .WORD   0               ; S.KS5
        .WORD   0               ; S.PKT
        .BYTE   0,0             ; S.CTM,S.ITM
        .BYTE   0,0             ; S.STS,S.ST3
        .WORD   0               ; S.ST2
        .WORD   0               ; S.KRB - NO KRB SINCE NO CONTROLLER

$BMEND::

        .END
        .TITLE  BMDRV - BLOCK MOVE DRIVER
        .IDENT  /01/
```

```
;+
; COPYRIGHT (c) 1981, 1982 BY DIGITAL EQUIPMENT CORPORATION.
; ALL RIGHTS RESERVED.
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED
; OR COPIED ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
;
;
;
;       THIS IS A SAMPLE DRIVER WHICH DEMONSTRATES HOW TO USE SOME
;       OF THE MORE SOPHISTICATED EXECUTIVE SERVICES AVAILABLE TO
;       I/O DRIVERS. THIS DRIVER DEMONSTRATES:
;
;       1) THE CHECKING OF ADDITIONAL USER BUFFERS PRIOR TO QUEUING
;          AN I/O PACKET.
;
;       2) USE OF THE CLOCK QUEUE FROM A DRIVER.
;
;       3) USE OF THE BUFFERED I/O MECHANISM
;
;       4) USE OF THE GENERAL BUFFERED I/O KERNEL AST MECHANISM
;
;       5) USE OF REGION LOAD KERNEL ASTS
;
;       6) USE OF BLXIO
;
;
;       THIS DRIVER UNDERSTANDS PRECISELY ONE QIO, WHICH IS:
;
;       ...     IO.WLB,.....,<DEST-BUFFER,LENGTH,TIME,SRC-BUFFER>
;                 OR
;               IO.WVB
;
;        THE DRIVER QUEUES A CLOCK BLOCK FOR THE TIME TICKS AND AT THE
;        END OF THAT TIME INTERVAL COPIES THE SOURCE BUFFER TO THE
;        DESTINATION BUFFER. IF POSSIBLE, THE REQUEST IS BUFFERED
;        INTERNALLY WHILE THE CLOCK REQUEST IS POSTED.
;
         .MCALL  CLKDF$, PKTDF$
         CLKDF$                  ;DEFINE CLOCK BLOCK OFFSETS
         PKTDF$                  ;DEFINE I/O PACKET OFFSETS
;
;        DEFINE MAXIMUM TRANSFER LENGTH WHICH WILL BE BUFFERED
;
         BUFLIM = 100.
         DDT$    BM,,NONE,,,NEW
```

```
;+
; ** - BMINI - I/O INITIATION ENTRY POINT
;
;
; INPUTS:
;
;       DRQIO (BECAUSE THE UC.QUE BIT IS SET IN THE UCB) SETS THE
;       REGISTERS TO THE FOLLOWING:
;
;       R1 = ADDRESS OF I/O PACKET
;       R4 = ADDRESS OF SCB
;       R5 = ADDRESS OF UCB
;
; OUTPUTS:
;
;       IF THE SPECIFIED CONTROLLER IS NOT BUSY AND AN I/O REQUEST IS WAIT-
;       ING TO BE PROCESSED, THEN THE REQUEST IS DEQUEUED AND THE I/O OPER-
;       ATION IS INITIATED.
;
;       I/O REQUEST PACKET FORMAT:
;
;       I.LNK       -- I/O QUEUE THREAD WORD.
;       I.PRI/I.EFN -- REQUEST PRIORITY, EVENT FLAG NUMBER.
;       I.TCB       -- ADDRESS OF THE TCB OF THE REQUESTER TASK.
;       I.LN2       -- POINTER TO SECOND LUN WORD IN REQUESTER TASK HEADER.
;       I.UCB       -- UCB ADDRESS OF DEVICE
;       I.FCN       -- I/O FUNCTION CODE (IO.WLB).
;       I.IOSB      -- VIRTUAL ADDRESS OF I/O STATUS BLOCK.
;       I.IOSB+2    -- RELOCATION BIAS OF I/O STATUS BLOCK.
;       I.IOSB+4    -- I/O STATUS BLOCK ADDRESS (DISPLACEMENT + 140000).
;       I.IOSB+6    -- VIRTUAL ADDRESS OF AST SERVICE ROUTINE.
;       I.PRM       -- RELOCATION BIAS OF SOURCE BUFFER.
;       I.PRM+2     -- BUFFER ADDRESS OF I/O TRANSFER.
;       I.PRM+4     -- NUMBER OF BYTES TO BE TRANSFERED.
;       I.PRM+6     -- TIME DISPLACEMENT IN TICKS
;       I.PRM+10    -- VIRTUAL ADDRES (TO BECOME RELOCATION BIAS) OF
;                      DESTINATION BUFFER
;       I.PRM+12    -- FILLED IN WITH DISPLACEMENT ADDRESS OF DESTINATION
;                      BUFFER
;       I.PRM+14    -- USED TO STORE BUFFER/CLOCK BLOCK ADDRESS
;       I.PRM+16    -- FILLED IN WITH PCB ADDRESS OF OUTPUT BUFFER
;-

        .ENABLE  LSB

;       ************************************************************
;       *                                                          *
;       *       I N I T I A T I O N   E N T R Y   P O I N T        *
;       *                                                          *
;       ************************************************************

BMINI:                          ; PRE-QUEUING INITIALIZE ENTRY POINT

;       ************************************************************
;       *                                                          *
;       *     ADDRESS CHECK THE SOURCE BUFFER WHILE THE TASKS      *
;       *     CONTEXT IS LOADED, AND FILL IN THE NECESSARY         *
;       *     PARAMETERS IN THE I/O PACKET                         *
;       *                                                          *
;       ************************************************************
```

```
        MOV   R1,R3             ; COPY ADDRESS OF I/O PACKET
        MOV   I.PRM+10(R1),R0   ; GET VIRTUAL ADDRESS OF SOURCE BUFFER
        MOV   I.PRM+4(R3),R1    ; AND LENGTH OF SOURCE BUFFER

;       +-------------------------------------------------------+
;       |                                                       |
;       |     THE INPUT PARAMETERS FOR $CKBFR ARE:              |
;       |                                                       |
;       |     R0 = STARTING ADDRESS OF BLOCK TO BE CHECKED      |
;       |     R1 = LENGTH OF THE BLOCK TO BE CHECKED            |
;       |     $ATTPT = ADDRESS OF I.AADA IN I/O PACKET          |
;       |              (ESTABLISHED IN DRQIO)                   |
;       |     CURRENT TASK HEADER MUST BE MAPPED THROUGH APR 6  |
;       |              (ESTABLISHED BY DIRECTIVE DISPATCHER)    |
;       |                                                       |
;       |     THE OUTPUT PARAMETERS ARE:                        |
;       |                                                       |
;       |     C = 0 IF CHECK AND PACKET UPDATE SUCCESSFUL       |
;       |           I.AADA OR I.AADA IN PACKET POINTS TO        |
;       |           RELATED ADB, P.IOC, A.IOC INCREMENTED       |
;       |     C = 1 IF CHECK UNSUCCESFUL OR I.AADA, I.AADA      |
;       |           ALREADY FILLED IN                           |
;       |                                                       |
;       +-------------------------------------------------------+

        CALL  $CKBFR            ; CHECK BUFFER, INCREMENT A.IOC AND
                                ; P.IOC FOR APPROPRIATE REGIONS
        BCC   10$               ; IF CC ALL WAS OK

;       ***********************************************************
;       *                                                         *
;       *     SOURCE BUFFER WAS ILLEGAL, FINISH I/O HERE          *
;       *                                                         *
;       ***********************************************************

        MOV   #IE.SPC&377,R0    ; SET COMPLETION STATUS
        CLR   R1                ; AND NUMBER OF BYTES TRANSFERRED

;       +-------------------------------------------------------+
;       |                                                       |
;       |     THE INPUT PARAMETERS FOR $IOFIN ARE:              |
;       |                                                       |
;       |     R0 = FIRST  WORD OF I/O STATUS TO RETURN          |
;       |     R1 = SECOND WORD OF I/O STATUS TO RETURN          |
;       |     R3 = ADDRESS OF I/O PACKET                        |
;       |                                                       |
;       |     THE OUTPUT PARAMETERS ARE:                        |
;       |                                                       |
;       |     R4 IS DESTROYED                                   |
;       |                                                       |
;       +-------------------------------------------------------+

        CALLR $IOFIN            ; COMPLETE I/O AND EXIT DRIVER

;       ***********************************************************
;       *                                                         *
;       *     BUFFER WAS LEGAL, CONVERT VIRTUAL ADDRESS TO        *
;       *     ADDRESS DOUBLEWORD AND STORE PARAMETERS             *
;       *                                                         *
;       ***********************************************************
```

```
;       +-------------------------------------------------------+
;       |                                                       |
;       |     THE INPUT PARAMETERS FOR $RELOC ARE:              |
;       |                                                       |
;       |     R0 = USER VIRTUAL ADDRESS TO RELOCATE             |
;       |                                                       |
;       |     THE OUTPUT PARAMETERS ARE:                        |
;       |                                                       |
;       |     R1 = APR6 RELOCATION BIAS OF USER BUFFER          |
;       |     R2 = DISPLACEMENT IN BLOCK + 140000               |
;       |                                                       |
;       +-------------------------------------------------------+

10$     CALL   $RELOC           ; RELOCATE BUFFER ADDRESS
        MOV    R1,I.PRM+10(R3) ; SAVE APR BIAS OF SOURCE BUFFER
        MOV    R2,I.PRM+12(R3) ; AND DISPLACEMENT ADDRESS
        CLR    I.PRM+16(R3)    ; INDICATE NOT BUFFERED I/O

;       ************************************************************
;       *                                                          *
;       *     NOW QUEUE THE PACKET IN THE DEVICE QUEUE             *
;       *                                                          *
;       ************************************************************

        MOV    R4,R0            ; COPY POINTER TO I/O QUEUE LISTHEAD
        MOV    R3,R1            ; AND ADDRESS OF I/O PACKET

;       +-------------------------------------------------------+
;       |                                                       |
;       |     THE INPUT PARAMETERS FOR $QINSP ARE:              |
;       |                                                       |
;       |     R0 = ADDRESS OF THE TWO-WORD LISTHEAD             |
;       |     R1 = ADDRESS OF THE PACKET TO BE INSERTED         |
;       |                                                       |
;       |     NO OUTPUT PARAMETERS                              |
;       |                                                       |
;       +-------------------------------------------------------+

        CALL   $QINSP           ; INSERT PACKET IN QUEUE

;       ************************************************************
;       *                                                          *
;       *     BEGIN SERIAL PROCESSING OF I/O PACKETS               *
;       *                                                          *
;       ************************************************************
```

```
;       +-------------------------------------------------------+
;       |                                                       |
;       |     THE INPUT PARAMETERS FOR $GTPKT ARE:              |
;       |                                                       |
;       |     R5 = ADDRESS OF THE UCB OF REQUESTING UNIT        |
;       |                                                       |
;       |     THE OUTPUT PARAMETERS ARE:                        |
;       |                                                       |
;       |     C = 0 IF A REQUEST WAS SUCCESSFULLY DEQUEUED      |
;       |             R1 = ADDRESS OF THE I/O PACKET            |
;       |             R2 = PHYSICAL UNIT NUMBER                 |
;       |             R3 = CONTROLLER INDEX                     |
;       |             R4 = SCB ADDRESS OF CONTROLLER            |
;       |             R5 = UCB ADDRESS OF UNIT                  |
;       |     C = 1 IF UNIT BUSY OR NO PACKETS QUEUED           |
;       |                                                       |
;       +-------------------------------------------------------+
BMIN1:  CALL    $GTPKT          ; ATTEMT TO GET A REQUEST
        BCC     20$             ; IF CC WE GOT ONE
        RETURN                  ; DEVICE BUSY OR QUEUE EMPTY
                                ; REFERENCE LABEL
;       ************************************************************
;       *                                                          *
;       *       ATTEMPT TO ALLOCATE CLOCK BLOCK                    *
;       *                                                          *
;       ************************************************************

        MOV     R1,R3           ; COPY I/O PACKET ADDRESS
        MOV     #C.LGTH,R1      ; SET LENGTH OF CLOCK BLOCK

;       +-------------------------------------------------------+
;       |                                                       |
;       |       THE INPUT PARAMETERS FOR $ALOCB ARE:            |
;       |                                                       |
;       |       R1 = SIZE OF THE BLOCK TO ALLOCATE (IN BYTES)   |
;       |                                                       |
;       |       THE OUTPUT PARAMETERS ARE:                      |
;       |                                                       |
;       |       C = 0 IF A BLOCK WAS SUCCESSFULLY ALLOCATED     |
;       |               R0 = ADDRESS OF THE ALLOCATED BLOCK     |
;       |               R1 = LENGTH OF THE ALLOCATED BLOCK      |
;       |       C = 1 IF NO BLOCK IS CURRENTLY AVAILABLE        |
;       |                                                       |
;       +-------------------------------------------------------+

        CALL    $ALOCB          ; ATTEMPT TO ALLOCATE
        BCC     30$             ; IF CC SUCCESSFUL
        MOV     #IE.NOD&377,R0  ; SET I/O STATUS
```

```
;       +-------------------------------------------------------+
;       |                                                       |
;       |      THE INPUT PARAMETERS FOR $IOALT ARE:             |
;       |                                                       |
;       |      R0 = FIRST WORD OF I/O STATUS BLOCK              |
;       |      R1 = SECOND WORD OF I/O STATUS BLOCK             |
;       |      R2 = STARTING AND FINAL RETRY COUNTS             |
;       |           (IF AN ERROR LOGGING DEVICE)                |
;       |      R5 = UCB ADDRESS OF UNIT TO COMPLETE             |
;       |                                                       |
;       |      THE OUTPUT PARAMETERS ARE:                       |
;       |                                                       |
;       |      R4 IS DESTROYED                                  |
;       |                                                       |
;       +-------------------------------------------------------+

        CALL    $IOALT          ; AND COMPLETE THE I/O
        BR      BMIN1           ; GO LOOK FOR MORE WORK
30$:    MOV     R0,I.PRM+14(R3) ; SAVE ADDRESS OF CLOCK BLOCK

;       ***********************************************************
;       *                                                         *
;       *      DETERMINE IF I/O REQUEST IS BUFFERABLE             *
;       *                                                         *
;       ***********************************************************

;       +-------------------------------------------------------+
;       |                                                       |
;       |      THE INPUT PARAMETERS FOR $TSTBF ARE:             |
;       |                                                       |
;       |      R3 = ADDRESS OF I/O PACKET TO TEST               |
;       |                                                       |
;       |      THE OUTPUT PARAMETERS ARE:                       |
;       |                                                       |
;       |      C = 0 IF REQUEST MAY BE BUFFERED                 |
;       |      C = 1 IF REQUEST MAY NOT BE BUFFERED             |
;       |                                                       |
;       +-------------------------------------------------------+

        CALL    $TSTBF          ; TEST FOR BUFFERABLE I/O REQUEST
        BCS     40$             ; IF CS CAN'T ALLOCATE A BUFFER

;       ***********************************************************
;       *                                                         *
;       *      ATTEMPT TO ALLOCATE A BUFFER                       *
;       *                                                         *
;       ***********************************************************

        MOV     I.PRM+4(R3),R1  ; GET LENGTH OF BUFFER
        CMP     R1,#BUFLIM      ; BIGGER TAN BUFFER LIMIT ?
        BHI     40$             ; IF HI YES, DON'T BUFFER
```

```
;       +-------------------------------------------------------+
;       |                                                       |
;       |      THE INPUT PARAMETERS FOR $ALOCB ARE:             |
;       |                                                       |
;       |      R1 = SIZE OF THE BLOCK TO ALLOCATE (IN BYTES)    |
;       |                                                       |
;       |      THE OUTPUT PARAMETERS ARE:                       |
;       |                                                       |
;       |      C = 0 IF A BLOCK WAS SUCCESSFULLY ALLOCATED      |
;       |             R0 = ADDRESS OF THE ALLOCATED BLOCK       |
;       |             R1 = LENGTH OF THE ALLOCATED BLOCK        |
;       |      C = 1 IF NO BLOCK IS CURRENTLY AVAILABLE         |
;       |                                                       |
;       +-------------------------------------------------------+

        CALL    $ALOCB          ; TRY TO ALLOCATE BUFFER
        BCS     40$             ; IF CS COULDN'T GET ONE

;       ************************************************************
;       *                                                          *
;       *       COPY USER BUFFER TO INTERNAL BUFFER                *
;       *                                                          *
;       ************************************************************

        MOV     R0,R4           ; SET ADDRESS OF DESTINATION BUFFER
        MOV     R3,R5           ; SAVE ADDRESS OF I/O PACKET
        MOV     I.PRM+4(R5),R0  ; SET LENGTH OF TRANSFER
        MOV     I.PRM+10(R5),R1 ; SET BIAS OF SOURCE BUFFER
        MOV     I.PRM+12(R5),R2 ; AND DISPLACEMENT
        BIC     #140000,R2      ; STRIP OFF APR6 ADDRESS BITS
        BIS     #120000,R2      ; AND SUBSTITUTE APR5
        MOV     R4,I.PRM+10(R5) ; SET INTERNAL BUFFER ADDRESS INTO PACKET

;       +-------------------------------------------------------+
;       |                                                       |
;       |      THE INPUT PARAMETERS FOR $BLXIO ARE:             |
;       |                                                       |
;       |      R0 = NUMBER OF BYTES TO MOVE                     |
;       |      R1 = SOURCE APR 5 BIAS                           |
;       |      R2 = SOURCE DISPLACEMENT                         |
;       |      R3 = DESTINATION APR6 BIAS                       |
;       |      R4 = DESTINATION DISPLACEMENT                    |
;       |                                                       |
;       |      THE OUTPUT PARAMETERS ARE:                       |
;       |                                                       |
;       |      R0 ALTERED                                       |
;       |      R1,R3 PRESERVED                                  |
;       |      R2,R4 POINT TO LAST BYTE OF SOURCE/DESTINATION +1|
;       |                                                       |
;       +-------------------------------------------------------+

        CALL    $BLXIO          ; COPY TO INTERNAL BUFFER

;       ************************************************************
;       *                                                          *
;       *       CONVERT TO BUFFERED I/O REQUEST                    *
;       *                                                          *
;       ************************************************************

        MOV     R5,R3           ; COPY I/O PACKET ADDRESS BACK
```

```
;       +-------------------------------------------------------+
;       |                                                       |
;       |     THE INPUT PARAMETERS FOR $INIBF ARE:              |
;       |                                                       |
;       |     R3 = ADDRESS OF THE I/O PACKET TO BUFFER          |
;       |                                                       |
;       |     NO OUTPUT PARAMETERS.                             |
;       |                                                       |
;       +-------------------------------------------------------+

        CALL    $INIBF          ; INITIALIZE BUFFERED I/O

;       ************************************************************
;       *                                                          *
;       *     QUEUE THE CLOCK BLOCK                                *
;       *                                                          *
;       ************************************************************

40$:    MOV     I.PRM+14(R3),R0 ; GET ADDRESS OF CLOCK BLOCK
        MOV     #CLKSRV,C.SUB(R0) ; SET ADDRESS OF SUBROUTINE
        CLR     R1              ; HIGH ORDER DELTA TIME
        MOV     I.PRM+6(R3),R2  ; LOW ORDER PART
        MOV     #C.SYST,R4      ; SET REQUEST TYPE
        MOV     R3,R5           ; USE PACKET ADDRESS AS IDENTIFIER

;       +-------------------------------------------------------+
;       |                                                       |
;       |     THE INPUT PARAMETERS FOR $CLINS ARE:              |
;       |                                                       |
;       |     R0 = ADDRESS OF THE CLOCK BLOCK TO QUEUE          |
;       |     R1 = HIGH ORDER HALF OF DELTA TIME                |
;       |     R2 = LOW ORDER HALF OF DELTA TIME                 |
;       |     R4 = REQUEST TYPE                                 |
;       |     R5 = ADDRESS OF REQUESTING TASK OR IDENTIFIER     |
;       |                                                       |
;       |     NO OUTPUT PARAMETERS.                             |
;       |                                                       |
;       +-------------------------------------------------------+

        CALLR   $CLINS          ; QUEUE CLOCK BLOCK AND TEMPORARILY
                                ; EXIT THE DRIVER

;       ************************************************************
;       *                                                          *
;       *     C L O C K   E N T R Y   P O I N T                    *
;       *                                                          *
;       ************************************************************

;       ************************************************************
;       *                                                          *
;       *     CHECK TO SEE IF THE I/O WAS BUFERED                  *
;       *                                                          *
;       ************************************************************

CLKSRV: MOV     C.TCB(R),R5     ; GET ADDRESS OF I/O PACKET
        TST     I.PRM+16(R5)    ; WAS IT BUFFERED I/O
        BNE     50$             ; IF NE YES, GO QUEUE KERNEL AST
```

```
;       ************************************************************
;       *                                                          *
;       *      COULDN'T BUFFER, PERFORM COPY HERE AND NOW          *
;       *                                                          *
;       ************************************************************

        MOV     I.PRM+4(R5),R0  ; SET LENGTH TO TRANSFER
        MOV     I.PRM+10(R5),R1 ; BIAS OF SOURCE BUFFER
        MOV     I.PRM+12(R5),R2 ; DISPLACEMENT OF SOURCE
        BIC     #140000,R2      ; STRIP OFF APR6 ADDRESS BITS
        BIS     #120000,R2      ; AND CONVERT TO APR5
        MOV     I.PRM(R5),R3    ; SET BIAS OF DESTINATION
        MOV     I.PRM+2(R5),R4   ; SET DISPLACEMENT

;       +-----------------------------------------------------------+
;       |                                                           |
;       |       THE INPUT PARAMETERS FOR $BLXIO ARE:                |
;       |                                                           |
;       |       R0 = NUMBER OF BYTES TO MOVE                        |
;       |       R1 = SOURCE APR 5 BIAS                              |
;       |       R2 = SOURCE DISPLACEMENT                            |
;       |       R3 = DESTINATION APR6 BIAS                          |
;       |       R4 = DESTINATION DISPLACEMENT                       |
;       |                                                           |
;       |       THE OUTPUT PARAMETERS ARE:                          |
;       |                                                           |
;       |       R0 ALTERED                                          |
;       |       R1,R3 PRESERVED                                     |
;       |       R2,R4 POINT TO LAST BYTE OF SOURCE/DESTINATION +1   |
;       |                                                           |
;       +-----------------------------------------------------------+

        CALL    $BLXIO          ; COPY BUFFER
        MOV     I.PRM+14(R5),R0 ; GET ADDRESS OF CLOCK BLOCK
        MOV     #C.LGTH,R1      ; GET LENGTH OF CLOCK BLOCK

;       +-----------------------------------------------------------+
;       |                                                           |
;       |       THE INPUT PARAMETERS FOR $DEACB ARE:                |
;       |                                                           |
;       |       R0 = ADDRESS OF BLOCK TO DEALLOCATE                 |
;       |       R1 = LENGTH OF BLOCK TO DEALLOCATE                  |
;       |                                                           |
;       |       NO OUTPUT PARAMETERS.                               |
;       |                                                           |
;       +-----------------------------------------------------------+

        CALL    $DEACB          ; DEALLOCATE IT
        MOV     R5,R3           ; COPY PACKET ADDRESS FOR $IODON
BMSUC:  MOV     #IS.SUC&377,R0  ; SET FINAL I/O STATUS
        MOV     I.RPM+4(R3),R1  ; AND LENGTH OF TRANSFER = REQUESTED
BMDON:  MOV     I.UCB(R3),R5    ; GET UCB ADDRESS OF DEVICE
```

```
;       +-------------------------------------------------------+
;       |                                                       |
;       |      THE INPUT PARAMETERS FOR $IODON ARE:             |
;       |                                                       |
;       |      R0 = FIRST  WORD OF I/O STATUS BLOCK             |
;       |      R1 = SECOND WORD OF I/O STATUS BLOCK             |
;       |      R2 = STARTING AND FINAL RETRY COUNTS             |
;       |           (IF AN ERROR LOGGING DEVICE)                |
;       |      R5 = UCB ADDRESS OF UNIT TO COMPLETE             |
;       |                                                       |
;       |      THE OUTPUT PARAMETERS ARE:                       |
;       |                                                       |
;       |      R4 IS DESTROYED                                  |
;       |                                                       |
;       +-------------------------------------------------------+

        CALL    $IODON          ; COMPLETE THE I/O
        BR      BMIN1           ; GO LOOK FOR MORE WORK
;       ***********************************************************
;       *                                                         *
;       *      BUFFERED I/O, CONVERT I/O PACKET TO KERNEL         *
;       *      AST AND EXIT FROM DRIVER                           *
;       *                                                         *
;       ***********************************************************

50$:    MOV     R4,R3           ; COPY CLOCK BLOCK ADDRESS FOR $REQUE
        MOV     I.TCB(R5),R0    ; POINT TO TCB OF TASK
        TST     (R4)+           ; SKIP LINK WORD
        MOV     #AK.GBI,(R4)+   ; SET A.CBL=AK.GBI
        MOV     KISAR5,(R4)+    ; SET APR BIAS OF SERVICE ROUTINE
        MOV     #KATSRV,(R4)+   ; SET ADDRESS OF PROCESSING ROUTINE
        MOV     R5,(R4)+        ; SAVE I/O PACKET ADDRESS IN CLOCK BLOCK

;       +-------------------------------------------------------+
;       |                                                       |
;       |      THE INPUT PARAMETERS FOR $REQUE ARE:             |
;       |                                                       |
;       |      R0 = TCB ADDRESS TO QUEUE AST BLOCK TO           |
;       |      R3 = ADDRESS OF THE PACKET TO QUEUE              |
;       |                                                       |
;       |      NO OUTPUT PARAMETERS.                            |
;       |                                                       |
;       +-------------------------------------------------------+

        CALLR   $REQUE          ; QUEUE AST TO TASK

;       ***********************************************************
;       *                                                         *
;       *      K E R N E L   A S T   E N T R Y   P O I N T        *
;       *                                                         *
;       ***********************************************************

;       ***********************************************************
;       *                                                         *
;       *      GET PCB ADDRESS AND SEE IF PARTITION IS RESIDENT   *
;       *                                                         *
;       ***********************************************************

KATSRV: MOV     10(R3),R5       ; GET I/O PACKET ADDRESS
        MOV     I.PRM+16(R5),R1 ; GET PCB ADDRESS OF BUFFER REGION
        BEQ     70$             ; IF EQ THERE IS NO COPY TO PERFORM
```

```
;       +-------------------------------------------------------+
;       |                                                       |
;       |      THE INPUT PARAMETERS FOR $TSPAR ARE:             |
;       |                                                       |
;       |      R0 = ADDRESS OF THE PACKET (THE KERNEL AST BLOCK)|
;       |      R1 = PCB ADDRESS OF THE PCB CONTAINING THE BUFFER|
;       |      R5 = TCB ADDRESS OF ASSOCIATED TASK              |
;       |                                                       |
;       |      THE OUTPUT PARAMETERS ARE:                       |
;       |                                                       |
;       |      C = 0 IF REGION IS RESIDENT AND CAN BE ACCESSED  |
;       |      C = 1 IF REGION IS NOT RESIDENT AND AST HAS      |
;       |           BEEN QUEUED                                 |
;       |                                                       |
;       +-------------------------------------------------------+

        CALL    $TSPAR          ; REGION IN MEMORY ?
        BCC     60$             ; IF CC REGION IN MEMORY

;       ************************************************************
;       *                                                          *
;       *      A REGION AST WAS QUEUED. BUMP BUFFERED I/O COUNT    *
;       *      BACK UP TO FORCE I/O RUNDOWN IN CASE OF ABORT AND   *
;       *      EXIT AST SERVICE ROUTINE.                           *
;       *                                                          *
;       ************************************************************

        MOV     I.TCB(R5),R0    ; GET TCB ADDRESS
        INCB    T.TIO(R0)       ; BUMP BUFFERED I/O COUNT
        RETURN                  ; EXIT AST SERVICE ROUTINE
;       ************************************************************
;       *                                                          *
;       *      PERFORM BUFFER COPY OPERATION                       *
;       *                                                          *
;       ************************************************************

60$:    MOV     I.TCB(R5),R0    ; GET TCB ADDRESS OF TASK
        INCB    T.IOC(R0)       ; ADJUST REAL I/O COUNT UPWARDS
        MOV     I.PRM+4(R5),R2  ; GET COUNT OF BYTES
        MOV     I.PRM+10(R5),R2 ; SET SOURCE BUFFER ADDRESS
        MOV     P.REL(R1),R3    ; GET STARTING BIAS OF PARTITION
        ADD     I.PRM(R5),R3    ; AND ADD IN OFFSET
        MOV     I.PRM+2(R5),R4  ; SET DISPLACEMENT

;       +-------------------------------------------------------+
;       |                                                       |
;       |      THE INPUT PARAMETERS FOR $BLXIO ARE:             |
;       |                                                       |
;       |      R0 = NUMBER OF BYTES TO MOVE                     |
;       |      R1 = SOURCE APR 5 BIAS                           |
;       |      R2 = SOURCE DISPLACEMENT                         |
;       |      R3 = DESTINATION APR6 BIAS                       |
;       |      R4 = DESTINATION DISPLACEMENT                    |
;       |                                                       |
;       |      THE OUTPUT PARAMETERS ARE:                       |
;       |                                                       |
;       |      R0 ALTERED                                       |
;       |      R1,R3 PRESERVED                                  |
;       |      R2,R4 POINT TO LAST BYTE OF SOURCE/DESTINATION +1|
;       |                                                       |
;       +-------------------------------------------------------+
```

```
        CALL    $BLXIO          ; COPY BUFFER
        MOV     I.PRM+10(R5),R0 ; GET BUFFER ADDRESS AGAIN
        MOV     I.PRM+4(R5),R1  ; GET LENGTH OF BUFFER

;       +-------------------------------------------------------+
;       |                                                       |
;       |       THE INPUT PARAMETERS FOR $DEACB ARE:            |
;       |                                                       |
;       |       R0 = ADDRESS OF BLOCK TO DEALLOCATE             |
;       |       R1 = LENGTH OF BLOCK TO DEALLOCATE              |
;       |                                                       |
;       |       NO OUTPUT PARAMETERS.                           |
;       |                                                       |
;       +-------------------------------------------------------+

        CALL    $DEACB          ; DEALLOCATE IT

;       ***********************************************************
;       *                                                         *
;       *       IF THIS WASN'T A REGION LOAD AST, FINISH THE I/O  *
;       *                                                         *
;       ***********************************************************

70$:    MOV     I.PRM+14(R5),R0 ; RETRIEVE AST BLOCK ADDRESS
        TST     (R0)            ; WAS THIS A REGION LOAD AST?
        BNE     80$             ; IF NE YES

        MOV     #C.LGTH,R1      ; SET LENGTH OF CLOCK BLOCK

;       +-------------------------------------------------------+
;       |                                                       |
;       |       THE INPUT PARAMETERS FOR $DEACB ARE:            |
;       |                                                       |
;       |       R0 = ADDRESS OF BLOCK TO DEALLOCATE             |
;       |       R1 = LENGTH OF BLOCK TO DEALLOCATE              |
;       |                                                       |
;       |       NO OUTPUT PARAMETERS.                           |
;       |                                                       |
;       +-------------------------------------------------------+

        CALL    $DEACB          ; DEALLOCATE CLOCK BLOCK
        MOV     I.IOSB(R5),R3   ; GET VIRTUAL ADDRESS OF I/O STATUS BLOCK
        MOV     #IS.SUC&377,-(SP) ; SET FIRST I/O STATUS WORD
        MTPD$   (R3)+           ; WRITE FIRST WORD OF STATUS (MAY TRAP)
        MOV     I.PRM+4(R5),-(SP) ; SET SECOND WORD OF I/O STATUS
        MTPD$   (R3)            ; WRITE SECOND WORD (MAY TRAP)
        CLR     I.IOSB(R5)      ; PREVENT $IODON ATTEMPT TO WRITE STATUS
        MOV     R5,R3           ; COPY I/O PACKET ADDRESS
        JMP     BMSUC           ; FINISH IN COMMON CODE

;       ***********************************************************
;       *                                                         *
;       *       RECONVERT REGION LOAD AST TO A TASK AST           *
;       *                                                         *
;       ***********************************************************

80$:    MOV     R0,R3           ; COPY BLOCK ADDRESS
        CLR     10(R0)          ; INDICATE NO BUFFER NEXT TIME
        MOV     I.TCB(R5),R0    ; GET TCB ADDRESS
```

```
;       +-------------------------------------------------------+
;       |                                                       |
;       |      THE INPUT PARAMETERS FOR $REQUE ARE:             |
;       |                                                       |
;       |      R0 = TCB ADDRESS TO QUEUE AST BLOCK TO           |
;       |      R3 = ADDRESS OF THE PACKET TO QUEUE              |
;       |                                                       |
;       |      NO OUTPUT PARAMETERS.                            |
;       |                                                       |
;       +-------------------------------------------------------+

        CALLR  $REQUE           ; RE-QUEUE TASK AST AND EXIT AST SERVICE

;       ************************************************************
;       *                                                          *
;       *       MISCELLANEOUS ENTRY POINTS                         *
;       *                                                          *
;       ************************************************************

;       ************************************************************
;       *                                                          *
;       *       C A N C E L   E N T R Y   P O I N T                *
;       *                                                          *
;       *       WE COULD DEQUEUE PENDING CLOCK REQUEST, ETC, HERE, *
;       *       BUT WE DON'T, WE JUST LET THEM COMPLETE LATER       *
;       *                                                          *
;       ************************************************************

BMCAN:

;       ************************************************************
;       *                                                          *
;       *       T I M E O U T   E N T R Y   P O I N T              *
;       *                                                          *
;       *       SINCE THERE'S NO PHYSICAL DEVICE TO TIME OUT, NO-OP *
;       *                                                          *
;       ************************************************************

BMOUT:

;       ************************************************************
;       *                                                          *
;       *       P O W E R F A I L   E N T R Y   P O I N T          *
;       *                                                          *
;       *       POWERFAIL DOESN'T AFFECT NON-EXISTENT DEVICES      *
;       *                                                          *
;       ************************************************************

BMPWF:

;       ************************************************************
;       *                                                          *
;       *       S T A T U S   C H A N G E   E N T R Y   P O I N T  *
;       *                                                          *
;       *       DON'T NEED TO TOUCH NON-EXISTENT DEVICE, JUST LET  *
;       *       EXEC PUT DEVICE ON/OFF LINE                        *
;       *                                                          *
;       ************************************************************

BMKRB:
BMUCB:
        RETURN                  ; ALL THESE ARE NO-OP FOR NOW
        .END
```

Appendix A
Converting A User-Supplied RSX-11M Driver

This appendix describes the modifications you must make to enable an RSX-11M, user-supplied driver to run on an RSX-11M-PLUS system. The modifications involve both the driver data base and the driver code.

A.1 Assumptions and General Approach

The discussion in this appendix assumes that the RSX-11M user-supplied driver runs on a mapped system. Also, samples of code from the RL01/RL02 driver (DLDRV) are used as examples in this appendix.

As a general approach to converting a driver, proceed in the following manner:

1. Read the *RSX-11M-PLUS and Micro/RSX I/O Drivers Reference Manual* to gain a feeling for the differences between RSX-11M and RSX-11M-PLUS drivers. Note especially the differences in the data structures (RSX-11M-PLUS has two additional structures).
2. Make the changes described in this appendix.
3. Incorporate the driver according to the guidelines given in Chapter 5.

For the purposes of this discussion, a standard disk driver is one that does not attempt to use any of the advanced driver features described in Chapter 1.

A.2 Modifying the Data Base Code

Before creating the driver data base, read the overview of programming user-written driver data bases (Section 4.2). It gives important information on ordering and labeling in the code.

A.2.1 Unit Control Block Changes

Make sure that the Unit Control Block (UCB) has the data needed for disk geometry calculations. Refer to the description of U.PRM in Section 4.4.4. The following code is an example of source statements that store the disk geometry:

```
.BYTE   40.,2           ;U.PRM
.WORD   512.            ;U.PRM+2
```

In the preceding code example, the values are device dependent and have the following meanings:

40.	Indicates the number of sectors per track
2	Indicates the number of tracks per cylinder
512.	Indicates the number of cylinders per volume

A.2.2 Status Control Block Changes

RSX-11M-PLUS requires a structure called the Controller Request Block (KRB). You can add the KRB data that RSX-11M-PLUS requires to the Status Control Block (SCB) data to effectively create one continuous structure. This arrangement is called the contiguous KRB/SCB and is described in Sections 4.2.4 and 4.4.7. Because the ordering of the SCB data differs from RSX-11M to RSX-11M-PLUS, you must rearrange the RSX-11M SCB data to accommodate the RSX-11M-PLUS requirements. If your driver refers to the SCB structures by symbolic offset and does not rely on physical ordering, you do not need to change the driver code that accesses the SCB. Refer to Sections 4.4.5 and 4.4.6 for a description of the offsets required.

There must be one KRB/SCB combination for each controller present on the system.

The following code example, from DLDRV, illustrates contiguous KRB/SCB and includes the proper offsets:

```
        .BYTE   PR5,160/4       ;K.PRI,K.VCT
        .BYTE   0*2,0           ;K.CON,K.IOC
        .WORD   KS.OFL          ;K.STS
$DLA::                          ;START OF KRB
        .WORD   174400          ;K.CSR
        .WORD   DLA-$DLA        ;K.OFF
        .BYTE   0,0             ;K.HPU
        .WORD   .DLO            ;K.OWN
$DLO::                          ;START OF CONTIGUOUS SCB
        .WORD   0,.-2           ;S.LHD/K.CRQ
        .WORD   0,0,0,0         ;S.FRK
        .WORD   0               ;S.KS5
        .WORD   0               ;S.PKT
        .BYTE   0,4.            ;S.CTM,S.ITM
        .BYTE   0,0             ;S.STS,S.ST3
        .WORD   S2.CON!S2.LOG   ;S.ST2
        .WORD   $DLA            ;S.KRB
        .BYTE   5.,0            ;S.RCNT,S.ROFF
        .WORD   0               ;S.EMB
        .BLKW   6               ;MAPPING ASSIGNMENT BLOCK
        .WORD   0               ;KE.RHB
DLA:
```

In the preceding code example, label DLA is used solely for calculating K.OFF (DLA-$DLA). K.VCT and K.CSR can be changed dynamically by reconfiguration commands when you bring the device on-line. Refer to the *RSX-11M-PLUS and Micro/RSX System Management Guide* for information on the reconfiguration task and commands.

A.2.3 The Controller Table

RSX-11M-PLUS requires a structure called a Controller Table (CTB). Add the code to define the CTB according to the rules described in Sections 4.2.5 and 4.4.8.

The following code example, from DLDRV, is a sample of the code needed to define the CTB:

```
        .WORD   0               ;L.ICB
DLCTB:                          ;START OF CTB
        .WORD   0               ;L.LNK
        .ASCII  /DL/            ;L.NAM
        .WORD   $DLDCB          ;L.DCB
        .BYTE   1,0             ;L.NUM,L.STS
$DLCTB::                        ;POINTER TO THE DEVICE CONTROL BLOCK (DCB)
        .WORD   $DLA            ;L.KRB
```

The preceding example assumes that you have a loadable data base. If the data base is resident, you must include the CTB macro before L.LNK. In the example, L.KRB points to the start of the KRB.

A.3 Modifying the Driver Code

Several changes must be made to the RSX-11M driver code. For an overview of the RSX-11M-PLUS coding requirements, refer to Section 4.3.

A.3.1 Conditional Symbols

You can remove most dependence on system-conditional definitions from the code. RSX-11M-PLUS always defines the symbols D$$IAG, M$$MGE, M$$EXT, M$$MUP, and E$$DVC.

A.3.2 Controller-Dependent Code

At the I/O initiation entry point in RSX-11M drivers, you will find code for defining a table of UCB addresses and loading the UCB address of the currently active unit in the table. Remove this code and replace it with the GTPKT$ macro call. For guidelines on doing this, refer to Sections 4.3.2 and 4.5.2.

The following is an example of the RSX-11M driver code that you must remove:

```
        CALL    $GTPKT          ;GET AN I/O PACKET TO PROCESS
        BCC     1$              ;IF CC PROCESS REQUEST
        RETURN                  ;RETURN IF BUSY OR NO REQUEST
1$:     MOV     R5,CNTBL(R3)    ;SAVE ADDRESS OF REQUEST UCB
```

Insert the RSX-11M-PLUS GTPKT$ macro call, as in the following example:

```
        GTPKT$  DL,R$$L11       ;GET NEXT I/O PACKET TO PROCESS
```

A.3.3 Driver Dispatch Table

Replace the code that defines the entry point addresses with the DDT$ macro call. Refer to Section 4.3.1 for a description of the call and its parameters. Refer to Section 4.5.1 for a description of the Driver Dispatch Table (DDT) and the format of the labels that it uses for the entry points.

The following is an example of the RSX-11M driver code that you must remove:

```
$DLTBL::.WORD   DLINI           ;DEVICE INITIATOR ENTRY POINT
        .WORD   DLCAN           ;CANCEL I/O OPERATION ENTRY POINT
        .WORD   DLOUT           ;DEVICE TIMEOUT ENTRY POINT
        .WORD   DLPWF           ;POWERFAIL ENTRY POINT
```

Insert the RSX-11M-PLUS DDT$ macro call, as in the following example:

```
        DDT$    DL,R$$L11       ;GENERATE DISPATCH TABLE
```

You do not have to add code to the driver to handle controller and unit status changes. The sample form of the macro call shown generates code to use the xxPWF entry point for controller and unit on-line and off-line status changes.

A.3.4 Reconfiguration Support

If you incorporate the device in the Reconfiguration task (HRC . . .) tables and the device calls the Executive volume valid routine, you must incorporate a local register pass routine in your driver, as in the following code example:

```
;+
; MOVE THE CONTROLLER/DRIVE REGISTERS INTO THE
; SPECIFIED BUFFER.
;
; INPUTS:
;       R2 = CSR ADDRESS
;       R3 = BUFFER ADDRESS
;
; OUTPUTS:
;       R3 - ALTERED
;-

REGPAS: MOV     (R2),(R3)+      ;MOVE RLCS
        MOV     RLBA(R2),(R3)+  ;MOVE RLBA
        MOV     RLDA(R2),(R3)+  ;MOVE RLDA
        MOV     RLMP(R2),(R3)+  ;MOVE RLMP
        CALL    DLGST           ;EXECUTIVE GET DRIVE STATUS FUNCTION
        MOV     RLMP(R2),(R3)   ;SAVE DRIVE STATUS
        RETURN                  ;
```

In the preceding code example, the index values RLxxx and the subroutine DLGST are device specific.

A.3.5 Volume Valid Processing

If the device is a disk and has a volume valid function, the RSX-11M-PLUS Executive must be able to handle the correct function codes. Refer to the description of volume valid processing in Section 4.5.9. For volume valid support, you may also need to include the code that appears in the following example:

```
        CALL    $VOLVD          ;VALIDATE VOLUME VALID
        BCS     IODON           ;IF CS WE FAILED
        TST     R0              ;TRANSFER FUNCTION?
        BMI     1$              ;IF MI YES
        TST     I.PRM+2(R1)     ;SIZE THE DISK
        BPL     IODON           ;IF PL NO
        MOV     S.CSR(R4),R2    ;RETRIEVE CSR ADDRESS
        CALL    DLRST           ;RESET DRIVE AND GET STATUS
        MOV     S.PKT(R4),R3    ;RETRIEVE I/O PACKET ADDRESS
        CALL    REGPAS          ;PASS REGISTERS TO HRC
        BR      IODON           ;FINISH I/O REQUEST
1$:                             ;REFERENCE LABEL
```

In the preceding example, the subroutine DLRST is device specific.

A.3.6 Converting Logical Block Numbers

The $CVLBN routine converts a Logical Block Number (LBN) to a physical disk address. You can replace special-purpose code in the RSX-11M driver with a call to this Executive routine, a description of which is in Section 7.4.6. A sample of RSX-11M logical block conversion code that you should remove appears in the following example:

```
MOV     #40.,R1         ;DIVIDE BY SECTORS/SURFACE
CALL    $DIV            ;CALCULATE CYLINDER NUMBER
.REPT   6.
ASL     R0              ;POSITION CYLINDER AND SURFACE
.ENDR
BIS     R1,40           ;MERGE SECTOR WITH CYLINDER AND SURFACE
```

The following code example, which includes the call to $CVLBN, illustrates RSX-11M-PLUS logical block number conversion:

```
CALL    $CVLBN          ;CONVERT BLOCK NUMBER TO DISK ADDRESS
ROR     R1              ;PUT SURFACE BIT IN CARRY
ROL     R2              ;MERGE IT WITH THE CYLINDER NUMBER
ASH     #6,R2           ;POSITION CYLINDER AND SURFACE
BIS     R2,R0           ;MERGE SECTOR WITH CYLINDER AND SURFACE
```

A.3.7 Interrupt Entry Processing

Make sure that the INTSV$ macro call appears as the first line of code at each interrupt entry point in the driver. Refer to Section 4.3.3 for a description of the INTSV$ macro call and to Section 4.5.8 for a discussion of processing at an interrupt entry point. The following code example illustrates the INTSV$ macro call:

```
INTSV$  DL,PR5,R$$L11   ;;;SAVE REGISTERS AND SET PRIORITY
```

Index

A

B

C

D

E

F

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

READER'S COMMENTS

Your comments and suggestions are welcome and will help us in our continuous effort to improve the quality and usefulness of our documentation and software.

Remember, the system includes information that you read on your terminal: help files, error messages, prompts, and so on. Please let us know if you have comments about this information, too.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

What kind of user are you? ___ Programmer ___ Nonprogrammer

Years of experience as a computer programmer/user: _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

--- Do Not Tear - Fold Here and Tape ---

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

--- Do Not Tear - Fold Here ---