VAXELN

digital

Internals Manual

Order Number: AA–NC72A–TE

VAXELN Internals Manual

Order Number: AA–NC72A–TE

This manual describes the internal data structures and operations of the VAXELN Kernel and its associated subsystems.

This is a **preliminary** version of the *VAXELN Internals Manual.* The complete edition of the manual is forthcoming and can be ordered separately.

Revision/Update Information:	This is a new manual.
Software Version:	VAXELN, Version 4.0

digital equipment corporation
maynard, massachusetts

First Printing, July 1989

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation.

Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software, if any, described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license. No responsibility is assumed for the use or reliability of software or equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Printed in U.S.A.

The 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	MicroVMS	ULTRIX–32m
DECmate	P/OS	UNIBUS
DECnet	PDP	VAX
DECsystem–10	PDT	VAX DEC/CMS
DECSYSTEM–20	Professional	VAX DEC/MMS
DECUS	Q-bus	VAX Rdb/ELN
DECwriter	Q22-bus	VAX Rdb/VMS
DEQNA	Rainbow	VAXBI
DEUNA	RSTS	VAXcluster
DIBOL	RSX	VAXELN
EduSystem	RT	VAXstation
IAS	rtVAX 1000	VMS
MASSBUS	ThinWire	VT
MicroVAX	ULTRIX	Work Processor
		digital™

MLO–S913

This document was prepared using VAX DOCUMENT, Version 1.1.

Contents

FIGURES

TABLES

Preface

Manual Objectives

The *VAXELN Internals Manual* describes the data structures, algorithms, and internal components of Version 4.0 of the VAXELN Kernel and a number of its associated subsystems. The detailed information presented in this manual should help VAXELN system designers and programmers understand how a VAXELN system functions and how best to take advantage of certain features of the VAXELN software.

There is no guarantee that any data structure or subroutine described in this manual will remain the same in subsequent releases of the VAXELN software. Therefore, the ultimate authority on how the kernel or any other component of the system works is the source code for that component.

Intended Audience

This manual is for VAXELN system architects and programmers who understand VAXELN programming and the VAX architecture and assembly language in depth and who need to understand the internal implementation of the VAXELN Kernel and its associated subsystems.

Structure of This Document

The *VAXELN Internals Manual* contains the following chapters and appendixes:

- Chapter 1, Overview: The Role of the VAXELN Kernel, introduces the internals of the VAXELN Kernel.
- Chapter 2, The VAXELN System Image, describes the structure and function of a VAXELN system image.
- Chapter 3, System Bootstrap, Kernel Initialization, and Application Start-Up, describes the kernel's initialization and the start-up of applications jobs.
- Chapter 4, Job and Process Creation and Deletion, describes the data structures and operations that support job and process creation and termination.
- Chapter 5, Software Interrupts, Kernel Synchronization, and Time Support, describes the data structures and operations that support the software interrupts, kernel synchronization, and time services.
- Chapter 6, Condition Handling, describes the data structures and operations that enable the kernel to detect, deliver, and handle exceptions, asynchronous exceptions, and software conditions.
- Chapter 7, Error and Event Reporting, describes the data structure and operations that support error logging, machine-check handling, and bugchecks.
- Chapter 8, Kernel Procedures and Procedure Dispatching, describes how the kernel dispatches calls to its procedure code.
- Chapter 9, Memory Management and Dynamic Allocation, describes the data structures and operations that support virtual address translation and the allocation of physical and virtual memory.
- Chapter 10, Kernel Objects and Their Management, describes the data structure and operations that support the creation, use, and deletion of kernel objects.
- Chapter 11, Job and Process Synchronization, describes the data structures and operations that enable processes to synchronize their execution by waiting for kernel objects.
- Appendix A, Kernel Parameters and Data, describes the System Builder parameters and dynamic data used by the kernel.

- Appendix B, Kernel Data Structures, summarizes the data structures manipulated by the kernel.

Associated Documents

In addition to the *VAXELN Internals Manual*, the VAXELN documentation set contains the following guides and reference manuals:

- *VAXELN Release Notes*. These notes describe enhancements made to the last version of VAXELN, current restrictions, and additions to documentation.
- *VAXELN Installation Guide*. This manual describes the VAXELN installation procedure.
- *Introduction to VAXELN*. This manual surveys the features of the VAXELN Toolkit, introduces VAXELN programming concepts and practices, and illustrates the design, coding, building, and running of a sample VAXELN application.
- *VAXELN Development Utilities Guide*. This manual explains how to use the VAXELN Host System Software and other utilities to develop and run VAXELN applications.
- *VAXELN Run-Time Facilities Guide*. This manual is a guide to using the VAXELN Run-Time Software.
- *VAXELN Application Design Guide*. This manual contains sample VAXELN applications for use and reference in designing VAXELN applications.
- *VAXELN Pascal Language Reference Manual*. This manual describes the components of the VAXELN Pascal language and the Pascal program-development process.
- *VAXELN Pascal Run-Time Library Reference Manual*. This manual describes the VAXELN Pascal interface to kernel and utility procedures.
- *VAXELN C Run-Time Library Reference Manual*. This manual is a guide to C programming under VAXELN and describes the C interface to kernel and utility procedures.
- *VAXELN FORTRAN Run-Time Library Reference Manual*. This manual is a guide to FORTRAN programming under VAXELN and describes the FORTRAN interface to kernel and utility procedures.
- *VAXELN Guide to DECwindows*. This manual describes how to program and build dedicated, real-time applications that use VAXELN and DECwindows software in concert.

- *VAXELN Messages Manual*. This manual describes the messages issued by the VAXELN Toolkit. Each message description includes an explanation and, where applicable, a suggested recovery procedure.
- *VAXELN Master Index and Glossary*. This index and glossary include index entries and glossary terms for the manuals in the VAXELN documentation.

The following documents are relevant to a discussion of VAXELN internals and will enhance your understanding of the information presented in this manual:

- *VAX Procedure Calling and Condition Handling Standard*. This document, part of the VMS documentation, defines the standards for all external interfaces that can be called from Digital's supported, standard system software and all external procedure calls generated by standard Digital language processors.
- *VMS Linker Reference Manual*. This manual describes how the VMS Linker works and how to use it.
- *VAX Architecture Handbook*. This handbook provides a detailed technical description of the VAX architecture, including virtual addresses, data representations, instruction formats, addressing modes, interrupt schemes, and memory management.
- *VAX Hardware Handbook*. This handbook provides general technical information for the VAX hardware product line. It includes descriptions and specifications for the VAX processors, data storage systems and devices, VAXcluster configurations, and communication products.
- *VAX/VMS Internals and Data Structures*. This book describes in detail the operation of the VMS operating system executive and its associated subsystems.

Conventions

The following conventions are used throughout this manual:

Convention	Meaning
kernel	The term kernel refers to the VAXELN Kernel, the real-time executive software that enables VAXELN systems to execute. In some usages, the term refers to the kernel's image file, such as QBUSKER.EXE. In others, it refers to the characteristics or actions of a procedure, subroutine, or service routine that resides within the kernel image portion of a VAXELN system image and is mapped into system virtual address space at run time.
module	The term module refers to a VAXELN system source file. A module name that appears without a facility name prefix ([*facility*]) is assumed to be a part of the [KERNEL] facility. Modules that are not part of the kernel are further identified by their facility names; for example, module [DEBUG]LOCALNUC refers to a module that is part of the VAXELN debugger.
data structures	Unless otherwise noted, illustrations of data structures and memory are aligned on longword boundaries. Furthermore, the lowest addresses appear at the top right portion of the diagram and increase toward the left and bottom.
<15:5>	Bit fields are shown between angle brackets. The upper and lower bounds of the field are shown from left to right, separated by a colon. For example, the notation <15:5> represents a bit field that contains bits 5 through 15 in a word.

Convention	Meaning
Lists	The following conventions apply to lists: • In lists that convey information with no order or hierarchy, list elements are indicated by bullets (•). Sublists without hierarchy are indicated by dashes (—). • In lists that convey ordered operations, list elements are numbered. Sublists that indicate ordered operations are lettered. • In numbered lists that relate to numbered items in a figure, element numbers are enclosed in circles, for example, ❶.
decimal notation	Numeric values are represented in decimal notation unless otherwise noted.
. . .	Vertical ellipsis points in a figure or example indicate that unnecessary or repetitive information has been omitted.
UPPERCASE characters	VAXELN and language-specific reserved words and identifiers are printed in uppercase characters, except for reserved words in C, which is a case-sensitive language. These terms are presented in **bold** lowercase characters.

Chapter 1

Overview: The Role of the VAXELN Kernel

The *VAXELN Internals Manual* describes the essential data structures and operations of the VAXELN real-time executive software, called the VAXELN Kernel. Viewed by the VAXELN programmer, the kernel presents an interface composed of a set of objects, representing such real-time entities as devices and synchronization points, and a set of procedures for creating and manipulating those objects.

The VAXELN programmer requires no special knowledge of these objects and procedures beyond their functional characteristics and calling sequences, as described in the user documentation, to create sophisticated real-time applications entirely in high-level languages. One of the major advantages of the VAXELN Toolkit is just this packaging of complex operations in a simple programming and development interface.

This manual presents information beyond that strictly required by VAXELN programmers in the belief that detailed technical knowledge of the kernel allows users to perform the following kinds of tasks more effectively:

- Design and code real-time and dedicated applications for maximum performance
- Tune System Builder parameters for optimized use of system resources
- Understand run-time behavior and troubleshoot problems
- Match processor and peripheral capacities to application requirements

- Understand overall system operation and interpret source code listings
- Modify and customize system components

To this end, the *VAXELN Internals Manual* focuses almost exclusively on the data structures and operations of the kernel, to the exclusion of other system components, such as common device drivers, run-time libraries, and file service. The System Builder and debugger utilities are discussed inasmuch as they significantly support or interact with the kernel. This focus on the VAXELN Kernel emphasizes its central role in the VAXELN programming environment.

This first chapter begins the close examination of the kernel by providing an overview of its role in supporting the execution of a VAXELN application. The following sections survey the structure and operation of the kernel (Section 1.1), the functions provided by the kernel (Section 1.2), and the kernel's relationship to the VAX architecture and hardware (Section 1.3). This overview material, however, can in no way substitute for a thorough knowledge of VAXELN derived from the user documentation and actual VAXELN programming experience. The *VAXELN Run-Time Facilities Guide* provides a starting point for preparation for the profitable use of this manual.

The remaining chapters of the manual fall roughly into three parts. Chapters 2 through 4 describe the foundation on which VAXELN applications execute: the structure and function of the system image, system initialization, and the creation of jobs and processes. Chapters 5 through 10 describe the kernel's control mechanisms: internal synchronization and timing, condition-handling logic, error and event reporting, kernel procedure dispatching, memory management, and kernel object management. A chapter on scheduling will join this part in a later edition. Chapter 11 returns attention to the job and process level by describing synchronization mechanisms. Chapters on device handling and communication will round out this part in a later edition.

1.1 Kernel Structure and Operation

The VAXELN Kernel was designed to provide a core of real-time functionality, such as multitasking and synchronization, that takes full advantage of the VAX architecture with a minimum of system overhead. The kernel image and a program provided by the user can potentially constitute the entire application system. More elaborate services, such as a file system or a terminal driver, can be added as they are needed. This leaves the programmer free to decide when the

cost of adding a service is too high. The core of functionality remains and can be used to build additional features the programmer requires.

The general philosophy behind the structure and operation of the kernel can be summarized as follows:

- Provide a small, efficient core of real-time functions. This allows applications to run on small memory targets and eliminates much of the overhead associated with conventional operating systems.
- Provide a simplified approach to real-time programming through the use of kernel objects and procedures to represent real-time entities and operations. This approach gives programmers full access to the VAX hardware from high-level languages, a distinct advantage in productivity and maintainability, especially for device-handling applications.
- Exploit fully the VAX hardware to provide assistance in accomplishing system tasks. For example, the kernel employs the VAX memory management hardware to perform virtual address translation and memory protection.
- Keep system and user components memory-resident at all times. Eliminating memory paging simplifies memory management and scheduling, makes address-translation times fast and predictable, and allows VAXELN systems to run in a diskless environment.

The kernel further increases its efficiency by not competing with user processes for system resources and processor cycles. The kernel itself creates no processes for its own use; that is, it does not function as an independently executing monitor program. Rather, it is a highly structured collection of data, procedures, and interrupt and exception service routines that execute when they are called from user code or activated in response to process or system events. The kernel runs exclusively at boot time to initialize the system and activate the user's application. Thereafter, the kernel is strictly driven by events. Its code is executed when kernel procedures are called, when hardware devices generate interrupts, or when processor or device controller microcode detects an error or anomaly.

1.2 Functions Provided by the Kernel

This section summarizes the major functions provided by the VAXELN Kernel. These kernel functions fall roughly into three classes:

- Initialization mechanisms: the system image (Section 1.2.1); system initialization (Section 1.2.2); and job and process creation (Section 1.2.3)
- Control mechanisms: software interrupts, kernel synchronization, and time services (Section 1.2.4); condition handling (Section 1.2.5); error and event reporting (Section 1.2.6); kernel procedure dispatching (Section 1.2.7); memory management (Section 1.2.8); object management (Section 1.2.9); and job and process scheduling (Section 1.2.10)
- Job-level support: job and process synchronization (Section 1.2.11); device handling (Section 1.2.12); and interjob communication (Section 1.2.13)

As the following sections indicate, these functions are also the subjects of the remaining chapters of this manual.

1.2.1 VAXELN System Image

A VAXELN system image is a file created by the System Builder, which combines the appropriate kernel image and data, system software, user programs, and shareable images into an image that can be loaded and executed on a target VAX processor. In general, the System Builder lays the groundwork for the efficient operation of the kernel by arranging system components and data to be readily accessible at run time. For example, the System Builder resolves program references to locations in shareable images so that this operation does not have to be performed during the creation of a VAXELN job.

Data stored as part of the image records the user's input to the System Builder's menus. When the image is booted on the target processor, the kernel establishes characteristics of the run-time system based on certain of these menu entries. For example, it creates the number of system pool blocks and page table slots that the user requests on the System Characteristics Menu.

The system image contains an image header (for some booting methods); the kernel image itself; program images, containing global data and code; the device list, which contains a series of device descriptions; and the shareable images against which the programs in the system image were linked. Chapter 2 describes the components of a system image and their function in an executing system.

1.2.2 System Initialization

Execution of a VAXELN system is initiated by the VAX processor's bootstrap program, called VMB, which loads the system image into physical memory and transfers control to the kernel. The kernel then begins executing its initialization sequence to establish essential data structures, create system virtual address space, enable VAX memory management, configure I/O adapters, and activate system and application programs to run as VAXELN jobs.

Chapter 3 describes these stages of system initialization and illustrates the structure of system virtual address space.

1.2.3 Jobs and Processes

Under VAXELN, most code is executed by entities called processes. A VAXELN process is defined by its hardware and software context. The hardware context is the set of processor registers defined by the VAX architecture. The software context includes some information unique to the process's execution and some that the process shares with other processes that are executing portions of the same program image. Together, these processes are called a job.

A VAXELN job represents the activation of a program image by the kernel and a collection of processes. A job is not itself an executable entity; rather, it is a set of data structures used to manage the creation, resources, and scheduling of its processes. A job's collection of processes contains a single master process and zero or more subprocesses to execute the program's code. The master process, which is created as a part of job creation, executes the program's main code, beginning at its transfer address. The master process can in turn create subprocesses to execute other procedures or functions in the program image. Each process can then execute independently of all other processes. When the master process terminates, the job and all its subprocesses are deleted from the system.

All the processes associated with a job share the same P0 virtual address space (through a single P0 page table). During job creation, the kernel maps the program image — its global data and code — into this P0 region. Also stored in this shared region are the job's program arguments, its dynamic heap, and its message buffers. For each process in a job, the kernel creates a private P1 virtual address space (through a unique P1 page table). This address region maps a process's stack space, which can be used to hold local data, procedure call frames, exception information, and debugger context data.

Chapter 4 describes the data structures and operations associated with the creation and deletion of jobs and processes. This chapter also describes the structure and use of job and process virtual address space.

1.2.4 Software Interrupts, Kernel Synchronization, and Time Services

The kernel exploits VAX hardware resources to manage services such as job rescheduling, synchronized access to data, and system timekeeping. The software interrupt mechanism allows the kernel to perform necessary services only as the need arises; for example, job rescheduling is initiated through an interrupt — the kernel has no need to monitor the scheduling data base periodically in search of idle jobs. Synchronized access to the kernel's data base is enforced through the use of VAX interrupt priority levels (IPLs) and multiprocessor spinlocks, which are based on VAX interlocked instructions. The VAX interval clock enables the kernel to maintain the system time by interrupting with every clock interval so that the time can be updated by the appropriate interval.

Chapter 5 describes the kernel's use of these hardware-based mechanisms.

1.2.5 Condition Handling

VAXELN delivers conditions — hardware exceptions, software conditions, and asynchronous exceptions — to processes according to the VAX Procedure Calling and Condition Handling Standard. This standard defines condition-handling data structures — the signal and mechanism arrays — and the ways in which a condition handler is located and can respond to a condition. In addition, VAXELN exploits the VAX hardware's support for asynchronous system traps (ASTs) to deliver signals asynchronously to a process's execution.

Chapter 6 describes condition handling under VAXELN.

1.2.6 Error and Event Reporting

The VAXELN error-logging subsystem enables the occurrences of processor, bus, and device events and errors to be recorded in a local or remote file for later analysis with the VMS Error Log Utility. Some of these errors, such as machine checks, require the system to bugcheck — to be shut down in an orderly manner. Chapter 7 describes the kernel's error-logging, bugcheck, and machine-check mechanisms.

1.2.7 Kernel Procedure Dispatching

VAXELN processes manipulate kernel objects and control their execution by calling kernel procedures. Each of these procedures has a public entry point, called its kernel vector, at the beginning of the kernel image. The procedure's vector contains the VAX instructions required to dispatch execution to the actual procedure code, elsewhere in the kernel image. Most vectors place the caller's process into kernel access mode by using the CHMK (Change Mode to Kernel) instruction. This enables the procedure to execute privileged instructions and alter kernel data structures. Other vectors simply dispatch execution with subroutine or branch instruction, so that the procedure executes in the access mode of the caller. When a procedure has completed, control returns to the vector, which then returns the procedure's results and completion status to the caller.

Chapter 8 describes the structure of these different kernel vectors, how they dispatch control to kernel procedure code, and how they return status values to their callers.

1.2.8 Memory Management

VAXELN employs the VAX memory management hardware to perform virtual address translation using page tables established during system initialization and job and process creation. Translation is simplified under VAXELN, because VAXELN systems are always entirely resident in physical memory. Therefore, no paging from a disk is required before a page of memory can be accessed after address translation.

The kernel's memory management is based on a simple data base comprised of bitmaps, which record the allocation state of every page of physical and virtual memory in the system. The use of bitmaps minimizes the amount of memory devoted to memory management itself and simplifies the algorithms used to allocate and deallocate physical and virtual memory. Bitmaps are also used to control the allocation of a fixed number of P0 and P1 pages tables.

The kernel divides a portion of system memory into fixed-length blocks called the system dynamic pool. These pool blocks are used in the creation of kernel objects. Creating an object removes a block from the available pool; deleting an object returns the block to the pool.

Chapter 9 describes the kernel's memory management data structures and operations. The allocation of system pool blocks is also described.

1.2.9 Object Management

Kernel objects provide VAXELN programmers with a simple way of coding real-time operations such as synchronization, communication, and device control in their applications. The kernel objects — area, device, event, message, name, port, process, and semaphore — allow otherwise complex programming operations to be carried out with simple calls to kernel procedures.

When an object is created, the kernel assigns it a unique identifier. When an object is used in a procedure, the information encoded in the identifier allows the kernel to locate the address of the object in a table. The kernel manages all objects except ports with the context of a job. This means that an object identifier can be shared among all the processes in a job; their object addresses are stored in the job's private address table. Port objects, which can represent message destinations across a local area network (LAN), are managed on a systemwide basis; therefore, an identifier for a port is valid for the whole system (and LAN) and contains the network address of the node on which the port was created. The addresses of port objects are stored in a single systemwide table.

When the kernel creates an object, it removes a block from the system pool, marks it with the object type, initializes the object, assigns it an identifier, stores the address of the object in a table, and returns the identifier to the calling process. When deleting the object, the kernel disassociates the object from any processes that might have been using it, returns the block to the system pool, and removes the address of the object from the table.

Chapter 10 describes the data structures and operations that enable the kernel to create, manage, and delete kernel objects. The function of individual objects, however, is described separately. For example, the function of the event object, used for synchronization, is described in Chapter 11, Job and Process Synchronization.

1.2.10 Job and Process Scheduling

A process is selected to execute by the kernel's scheduling mechanism. Scheduling under VAXELN is based on a simple scheme called preemptive priority-based scheduling. This means that the process with the highest priority runs before any lower-priority process can run. Priorities are assigned on a job and process basis. The programmer assigns a job a priority based on its relative importance in the system. Within each job, the processes created are then assigned priorities based on their importance to the fulfillment of the job's mission. The scheduler selects a process to run on the basis of its combined job and process priorities.

In general, the highest-priority process in the highest-priority job runs until it waits for an event or resource or until a process with a higher combined priority becomes eligible to run. In the first case, the process postpones execution so that it may synchronize with another process. In the second case, the process is preempted from execution. The scheduler preempts a process only when a higher-priority process must run; preemption based on execution time never occurs. Programmers, however, can build time-based preemption into their applications using simple synchronization techniques.

The kernel maintains a scheduling data base that reflects the scheduling state and priority of every job and process in the system. Much of this data is summarized in simple bit masks, which allow the kernel to scan the data base quickly in search of the highest-priority job and process.

Job and process scheduling is not discussed in detail in this edition of the *VAXELN Internals Manual*.

1.2.11 Job and Process Synchronization

Synchronization enables a process to coordinate its execution with real-world events, such as device interrupts, and with other processes. Synchronization techniques can be used to cause events to occur in the correct order or to ensure exclusive access to shared data. Under VAXELN, synchronization points in an application are represented by kernel objects, and processes synchronize by waiting for these objects to change state. Area, device, event, port, process, and semaphore objects can be used to develop the synchronization schemes that real-time applications require. Chapter 11 describes the kernel objects and other data structure used for synchronization and the operations provided by the kernel to allow processes to wait for those objects to change their states.

1.2.12 Device Handling

Under VAXELN, an I/O channel to a device is represented by a device object. Creating a device object in a device driver program associates the device's interrupt vector in the system control block (SCB) with an interrupt service routine (ISR) and maps the device's control/status registers into system virtual address space accessible to both the ISR and the device driver. The driver synchronizes access to this device communication region by waiting for the device object to be signaled from the ISR. When the device interrupts, the kernel dispatches execution to the ISR to service the interrupt, which may involve moving data to or from the communication region. Once the interrupt is serviced, the ISR can inform the driver by signaling the device object, which allows the waiting driver to continue execution.

Drivers can create multiple device objects to represent separate I/O channels for the same device. For example, a driver that controls a serial-line device might create one device object for an input line and another for an output line. Each device function, then, can have a separate ISR and driver process to service its operation. Device objects also support polled I/O, a technique used for drivers that control devices without the use of interrupts.

The kernel's support for device handling is not discussed in detail in this edition of the *VAXELN Internals Manual*; however, the kernel data structures that support device handling — the adapter control block, the device object, and the interrupt dispatch block — are illustrated and described in Sections B.2, B.5, and B.9, respectively.

1.2.13 Interjob Communication

Independently executing VAXELN jobs can communicate with one another using areas and messages. Area objects represent a contiguous region of physical memory that can be accessed, using virtual addresses, by multiple jobs in a VAXELN system. Access to the shared area is controlled through a binary semaphore built into the area control structures. Jobs can use the area as an efficient way to share data or to synchronize execution using only the area's semaphore. The kernel's support for shared areas is not discussed in detail in this edition of this manual; however, the data structures that support areas — the area control block and the area object — are described in Sections B.1 and B.3, respectively.

VAXELN messages can be sent and received between processes, jobs, and DECnet nodes in a local area network. A message is a data buffer in physical memory represented by a message object. Messages are transmitted to queues called ports. Ports can be given names that are known on a local node or throughout a local area network. Ports can be connected in logical links called circuits. Communication over circuit connections uses a protocol that guarantees the orderly delivery of messages over an intact link.

To send a message to a local port, the kernel simply unmaps the data buffer from the sending job's virtual address space and queues it to the destination port. When the process waiting for a message in the port receives the message, the kernel maps the message buffer into the receiving job's virtual address space. To send a message to a remote port, the kernel uses the local-sending method to transmit the message to the local node's network service. The network service then uses its datalink driver to route the message across the physical network link to the remote node. At the destination node, the local network service sends the message locally to the kernel, which then queues the message into the destination port.

The kernel's support for message passing is not discussed in detail in this edition of the *VAXELN Internals Manual*; however, the kernel data structures that support message passing — the message, name, and port objects — are illustrated and described in Sections B.11, B.12, and B.18, respectively. Two structures involved in circuit connections and name support — the network connection message and the name service message — are described in Sections B.13 and B.14, respectively.

1.3 Notes on the Kernel and the VAX Hardware

The VAXELN Kernel takes ready advantage of many features provided by the VAX processor. The richness and advantages of the VAX architecture are documented in detail in the *VAX Architecture Reference Manual* and have been used extensively by other VAX-based software executives, such as the VMS operating system (see *VAX/VMS Internals and Data Structures*). The following notes highlight features of the VAX architecture and hardware used by the VAXELN Kernel:

- System bootstrap. VAXELN employs the generic VAX bootstrap program called VMB to load a system image over a network link or from disk, tape, or read-only memory. Once VMB has loaded the image into main memory, it transfers control to the kernel to begin system initialization.
- Memory management. VAXELN employs the VAX memory management hardware to perform virtual address translation. Because the kernel does not require support for memory paging, certain aspects of this mechanism, such as the page-fault mechanism, are unused.
- Protection mechanisms. The VAX memory management and protection scheme are used to protect code and data used by the kernel and kernel-mode programs from user-mode programs. Note that VAXELN requires only two of the four VAX access-control modes: kernel and user. Kernel procedures, ISRs, exception service routines, and system jobs, such as device drivers usually execute in kernel mode. The user can specify the access mode of a program by using the System Builder. Implicit protection is built into special instructions that can only be executed from kernel mode, such as MTPR (Move to Privileged Register), LDPCTX (Load Process Context), and HALT.
- Exceptions, interrupts, and the REI (Return from Exception or Interrupt) instruction. The VAX exception and interrupt mechanisms are critical to the operation of the kernel. The exception mechanism transfers control to a specific service routine when the hardware detects a specific anomaly during the execution of an instruction. The interrupt mechanism transfers control to a specific service routine when a software- or hardware-generated interrupt occurs. The REI instruction provides a common exit path for both mechanisms. REI also offers the only valid means of returning access mode from kernel to user mode.

- Interrupt priority level (IPL). The kernel raises the processor's IPL to block interrupts of equal or lower levels. IPL is also elevated to synchronize access to kernel data. The assignment of various hardware and software interrupts to specific IPL values establishes an order of importance to the interrupt services that the kernel performs.
- Asynchronous system traps (AST). The VAX AST mechanism allows the execution of a process to be diverted asynchronously using the ASTLVL processor register and the REI instruction. VAXELN uses this technique to deliver called asynchronous exceptions, which supply services such as the process-quit signal and power-failure notification.
- Procedure-calling mechanism. The VAX general-purpose calling mechanism is the primary path into the kernel from system and user programs. The kernel's services are coded as VAX procedures, so that they can potentially be called from any higher-level VAX language.
- Process structure. The VAX architecture defines a data structure called a hardware process control block that contains copies of all a process's registers when a process is not executing. Under VAXELN, this structure is commonly referred to as the hardware context block, or PTX. When a process is selected for execution, the contents of its PTX are copied into the actual registers inside the processor with a single instruction, LDPCTX. A corresponding instruction, SVPCTX, saves the contents of the general registers when the process is removed from execution.
- Process and system context. Normal execution under VAXELN takes place within the bounds of a process, a state called process context. Most kernel procedures and exception service routines execute in this context.

  Some portions of the kernel, however, execute outside the context of a specific process. This limited-context state is called system or interrupt context, because the only stack available in this context is the systemwide interrupt stack. Interrupt service routines are the most common code to execute in system context. Portions of the initialization sequence execute in this state because no process yet exists. The scheduler also executes on the interrupt stack between the time that it removes one process from execution and places another into execution. Most kernel procedures require process context for execution and therefore cannot be called from code that may execute outside of process context, such as device ISRs.

- Multiprocessing. Certain VAX products, such as the VAX 8000 and VAX 6000 series, can provide multiprocessing configurations. The kernel takes advantage of these configurations with a mechanism called tightly coupled symmetric multiprocessing. This scheme provides one copy of the kernel and its data in memory shared by all processors. Synchronization techniques built into the kernel ensure the integrity of system data, and the scheduling software enables different jobs to execute concurrently on the multiple processors.

  Another scheme, called closely coupled symmetric multiprocessing, allows multiple KA800 single-board processors to be linked together with a VAX 8000- or VAX 6000-series processor over an I/O bus. Each KA800 processor runs a private VAXELN system and can communicate with the other processors through shared memory and VAXELN messages. The primary processor in the system can run under VAXELN or under VMS using the VAX RTA software. The *VAXELN Run-Time Facilities Guide* provides more information on these multiprocessing configurations.

Chapter 2

The VAXELN System Image

A VAXELN application, or part of a distributed application, exists on a target VAX processor as a system image. A VAXELN system image is not in fact a VAX executable image; rather, it is a composite of system and user code and data preceded by an image header appropriate for the system's intended boot method. A system image may contain many individual executable and shareable images accompanied by blocks of system and program information required by the VAXELN Kernel. These images and information are placed in the system image by the VAXELN System Builder.

Since all nondynamic components of a VAXELN system reside in the system image, familiarity with its structure is basic to understanding the dynamic operation of the kernel. This chapter first describes how the System Builder creates the system image (Section 2.1) then describes each major element in the image:

- The image header, which enables a system to be booted from certain devices (Section 2.2)
- The kernel image, which contains the kernel's data and code (Section 2.3)
- Program images, which contain code and data for system and user programs (Section 2.4)
- The device list, which contains information about the devices configured for a system (Section 2.5)
- Shareable images, which contain code and data that can be shared among the programs in a system image (Section 2.6)

2.1 Role of the System Builder

The VAXELN System Builder is a utility that runs on the host development system in response to the EBUILD command. The main function of the utility is to construct the system image specified by the user. The System Builder accepts user input interactively through a series of menus and from a data file that contains information generated during earlier interactive sessions. The System Builder potentially generates three output files:

- The system image file containing the system to be executed on the target computer
- A map file describing the content of the system image
- A data file recording the menu settings made during an interactive session

In creating a system image, the System Builder performs, to some degree, the roles that the SYSGEN utility, the linker, and the image activator play under the VMS operating system.

The System Builder accepts menu input specifying the following aspects of a VAXELN system:

- Target processor. The identity of the target processor determines which kernel image the system image will contain.
- System characteristics. These menu entries determine global aspects of the completed system, including the availability of the debugger, the console, and VAX instruction emulation; the boot method for the system; the maximum number of jobs and subprocesses that can exist simultaneously in the system and the largest amount of virtual memory available to those jobs and processes; the number of system pool blocks and message ports available; and the size of the interrupt stack and dynamically allocated system memory.
- Network characteristics. These entries determine the nature of a system's participation in a local area network.
- Program descriptions. These entries specify the name and characteristics of a program image that will be incorporated into the system image. Characteristics include whether the program will be started up automatically at system initialization, whether the debugger will take initial control, the execution mode of the program, the priorities assigned to the job and processes that will execute the

program's code, and any textual arguments passed to the job as it is created.

- Device descriptions. These entries determine the characteristics of a device that can be accessed from the executing system, such as the name of the device, its register and interrupt vector addresses, and its hardware interrupt priority.
- Terminal descriptions. These entries determine the characteristics of a terminal or other serial device that can communicate with the executing system, such as its controller type, baud rate, and parity.
- Console characteristics. These entries determine the setup of the console terminal attached to the target processor.
- Error log characteristics. These entries determine the destination of error log entries and the number of error log buffers available to the error logging service.
- DECwindows Server characteristics. These entries determine the configuration of the DECwindows Server on a VAXstation target.

The System Builder supplies defaults for many of these characteristics. In addition, the System Builder supplies program and device descriptions required by the system but not explicitly requested by the user. For example, if a user requests remote debugging capability but does not request the inclusion of the Network Service, the System Builder determines that the service is required, builds a program description for it, and includes its image in the system image.

In creating a system image, the System Builder takes the following steps:

1. Obtains the kernel image appropriate for the selected target processor
2. Creates a list of descriptors describing the programs to be included in the image
3. Creates a list of configuration records describing the devices to be included in the system
4. Creates a shareable image table describing the shareable images referenced by programs and other shareable images within the system
5. Copies the programs, program descriptors, device and terminal descriptors, shareable image table, and shareable images into the system image
6. Initializes parameter and data cells within the kernel

7. Resolves the program's references to shareable images by adjusting the referenced addresses to reflect the location of the shareable image within program or system address space, a process called address relocation
8. Writes out the completed system image and, optionally, a map file describing the system image, and an updated data file

The resulting system image is structured along the lines of Figure 2–1.

Table 2–1 briefly defines each element of the system image. These elements are described in greater detail in subsequent sections.

Figure 2–1: VAXELN System Image

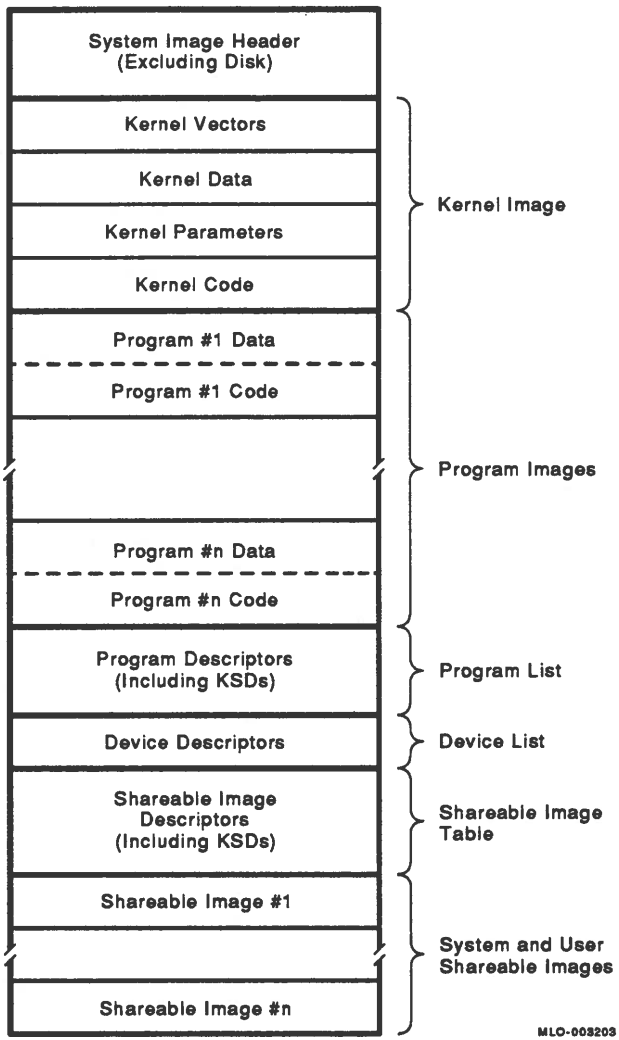


Table 2–1: Elements of a System Image

Element	Description
System image header	A page containing information used by the VAX VMB bootstrap program to load the system image into a processor's memory. The need for and type of header is determined by the boot method selected for the system. Systems booted from ROM or over the network require an image header; those booted from disk or tape do not.
Kernel vectors	Entry points for KER$ kernel procedures. These vectors transfer control to the location of the actual procedure code by executing a Change Mode to Kernel (CHMK) instruction.
Kernel data	Cells that hold global data elements for use by the kernel and its support routines, such as listheads, Boolean flags, and the system time.
Kernel parameters	Cells that hold values established by the System Builder that record menu settings and other systemwide values, including the size of the current system and the locations of program and device descriptors within the system image.
Kernel code	The executable code of the kernel. The code starts with the kernel's initialization sequence and contains, among other routines, the code for the kernel procedures whose entry points appear in the kernel vectors.
Program data and code	The data and executable code of the user's programs and support programs such as device drivers. (Programs linked against object libraries also contain the code and data of the routines they reference in those libraries.)
Program descriptors	A list of elements describing the programs loaded into the system image by the System Builder. Each program descriptor is accompanied by the text of parameters to be passed to the program and a series of kernel section descriptors (KSDs) that describe the program's image sections. Taken together, this information allows the kernel to create a job to execute the program's code.

Table 2–1 (Cont.): Elements of a System Image

Element	Description
Device descriptors	A list of elements describing the devices to be supported by the system.
Shareable image descriptors	A list of elements describing the shareable images loaded into the system image by the System Builder. Each descriptor is accompanied by a series of KSDs that describe the shareable image's image sections. The information in the descriptors is used in address relocation for dynamically loaded programs that reference shareable images. The KSDs enable the kernel to map writeable shareable images into a referencing program's address space.
Shareable images	The actual code and data for the shareable images loaded into the system image by the System Builder. In the case of VAXELN run-time libraries, the code is preceded by a block of transfer vectors through which the actual code of the run-time procedure is located. The images included are those referenced by the programs in the program list (and linked against shareable libraries), those specified on the Guaranteed Image List, the VAXELN console I/O routines, and VAX instruction emulation images selected on the System Characteristics Menu.

If the /MAP and /FULL qualifiers are specified on the EBUILD command line, the System Builder produces a map file describing the contents and layout of the system image. In a full system map, the following information is provided:

- The name of the system image file and the time of its creation.
- The name of the VAXELN Kernel image included and the starting address of the vector, parameter, and code blocks.
- Names of programs included in the system, including system programs (such as device drivers) that are not explicitly specified on the Program Description Menu. The characteristics of the program (for example, its mode and job priority) and its image sections (section type, base address, and size) are shown. If the program references any writeable shareable images — which will be mapped into the program's address space — those images are identified as well.

- Device descriptions reflecting the device entries created on the Device Description Menu and descriptions supplied by the System Builder.
- Terminal descriptions reflecting the terminal entries created on the Terminal Description Menu and showing the characteristics selected for each terminal.
- Names of shareable images implicitly or explicitly included in the system image. The descriptions show the image identification, whether the image is mapped into a referencing program's address space (that is, whether the image is writeable), and the type, base system virtual address, and size of the shareable image sections.
- Network characteristics, reflecting the entries on the Network Node Characteristics Menu.
- System characteristics, reflecting the entries on the System Characteristics Menu.
- DECwindows Server characteristics, reflecting the entries on the DECwindows Server Characteristics Menu.
- The size of the system image in pages and bytes.
- The System Builder command line.

Consulting a sample System Builder map can help illuminate the structure of the system image as discussed in the sections that follow.

2.2 System Image Header

The user selects the boot method for a VAXELN system on the System Characteristics Menu, selecting disk, ROM, or down-line loading. The menu selection determines whether the system image will have a system image header and the type of that header. Only Q-bus targets can be booted from ROM, using the MRV11 Q-bus module.

If a VAXELN system image is booted from ROM or over the network, the System Builder adds a one-page header to the start of the system image, preceding the kernel image. For ROM systems, a Q-bus ROM header is written; for network systems, a standard VMS image header is written. Systems to be booted from disk or tape require no image header.

The system image header supplies the information required by the bootstrap loader — such as the size of the image — to load the system image into the memory of the target computer. Down-line loading of a target is performed using the DECnet maintenance operation protocol (MOP). The down-line loading sequence (initiated by a MOP message requesting a program load from the target) expects to find an ordinary VMS image header.

The bootstrap sequence on a Q-bus target will boot the VAXELN system from the MRV11 PROM module if it finds a special ROM footprint on a 4K-byte boundary in the target's memory. This unique bit pattern is provided by the ROM image header supplied by the System Builder.

Systems that boot from disk or tape devices require no image header. Instead, the system must be copied contiguously to the [SYS0.SYSEXE] directory on the boot device, an operation performed by the COPYSYS command procedure in the ELN$ directory of the host system. As the system image is copied to the boot device, its name is changed to SYSBOOT.EXE, the name of the VAX secondary bootstrap program.

2.3 Kernel Image: Vectors, Data, Parameters, and Code

As shown in Figure 2–1, each kernel image consists of four elements: vectors, data, parameters, and code. Early in its operation, the System Builder copies the appropriate kernel image from the ELN$ directory to the system image file. As the kernel image is copied, the System Builder strips off the original image header created by the VMS Linker.

The selection a user makes on the Select Target Processor Menu determines which version of the VAXELN Kernel will be written to the system image by the System Builder. Table 2–2 shows the versions of the kernel and the processors supported by each one.

Table 2–2: VAXELN Kernel Images

Image	Processors Supported
4NNKER	MicroVAX 2000, VAXstation 2000, VAXstation 3100
QBUSKER	MicroVAX I, MicroVAX II, MicroVAX 3000 Series, VAXstation II/GPX, VAXstation 3200, VAXstation 3500, KA620
UBUSKER	VAX–11/725, VAX–11/730, VAX–11/750

Table 2–2 (Cont.): VAXELN Kernel Images

Image	Processors Supported
6CCKER	VAX 6000 Series
8SSKER	VAX 8200 Series
8NNKER	VAX 8500 Series, VAX 8700, VAX 8810
800KER	KA800 Single-board computer (VAX/RTA)
MP8800KER	VAX 8800, VAX 8820–N

In general, each kernel supports a class of target processors. The criterion that differentiates a class may be bus architecture, as in the case of the QBUS and UBUS kernels, or it may be processor-specific differences within a bus architecture, as in the case of several of the VAXBI-based versions of the kernel, such as 8NNKER.EXE and MP8800KER.EXE, which differ in their support of multiprocessing.

The different kernels are created when the kernel is assembled, by the selective inclusion of processor- and/or bus-specific initialization, error-logging, and machine-check modules. At run time, hardware-dependent routines are executed through branches to generic subroutine entry points; the code that appears in those subroutines depends on which processor-specific module was included at kernel creation. Processor dependence is largely avoided in the kernel's common code to minimize the kernel's need to determine the processor type at run time.

Each kernel image file is accompanied in the ELN$ area by a linker map file produced when the kernel image was linked. The linker map file describes the sizes, locations, and attributes of the kernel's program sections — one section each for vectors, data, parameters, and code. Also listed in the map file are the definitions of the symbols and labels that appear throughout the source code for the kernel.

2.3.1 Kernel Vectors

The kernel vectors represent the first stage in dispatching calls to kernel procedures. The vector block occupies the first two pages of the kernel image; when a VAXELN system begins execution and the kernel is mapped to S0 space, the vector block begins at system virtual address 80000000_{16}. The code for the kernel vectors spans three source modules, SYSVECTOR, VECTORTAB, and VECTOREND. (You can

determine the exact starting addresses of the various sections of the kernel by examining the appropriate kernel linker map file.)

The vector block contains a series of quadword-aligned entry points to the kernel procedures. Each vector consists of the register entry mask for the procedure, an instruction to transfer control to the procedure, and a variable number of instructions to effect the return of values and control to the caller. The following code fragment shows the vector for the KER$RECEIVE procedure, a typical instance:

```
KER$RECEIVE::
        .WORD   ^XFFC           ; entry mask for registers R2 to R11
        CHMK    #29             ; entry 29 in the CHMK dispatch table
        MOVL    R1,@8(AP)       ; return message object identifier
        MOVL    R2,@12(AP)      ; return address of message buffer
        MOVL    R3,@16(AP)      ; return size of message in bytes
        BRW     KER$RETURN_STATUS
                                ; check status and return to caller
```

The vector is entered by means of a CALL instruction, and the procedure entry mask causes registers R2 through R11 to be saved for the caller. Control is then transferred to the code for KER$RECEIVE through the CHMK instruction. After KER$RECEIVE has executed, control is returned to the instruction following the dispatch instruction, and the values returned by the procedures in registers R1, R2, and R3 are returned to the caller by means of pointers to writeable variables in the argument list. Finally, control branches to the local subroutine KER$RETURN_STATUS, which checks the status value returned by KER$RECEIVE, and returns control to the caller via a Return (RET) instruction. The dispatching of kernel procedure calls is described in detail in Chapter 8.

The vector block is identical for all versions of the kernel as well as all releases of the VAXELN software. This way, user programs need not be relinked to use different versions of the kernel or the VAXELN software; only rebuilding the system image is required. If new kernel procedures are added, their vectors are inserted at the end of the vector block, leaving the locations of previously existing vectors unaltered.

2.3.2 Kernel Data

The pages in the kernel image devoted to writeable data are a repository for any values that must be globally available to programs and routines executing throughout the system. This global data block, defined in module SYSTEMDAT, is mapped to a base system virtual address of 80000400_{16} and occupies the four pages of the kernel following the vector block. When the system image resides in a MicroVAX I PROM module, the kernel's initialization routine copies the data block in PROM to the first available pages in the physical memory of the target computer.

The values stored in the data block can be used as constants or as storage for values that are updated during execution. For example, the maximum virtual memory functions as a constant, whereas the system time is updated constantly during execution.

Table A–2 shows the names and uses of the cells within the data block. Kernel data is directly accessible from kernel routines. Individual elements in the data block can also be accessed from a user program if the items to be referenced are declared to be external at module level and the program is linked against RTL.OLB (the locations of the KER$ data items are defined in the KER$DATA object module within RTL.OLB).

2.3.3 Kernel Parameters

When the System Builder parses the input from a data file and an interactive menu session, it writes the resulting values and characteristics to an internal buffer. As the kernel image is being copied to the system image, the System Builder transfers the contents of the internal buffer to the kernel parameter block. During later processing, other values, such as the size of the system image, are written to the parameter block.

The kernel parameters, defined in module PARAMETER (and described in Table A–1), allow the System Builder to transmit information about the system image to the kernel. During execution, the kernel can consult values in the parameter block to determine, for example, the size of the system image and whether the console terminal is present in the system. Some parameters, namely those with the word "initial" in their names, are copied during system initialization from the read-only parameter block to the writeable data block. For example, the

parameter KER$GW_P0_INITIAL_SLOT_SIZE is copied to the data cell KER$GW_P0_SLOT_SIZE.

The parameter block, occupying approximately one-quarter of a page, is mapped to system virtual address $80000C00_{16}$. It is followed immediately by a read-only image section containing the executable code for the kernel.

2.3.4 Kernel Code

The actual code for the VAXELN Kernel follows the kernel parameter block in the system image. The kernel code starts at a system virtual address midway through the sixth page of the kernel image, approximately $80000C98_{16}$. At run time, the kernel code executes in the VAX kernel access mode.

The first code to appear is the system initialization sequence, in module INITIAL. The first part of the module contains a table that acts as a prototype for the system control block (SCB) that is built at a later stage of initialization. Following this table, the executable code for the kernel begins at the global label KER$START. When the primary bootstrap program transfers control to the start of the kernel image — the first page of kernel vectors — it encounters the following instruction:

```
BRW     KER$START
```

This instruction transfers control to the start of the initialization code, whence execution continues.

The remainder of the kernel code occupies approximately sixty pages — the exact size depends on the version of the kernel — and is divided into two image sections. The first of these sections contains nonprocessor-specific kernel procedures and routines that must fall within the range of the signed-word displacement (32K bytes forward or backward) used by the kernel procedure dispatcher (see Chapter 8, Kernel Procedures and Procedure Dispatching). The second image section contains internal processor-specific code that is entered directly through Jump (JMP) instructions and may therefore reside at addresses beyond the range of the word displacement required by the first image section.

Appearing throughout the first code section are the internal entry points and instructions for the kernel procedures whose public entry points appear in the vector block. The remainder of the code in the kernel contains the entry points and instructions for procedures and

subroutines used by the kernel for such operations as scheduling, resource management, device control, and condition handling.

2.4 Program Images

One of the chief roles of the System Builder is to incorporate the programs that the user specifies on the Program Description Menu into the system image. Other programs required for the system's operation but not explicitly specified by the user, such as debugger, error-logging, and device driver programs, must also be identified and included in the system image.

Processing user and system program images involves more than simply copying the images to the system image file. To include a program in the system image, the System Builder must perform the following tasks:

- Create a descriptor for each program to record information about the program such as its name, its execution mode, the size of its stack, its transfer address, its default job and process priorities, the message limit for its job port, and the location in memory of its program arguments. Some of this information comes from the Program Description Menu, some from the program image. Descriptors for system programs such as device drivers are created from information internal to the System Builder. The program descriptor is then inserted into a global list of descriptors for the entire system.
- Parse the arguments for each program as specified on the menu, place the text of the argument in a program parameter block, and insert the block into a list of parameter blocks for the program. The parameter blocks for a program are considered part of the program descriptor.
- Open the program image file, read each image section descriptor (ISD), translate each ISD into a kernel section descriptor (KSD), link the KSD to the program descriptor, and copy each image section to the system image file. As they are created, the KSDs are built into a block, and the address of the first KSD is stored in the program descriptor. If the program refers to any shareable images, steps are taken to load these images into the system image as well.

While the System Builder processes an image, it keeps a running account of the current virtual address within the system image at which each element being processed will appear. Thus, the run-time virtual address of any system structure is known to the System Builder and can be used to establish pointers within its own and the kernel's data structures.

The following sections describe the data structures and operations the System Builder uses in processing an image.

2.4.1 Data Structures for Image Processing

The System Builder analyzes and creates a number of data structures in the course of incorporating an executable image (program) into a VAXELN system image. Those structures fall into the following classes:

- The program descriptors and program list. Program descriptors are kernel data structures that record information about a program image included in the VAXELN system. All program descriptors are linked into a list called the program list, which the kernel uses to look up information about the images in the system.
- Structures within the VMS program image. To incorporate an executable image into the system image, the System Builder must analyze data structures written to the program image by the VMS Linker. The most significant of these structures is the image header, which contains information describing the image. Part of the header consists of image section descriptors (ISDs) that describe the makeup and virtual memory requirements of the program sections within the image. Using the ISDs, the System Builder creates a set of kernel section descriptors (KSDs) to describe the image to the kernel at run time.
- Kernel section descriptors (KSDs). Created from the ISDs within the program image header and associated with a program descriptor, KSDs describe the makeup and virtual memory requirements of a program's image sections as they exist within the system image. The kernel uses KSDs at run time to map a program image into a VAXELN job's virtual address space, where it can be executed.

The following sections describe these structures in more detail.

2.4.1.1 Program Descriptors and the Program List

Each program image in the system image — whether a user program or a system program — is represented by a program descriptor. The information stored in the descriptor during the system build is used at run time by the KER$CREATE_JOB kernel procedure to map the program into the job's virtual address space and to make the program arguments available to the job. The system start-up job also uses the descriptors to determine which programs must be created and run during system initialization.

The System Builder inserts the program descriptors into a list and copies the list to a location in the system image following the actual code and data for the programs. The system virtual address of the first program descriptor in the list is stored at the location KER$GA_PROGRAM in the kernel's parameter block; this value is copied to the location KER$GA_PROGRAM_LIST in the kernel data block during system initialization.

A program descriptor contains the fields shown in Table 2–3. A good portion of this information is derived from the descriptions entered on the Program Description Menu. Other items, such as the address of the first KSD and program parameters, are determined and recorded during the system build.

Table 2–3: Program Descriptor Fields

Field	Meaning
PRG$L_NEXT	The Address of the next program descriptor
PRG$W_CPU_MASK	The processor ineligibility mask (used to prohibit the program from running on certain processors in a multiprocessing system)
PRG$W_KERNEL_STACK	The size in words of the kernel stack
PRG$L_TRANSFER	The transfer address of the program in user address space
PRG$L_MESSAGE_LIMIT	The job port message limit
PRG$W_USER_STACK	The starting size of the user stack
PRG$W_JOB_PARAMETER	The offset to the first job parameter block
PRG$L_KSD	The address of the first kernel section descriptor in the list of KSDs

Table 2–3 (Cont.): Program Descriptor Fields

Field	Meaning
PRG$B_JOB_PARAMETER_COUNT	The count of job parameters
PRG$B_MODE	The program mode (kernel or user)
PRG$B_JOB_PRIORITY	The job priority
PRG$B_PROCESS_PRIORITY	The default process priority for the master process and subprocesses
PRG$B_OPTION_FLAGS	A bit field specifying other characteristics of the program, as follows:

Bit	Meaning When Set
PRG$V_AUTO_START	Start program at system start-up (that is, "run")
PRG$V_SEQ_INITIAL	Program must be initialized (sequential start-up, that is, "init required")
PRG$V_START_DEBUG	Start debugger when job is created
PRG$V_POWER_RECOVERY	Raise an exception during power-failure recovery
PRG$V_DELETED	Dynamically loaded program should be deleted from system when reference count reaches 0
PRG$V_DYNAMIC_PROGRAM	Program was dynamically loaded
PRG$V_DEBUG_WARM	Debugger is present but do not pass it control on job creation

PRG$W_REF_COUNT	The number of jobs executing this program's code if the program was dynamically loaded
PRG$T_NAME	The string descriptor containing the size of the program's name and the text of the name

As each program descriptor is created by the System Builder, the program arguments entered on the menu are separated and entered into separate parameter blocks. The fields in the parameter block are shown in Table 2–4. All the parameter blocks for the program are linked through the JPB$A_NEXT field. The byte offset from the base of

the program descriptor to the first parameter block is inserted into the PRG$W_JOB_PARAMETER field of the program descriptor, and the number of parameters is recorded in the descriptor as PRG$B_JOB_PARAMETER_COUNT.

Table 2–4: Job Parameter Block Fields

Field	Meaning
JPB$A_NEXT	The address of next job parameter block
JPB$L_SIZE	The byte count for the parameter string stored in this JPB
JPB$L_TOTAL_SIZE	The total byte count for all the parameter strings in all the JPBs for this program
JPB$B_TOTAL_COUNT	The number of program arguments, stored in the first JPB only
JPB$T_PARAMETER	The parameter string text, up to 100 characters in length

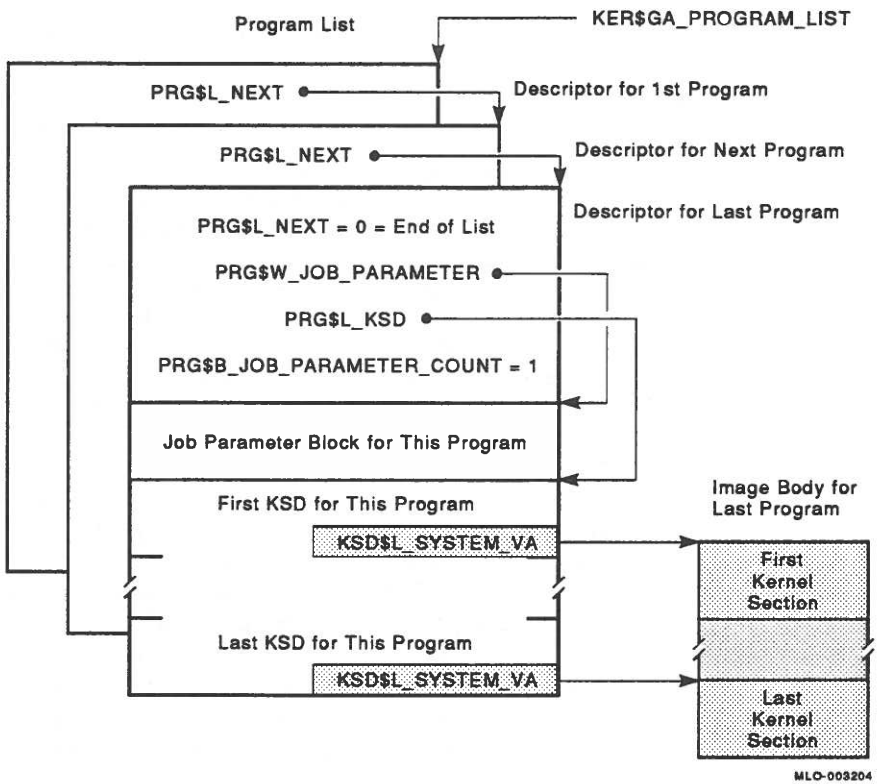
When a program is created as a job, the kernel copies the program's parameters into the job's virtual address space. The number of parameters and their text can be returned to the program through the run-time library routines ELN$PROGRAM_ARGUMENT and ELN$PROGRAM_ARGUMENT_COUNT.

Figure 2–2 illustrates the relationships among the elements in the program list and its related structures — program descriptors, parameter blocks, and KSDs (described in Section 2.4.1.3).

Although system programs usually do not appear explicitly on the Program Description Menu, they too require and receive program descriptors. The following system programs or classes of programs require these internally created program descriptors:

- Local and remote debuggers
- Network device drivers, such as XQDRIVER.EXE
- File access listener (FAL)
- Authorization Service
- Terminal drivers (such as DMBDRIVER.EXE)

Figure 2–2: Program List and Program Descriptors



- Console driver
- LAT driver
- DECwindows Server
- DECwindows terminal and console emulators

As it does with user programs, the System Builder creates parameter blocks for system programs. For example, system device drivers are passed appropriate device names as their program arguments. During its initialization, the driver program retrieves the program argument and uses that text — the device name — in a call to the KER$CREATE_DEVICE kernel procedure. Thus, the use of program

arguments allows the driver to access the device description for the device it supports.

Once the System Builder has assembled a complete program list, in which all specified user programs and required system programs have been included, the System Builder sorts the list into job priority order based on job start-up characteristics. In other words, the program descriptors are sorted into three classes within the program list:

1. Programs that require initialization at system start-up. These are the programs for which the **Init Required** characteristic has been selected in their program descriptions (that is, the PRG$V_SEQ_INITIAL bit is set in the PRG$B_OPTION_FLAGS field). The System Builder places all such programs at the head of the program list in priority order — the highest priority job (with 0 as the highest priority) comes first. Initialization jobs with equal priorities retain their positions relative to one another in the list. No subsequent initialization program will be created until the previous program exits or calls the KER$INITIALIZATION_DONE kernel procedure.
2. Programs that must be created at system start-up. These are the programs for which the **Run** characteristic has been selected in their program descriptions (that is, the PRG$V_AUTO_START bit is set in the PRG$B_OPTION_FLAGS field). The System Builder places all such programs immediately after the initialization jobs in the program list in priority order. Auto-start jobs with equal priorities retain their positions relative to one another in the list.
3. All other programs. Programs that require neither initialization nor automatic start-up appear at the end of the program list in the order in which they were processed.

As a special case, the System Builder checks to make sure that the program descriptor for the debugger component on the target computer appears before the descriptors of any programs that require debugging (PRG$V_START_DEBUG bit set), regardless of job priority.

This highly structured program list simplifies the task of the start-up job that runs during system initialization. The start-up job (described more fully in Chapter 3) is the first job created by the kernel as a VAXELN system boots. Its mission is to walk the program descriptors in the program list and create a job for each program it encounters for which the initialization or auto-start bit is set. Thus, the highest-priority initialization program will be the first job created by the start-up job. For example, in networked applications, this first job is usually the network device driver, which runs at a job priority of 1.

The complete and ordered program list is used throughout the System Builder's subsequent operations. At run time, the program list is used in a number of kernel operations and by several VAXELN utilities.

2.4.1.2 VMS Image Structures Used in Image Processing

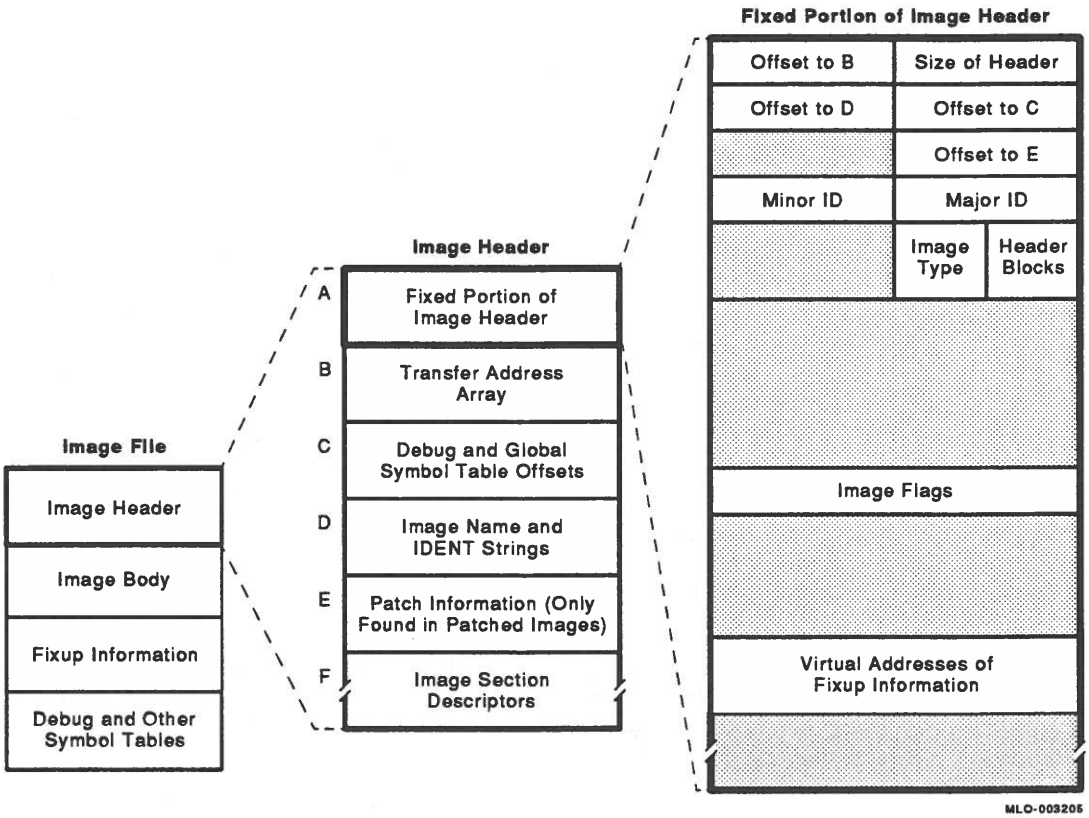
The program images processed by the System Builder are standard VMS images, that is, images generated by the VMS Linker. An image includes the following components, illustrated in Figure 2–3:

- The image header identifies the image, describes its characteristics, and specifies the locations within the image file of other image elements, such as the fixup data for address relocation. Appended to a fixed portion of the header are the variable-length ISDs that describe the characteristics of the image sections in the image body.
- The image body contains the actual code and data for the program in the form of distinct page-aligned image sections. An image section is an assemblage of program sections with like attributes. Each image section in the image file is described by an ISD in the image header.
- The fixup vector is a block containing data required to perform address relocation. Fixup data is stored in an image section and is described by an ISD.

The image header contains the bulk of the information the System Builder needs to process images. For example, the header provides information that allows the image to be copied from the image file into the VAXELN system image file.

Of particular importance within the image header are the ISDs, which appear at the end of the header. Figure 2–4 shows the general layout of an ISD. As shown in the figure, the length of an ISD depends on the type of image section it describes.

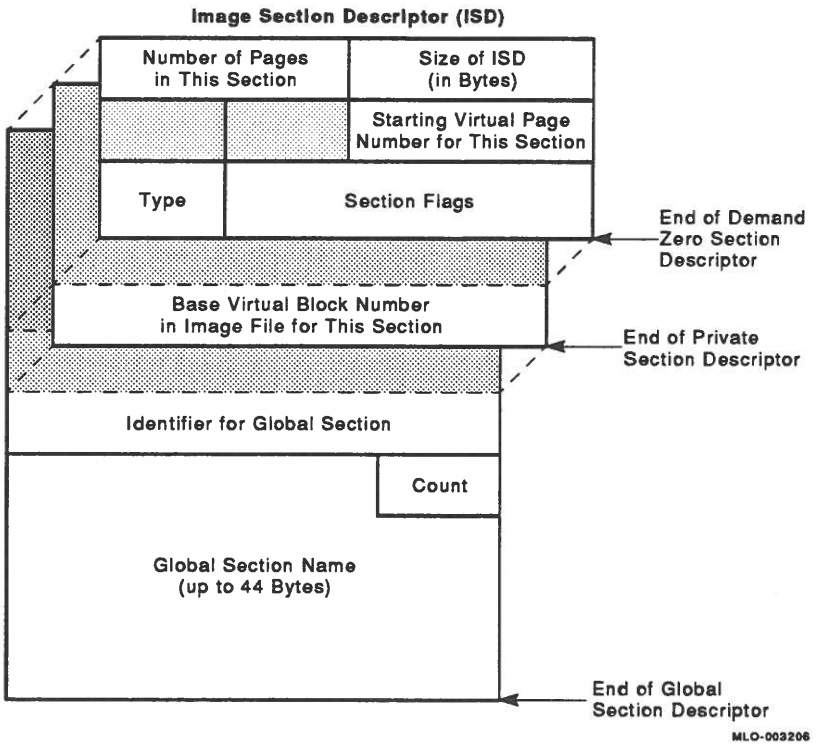
Figure 2–3: Structure of a VMS Image



Each ISD describes a portion of the image's virtual address space, including its size and base address, as determined by the VMS Linker. The contents of an ISD's fields vary to reflect the characteristics of the image section it describes. There are three types of ISDs relevant to processing program images:

- A private section ISD describes code and data that is present in the body of the image file. A private image section can be either read-only or read/write depending on the attributes of the program sections that make up the image section. Image sections containing fixup information are classed as private image sections.

Figure 2–4: General Structure of a VMS ISD



- A demand-zero ISD describes a contiguous range of virtual address space that will be initialized with zeros when the image is mapped to virtual memory. This demand-zero compression saves disk space for the image file, since no actual zero-filled image section appears in the file; only the descriptor for the demand zero section appears.
- A global ISD describes a range of virtual address space to which a shareable image will be mapped into a referencing program's address space. It also identifies the name of the shareable image referenced by the program. More specific information about references to the shareable image appears in the program's fixup vector image section.

Two fields within an ISD identify the nature of the image section. One is the type field. For program images, the System Builder looks for only three of the possible values in the type field:

- ISD$K_NORMAL, which indicates that the image section contains normal data and code
- ISD$K_SHRPIC, which indicates that the ISD describes a shareable image referenced by the program
- ISD$K_USRSTACK, which indicates that the ISD specifies the size and P1 address of the program's VMS-based user stack as generated by the VMS Linker

Another field in the ISD, the flags field, contains bits that, when set, indicate the characteristics of the image section being described; multiple bits can be set. The following flags are relevant to processing executable images:

- ISD$V_GBL, which indicates that the ISD describes a shareable image to be mapped to the program's address space
- ISD$V_CRF, which indicates that the image section must be copied into the program's address space
- ISD$V_DZRO, which indicates that the ISD describes a demand zero image section
- ISD$V_FIXUPVEC, which indicates that the ISD describes a fixup vector image section
- ISD$V_WRT, which indicates that the image section is writeable

Using the information in the ISDs, the System Builder generates kernel section descriptors (KSDs) to describe each image section to the kernel. At run time, the information in the KSDs allows the kernel to map the image into a job's virtual address space. The creation of KSDs for executable images is described in Section 2.4.2.

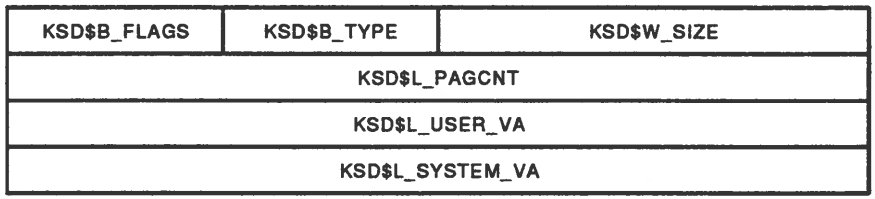
The image header itself is not copied to the system image file. Once the necessary information has been extracted from the header, it is no longer of any use to System Builder — the extracted information now appears in the program descriptor for the image and in the KSDs associated with that descriptor.

2.4.1.3 Kernel Section Descriptors for Program Images

Kernel section descriptors — KSDs — contain information, extracted from an image's ISDs, that describes the characteristics and virtual memory requirements of a VAXELN image section. A program's KSDs are created and linked to the program's descriptor (PRG) by the System Builder. The creation of KSDs for an executable image is described in Section 2.4.2.

At job creation, the kernel scans the program's KSDs and uses their contents to map the program sections into the job's virtual address space. KSDs that describe image sections containing nonshareable read/write or read-only data and code are called private KSDs. Figure 2–5 shows the structure of a private KSD.

Figure 2–5: Structure of a Private KSD



(Two other types of KSDs exist: global and shareable. A global KSD describes a range of virtual addresses within a program's address space to which a shareable image with writeable image sections will be mapped. A shareable KSD describes an image section in a shareable image. Global and shareable KSDs are discussed further in Section 2.6.1.2, Kernel Section Descriptors for Shareable Images.)

Table 2–5 describes the fields in a private KSD.

Table 2–5: Private KSD Fields

Field	Meaning
KSD$W_SIZE	The size of this kernel section descriptor, used to walk list of KSDs
KSD$B_TYPE	A value indicating image section type, as follows:

Value	Meaning
KSD$K_CODE	The associated image section contains read-only data/instructions.
KSD$K_FIXUP	The associated image section contains address relocation fixup data.
KSD$K_DATA	The associated image section contains read/write data.
KSD$K_DZRO	The associated image section should be created as a demand-zero section.
KSD$K_GBL	This KSD represents a shareable image that contains writeable data that must be mapped into the program's address space.
KSD$K_SHARE_DATA	This KSD represents a shareable read/write image section; that is, a global common.

KSD$B_FLAGS — A bit field indicating the characteristics of the image section, as follows:

Bit	Meaning
KSD$V_CRF	The associated image section should be copied into the referencing program's address space.
KSD$V_RWADDRDATA	The associated shareable image section contains read/write address data (.ADDRESS references).

Table 2–5 (Cont.): Private KSD Fields

Field	Meaning
KSD$L_PAGCNT	The number of pages in the image section
KSD$L_USER_VA	The starting virtual address for the image section in the program's address space
KSD$L_SYSTEM_VA	The starting system virtual address of the image section; that is, its location within the mapped system image

2.4.2 Processing Program Images

Once the program list is assembled, the System Builder processes the programs in the list and copies their data and code to the system image file. In this stage of its operation, the System Builder analyzes a program image in a manner similar to that of the VMS image activator. However, the goal of the VMS image activator is to create virtual memory for a program, map its image sections into that address space, and transfer control to the program, all at run time. The System Builder, by contrast, is responsible only for creating the mapping information at build time; this information is then used by the kernel at run time to activate the program as a job.

Both the VMS image activator and the System Builder perform their work on images created by the VMS Linker. See Section 2.4.1.2 for information on the structure of a VMS image.

In processing an executable image, the System Builder's function is fourfold:

- Copy the program's transfer address from the image header to the PRG$L_TRANSFER field in the program descriptor.
- Analyze each ISD in the image header, translate it to an appropriate KSD, and include that KSD as part of the program descriptor.
- Process the shareable images referenced by the program. The System Builder must see to it that every shareable image referenced by the program is included in the system image.
- Copy the program's image sections (including the fixup image section) to the VAXELN system image. Only image sections are copied; that is, the image header is discarded.

The ISDs in a program's image header form the basis for the System Builder's image processing. To translate ISDs into KSDs, the System Builder opens the image file and scans each ISD to determine its type and characteristics. For executable images, an ISD is translated into a private KSD, which describes an image section containing code or data. KSDs for program images are described in Section 2.4.1.3.

The ISDs being processed must be of type ISD$K_USRSTACK, ISD$K_SHRPIC, or ISD$K_NORMAL. If not, processing of the image stops. When an ISD has been processed, the image section it describes is copied from the program image file to the VAXELN system image file.

This same process is applied to every ISD in every program in the program list. When the program list has been exhausted, all system and user programs will be in the system image. Following this block of image sections in the system image is the block of program descriptors and KSDs. In the course of processing the program images, the System Builder will also have processed all the shareable images referenced by the programs in the program list.

The following sections describe how each type of ISD is processed by the System Builder.

2.4.2.1 Processing ISDs of Type ISD$K_USRSTACK

ISDs of type ISD$K_USRSTACK are discarded; they describe the user stack for an image running under VMS. Instead of using this VMS stack, the VAXELN Kernel creates a user-mode stack in P1 space when a process is created. No KSD is created, and no image section is copied to the system image.

2.4.2.2 Processing ISDs of Type ISD$K_SHRPIC

ISDs of type ISD$K_SHRPIC represent a shareable image referenced by the program. The shareable image's identification code is copied from the ISD to the identification field for the referenced shareable image in the program's shareable image list. No KSD is created, and no image section is copied to the system image.

2.4.2.3 Processing ISDs of Type ISD$K_NORMAL

ISDs of type ISD$K_NORMAL require the creation of a private KSD for the described image section. The System Builder allocates a KSD structure and initializes it with zeros. The system virtual address of the program's first KSD is written to the PRG$L_KSD field in the program descriptor. At run time, the KER$CREATE_JOB procedure uses this address to access the KSDs for the program. As each private KSD is created, it is added to the list of the program's KSDs. This block of KSDs is contiguous to the program descriptor and is later copied to the system image file as part of the program list.

The values for the KSD's KSD$L_PAGCNT and KSD$L_USER_VA fields are copied from parallel fields in the ISD. The number of pages specified by KSD$L_PAGCNT will be mapped from the image section described by the KSD to the program's virtual address space beginning at the P0 address found in KSD$L_USER_VA.

The values of the KSDB_TYPE, KSDB_FLAGS, KSD$L_SYSTEM_VA fields depend on the bit settings in the ISD's flags field. The following sections describe how the remaining fields of a KSD are completed, depending on the flag settings in the ISD.

2.4.2.3.1 ISDs with No Applicable Flags Set — Code Sections

If none of the applicable ISD flag bits is set, then a read-only image section has been encountered. Therefore, the KSD type is set to KSD$K_CODE, and KSD$L_SYSTEM_VA is set to the system virtual address at which the image section will appear in the system image. If this image section is adjacent to another private read-only image section, the System Builder combines them and adds the value of KSD$L_PAGCNT in the merged KSD to the same field in the previous KSD.

When the KER$CREATE_JOB procedure encounters a code KSD, it simply maps the image section into the program's address space, starting at KSD$L_USER_VA, by copying the system page table entries for the image section into the job's P0 page table (a process called double mapping). No new physical memory is allocated, and multiple jobs can execute a single copy of the program code.

2.4.2.3.2 ISDs with the ISD$V_DZRO Flag Set — Demand-Zero Sections

If bit ISD$V_DZRO in the ISD flags field is set, then a demand-zero image section has been encountered. Therefore, the KSD flag KSD$V_CRF is set, the KSD type is set to KSD$K_DZRO, and KSD$L_SYSTEM_VA is set to 0. When the KER$CREATE_JOB procedure encounters a demand-zero KSD, it allocates the required KSD$L_PAGCNT number of pages from physical memory, maps them into the program's address space starting at KSD$L_USER_VA, and zeros the pages.

2.4.2.3.3 ISDs with the ISD$V_WRT and ISD$V_CRF Flags Set — Data Sections

If bits ISD$V_WRT and ISD$V_CRF in the ISD flags field are set, then a read/write image section (that is, data) has been encountered. Therefore, the KSD flag KSD$V_CRF is set, the KSD type is set to KSD$K_DATA, and KSD$L_SYSTEM_VA is set to the system virtual address at which the image section will appear in the system image. If this image section is adjacent to another private data image section, the System Builder combines them and adds the value of KSD$L_PAGCNT in the merged KSD to the same field in the previous KSD.

When the KER$CREATE_JOB procedure encounters a data KSD, it allocates the required number of pages from physical memory, maps them in the program's address space starting at KSD$L_USER_VA, and copies the data pages from system virtual address space into the program's address space. This way, multiple jobs executing the same program image will each have a private copy of the program's read/write data.

2.4.2.3.4 ISDs with the ISD$V_FIXUPVEC Flag Set — Fixup Vector Sections

If bit ISD$V_FIXUPVEC in the flags field is set, then an image section containing fixup data has been encountered. Therefore, the KSD type is set to KSD$K_FIXUP, and KSD$L_SYSTEM_VA is set to the system virtual address at which the fixup data will appear in the system image. This address will be used during later processing to resolve the program's references to shareable images, as described in Section 2.6.2.2.

When a fixup vector ISD is encountered, the System Builder interrupts processing of the program image and processes the shareable images referenced by the program, as described in Section 2.6.2.1. If the program references a shareable image that contains a writeable image

section, then a global KSD describing the required address range of the shareable image is created and inserted into the referencing program's list of KSDs. When the KER$CREATE_JOB procedure encounters a global KSD, it locates the shareable image's KSDs and uses them to map the shareable image's sections into the program's address space.

2.5 Device List

A VAXELN system can support a variety of devices, from the console terminal connected to the host processor to, for example, a customized real-time device connected to an IEEE instrument bus controller. Each device or device controller to be used by a VAXELN system must be represented to the kernel by a device descriptor called a system configuration record (SCR). The System Builder creates SCRs from information entered on the Device Description Menu, inserts them into a list, and writes the list to the system image file. At run time — in response to a call to the KER$CREATE_DEVICE procedure — the kernel uses the SCRs to create device objects and associate them with interrupt service routines in device driver jobs.

The System Builder creates SCRs for user-supplied device descriptions by allocating a block of memory and transposing the fields on a System Builder device description into the fields of an SCR, shown in Table 2–6. SCRs for the network controller and console device are added implicitly by the System Builder if required.

Table 2–6: System Configuration Record Fields

Field	Meaning
SCR$L_NEXT	The offset to the next SCR
SCR$W_SIZE	The size in bytes of the device name
SCR$T_NAME	The device name text string
SCR$L_DEVICE	The device register address
SCR$W_VECTOR	The interrupt vector address
SCR$B_IPL	The device hardware interrupt priority level
SCR$B_BI_NUMBER	The BI bus number
SCR$B_ADAPTER_NUMBER	The adapter/BI node number

The System Builder places the length of the device name entered on a device description into the SCR$W_SIZE field and the text of the name in SCR$T_NAME. The hardware interrupt priority level (IPL) of the device is calculated by adding 16 to the bus priority value entered in the interrupt priority field of the Device Description Menu; the result is written to the SCR$B_IPL field. For example, the System Builder default interrupt priority value is 5, which yields a VAX hardware IPL of 21 (5 + 16).

As each SCR is created, it is inserted into the device list. At run time, the kernel locates the completed device list by using the value of the parameter KER$GA_DEVICE_LIST. This value, calculated by the System Builder, is the byte offset from KER$GA_DEVICE_LIST itself to the first SCR in the list. As SCRs are created, the SCR$L_NEXT field is filled with the size of an SCR structure (defined by the global constant SCR$K_LENGTH), rather than with the absolute address of the next SCR in the device list. The value of SCR$L_NEXT, then, is the byte offset to the next SCR in the list.

These offsets, rather than absolute virtual addresses, are used for the values of KER$GA_DEVICE_LIST and SCR$L_NEXT, because the kernel must traverse the list both with memory management disabled (using physical addresses) and subsequently with memory management enabled (using virtual addresses). Since the SCRs in the list are contiguous in the system image, the physical or virtual address of the next SCR in the list is calculated by adding SCR$L_NEXT to the base address of the current SCR.

Once the list of SCRs has been created, the System Builder scans the list and attempts to associate a device driver program with each device, a process called *autoloading*. (Autoloading can be disabled on the Device Description Menu, in which case the user must associate the driver program and device description manually.) To autoload the device, the System Builder takes the first two characters of the device name (for example, "MU"), appends the string "DRIVER" to them (in this case, creating the name "MUDRIVER"), and adds a program descriptor for the driver program to the program list.

Before adding the driver's program descriptor to the program list, the System Builder writes the device name into a job parameter block (JPB) and inserts the block into the driver program's list of parameter blocks. For example, the MUDRIVER program is passed the device name MUA as its program argument. Often, when the driver initializes at system start-up, it obtains this argument and uses it in a call to the CREATE_DEVICE kernel procedure to establish a device object for the device.

Autoloading for terminal controllers works differently because, traditionally, the name of a terminal does not match the name of the driver — a TTA device is not usually controlled by TTDRIVER.EXE. In the case of terminals, the System Builder attempts to associate a device descriptor for a terminal controller (for example, TTA), a collection of terminal descriptions (from the Terminal Description Menu — for example, TTA0, TTA1), and a terminal driver program (for example, DMBDRIVER.EXE).

Specifically, the System Builder takes the first three characters of the name of a terminal line (for example, TTA1) and searches the device list for a match with a device name (for example, TTA). If a match is found, the name of the driver for the terminal controller is derived from the terminal type field in the terminal description (for example, DMB, yielding a driver name of DMBDRIVER.EXE). If a program by that name is not already in the program list, a program descriptor is added, and the terminal controller name is passed to the program in a JPB.

Once all terminal controller device descriptions have been associated with appropriate driver programs, the System Builder scans both the program list and the terminal descriptions searching for matches between a terminal driver's job parameter (for example, TTA) and the first three characters of a terminal's name. If a match is found, then the terminal characteristics, in the form of binary packets, are added as JPBs for the terminal driver program. For example, if DMBDRIVER.EXE has a program argument of TTA, its subsequent program arguments will contain the binary characteristics data for terminal lines TTA0, TTA1, and so on. During its run-time initialization, the driver job will obtain these arguments to establish the line characteristics for each line associated with its controller. In this way, the terminal characteristics entered on the Terminal Characteristics Menu find their way to the device driver for the associated controller.

2.6 Shareable Images

Under the VMS operating system, a program's references to shareable images are resolved at image activation time, specifically, at the point when the image activator installs the program image into process address space. If the shareable image is installed as shareable (/SHARED) on the system, the program executes the shared copy of the image in system virtual address space. If the shareable image is not installed as shareable, then the image must be mapped into the user's address space. In either case, the image activator must perform address relocation on the shareable images.

Address relocation must wait until activation time, because the VMS Linker does not know where in virtual memory a shareable image will be installed. This feature allows a shareable image to be modified without requiring programs that reference it to relink against the new version. Still, deferring this process beyond link time adds to the time required to activate an image.

Under VAXELN, all shareable images are, in effect, installed in system address space by the System Builder. When the kernel creates a job, no extra steps are required to activate the shareable images that the program references. All virtual address mapping information for the program has been distilled into its KSDs, and address relocation for the shareable images it references has taken place at system build time.

Address relocation can occur at build time instead of at run time, because the System Builder knows where the shareable image will be based in virtual memory. Therefore, at run time, the kernel simply reads the job's KSDs and maps the corresponding image sections to the job's virtual address space. Thus the use of shareable images has virtually no impact on the time required to create a job (activate a program image) under VAXELN.

The System Builder incorporates shareable images into the system image under these circumstances:

- A program image refers to a shareable image. When such a reference occurs, the shareable image will be identified in the fixup vector section of the referencing image. When fixup vectors appear in an image, the System Builder scans the data in the fixup vector and includes each shareable image cited. If the referenced shareable image in turn references a shareable image, the second shareable image is included as well. For example, the VAXELN shareable image run-time libraries, such as PASCALMSC.EXE and CMSC.EXE, are included in the system when they are referenced by programs in the program list.
- The name of a shareable image appears in the **Guaranteed image list** entry on the System Characteristics Menu. The names of shareable images that may be referenced by dynamically loaded programs must appear on this list if these shareable images may not be included by appearing in an executable image's fixup vector.
- VAX instruction emulation is requested on the System Characteristics Menu. Instruction emulation is provided through shareable images, which the System Builder incorporates into the system image.

The System Builder also includes a shareable image containing basic console I/O routines that the kernel uses to log messages, such as the system initialization message, to the system console. The use of the console I/O shareable image enables the kernel to write messages to the console without the console driver being present. The console shareable image is processor- and display-hardware specific; the System Builder includes the appropriate version based on the processor and display type selected by the user.

The following sections describe the data structures and operations the System Builder uses in processing shareable images.

2.6.1 Data Structures for Shareable Image Processing

The System Builder analyzes and creates a number of data structures in the course of incorporating a shareable image into a VAXELN system image. Those structures fall into the following classes:

- The shareable image descriptors and the shareable image table. Analogous to program descriptors for executable images, shareable image descriptors are kernel data structures that record information about shareable images included in the VAXELN system. All image descriptors are linked into a list called the shareable image table. The System Builder uses this table to copy the shareable images to the system image file. At run time, the table is used only during dynamic program loading.
- Structures within the VMS shareable image and the image that references it. As with an executable image, the shareable image's header and ISDs must be processed and appropriate KSDs must be created for the shareable image. In addition, the System Builder must access and possibly modify the fixup vector section within the image that references a shareable image.
- Shareable and global KSDs. Analogous to the private KSDs associated with a program descriptor, shareable KSDs describe the makeup and virtual memory requirements of a shareable image's image sections. The kernel may use these KSDs at run time to map a shareable image into a referencing job's address space.

  Global KSDs describe the virtual address requirements of a shareable image that must be mapped into a referencing job's virtual address space. Although a global KSD is created during shareable image processing, it takes its place in the referencing program's list of KSDs. At job creation, the kernel uses the global KSD to locate

the shareable image's shareable KSDs and map them into the referencing job's address space. Global KSDs are required only when the referenced shareable image contains writeable image sections.

The following sections describe these structures in more detail.

2.6.1.1 Shareable Image Descriptors and the Shareable Image Table

The information stored in a shareable image descriptor is used at run time by the ELN$LOAD_PROGRAM utility to resolve a dynamically loaded program's references to shareable images. When the System Builder inserts the shareable image descriptors into the shareable image table, it copies the list to the location in the system image following the device list. The system virtual address of the first descriptor in the table is stored at the global location KER$GA_SHAREABLE_IMAGE_LIST in the kernel data block.

A shareable image descriptor contains the fields shown in Table 2–7.

Table 2–7: Shareable Image Descriptor Fields

Field	Meaning
SHT$L_NEXT	The address of next table entry
SHT$L_IDENT	Shareable image identification data
SHT$L_KSD	The address of the first KSD in the list of KSDs for this image
SHT$L_FIXUP	The address of the fixup vector section for this image
SHT$B_MATCHCTL	The image identification matching information

Table 2–7 (Cont.): Shareable Image Descriptor Fields

Field	Meaning
SHT$B_FLAGS	A bit field indicating characteristics of this image, as follows:

Bit	Meaning
SHT$V_LOCAL_COPY	The image contains a writeable image section and must be mapped to a referencing program's address space.
SHT$V_RWADDRDATA	The image contains read/write address data (.ADDRESS references to itself or other shareable images); its KSDs must be copied to the referencing program's KSD list, and extra copies must be made of its image sections that contain .ADDRESS references.

SHT$T_NAME	The string descriptor containing the size of the image name and the text of the name

The System Builder uses the shareable image table to locate and manipulate the shareable images it has read into internal memory. By the time the table is written to the system image, it includes descriptors for all the shareable images referenced by programs, those specified on the guaranteed image list, those requested by the selection of VAX instruction emulation, and the console I/O image.

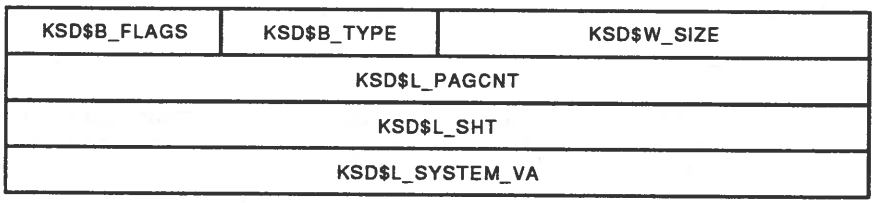
At run time, however, the kernel has no need to use the shareable image table, because the execution of shareable images is completely transparent under normal circumstances. The shareable image table is used only for dynamically loaded programs. In these cases, the ELN$LOAD_PROGRAM utility needs the table to perform address relocation for the loaded program's references to shareable images. Since the dynamically loaded program is not known to the System Builder, build time address relocation cannot resolve the program's references to shareable images; instead, the relocation must occur at run time, before KER$CREATE_JOB activates the program.

2.6.1.2 Kernel Section Descriptors for Shareable Images

As described in Section 2.4.1.3, KSDs describe the characteristics and virtual memory requirements of a VAXELN image section. When processing a shareable image, the System Builder translates the image's ISDs into shareable KSDs. These KSDs are then linked to the shareable image's descriptor, becoming part of the shareable image table entry for that shareable image.

Two types of KSDs are associated with shareable images: shareable and global. A shareable KSD is the shareable-image equivalent of a private KSD for executable images; that is, the System Builder translates the ISDs in the shareable image to KSDs as it does for the ISDs of an executable image, except that the resulting structure is known as a shareable KSD, shown in Figure 2–6.

Figure 2–6: Structure of a Shareable KSD



The shareable KSD records the address of the shareable image descriptor to which it belongs in the KSD$L_SHT field. Table 2–8 describes the fields that appear in both shareable and global KSDs. See the KSD$B_TPYE and KSD$B_FLAGS entries in Table 2–5 for the bits and constant values, respectively, that can be recorded in the KSD$B_TYPE and KSD$B_FLAGS fields of both shareable and global KSDs.

Table 2–8: Shareable and Global KSD Fields

Field	KSD Type	Meaning
KSD$W_SIZE	Both	The size of this kernel section descriptor, used to walk list of KSDs
KSD$B_TYPE	Both	The storage for a constant value indicating the image section type
KSD$B_FLAGS	Both	A bit field indicating the characteristics of the image section
KSD$L_PAGCNT	Both	The number of pages in the image section
KSD$L_USER_VA	Global	The starting virtual address for the image section in the program's address space
KSD$L_SYSTEM_VA	Shareable	The starting system virtual address of the image section; that is, its location within the mapped system image
KSD$L_SHT	Shareable	The address of the shareable image table entry describing the shareable image to which this KSD belongs
KSD$L_GBL_KSD	Global	The system virtual address of the first KSD in the list of KSDs for the shareable image the global KSD represents

When a shareable image contains a writeable image section, its shareable KSDs must be mapped into the address space of the referencing job. The System Builder creates a global KSD describing the location of the shareable KSDs and the number of virtual pages they require; this global KSD is then inserted into the referencing job's list of KSDs. Figure 2–7 shows the structure of a global KSD.

When the KER$CREATE_JOB procedure encounters the global KSD while mapping the referencing program into virtual memory, it uses the data in the global KSD to locate and map the shareable image's KSDs into the program's address space. In particular, the kernel uses the address in KSD$L_GBL_KSD to locate the first KSD in the shareable

Figure 2–7: Structure of a Global KSD

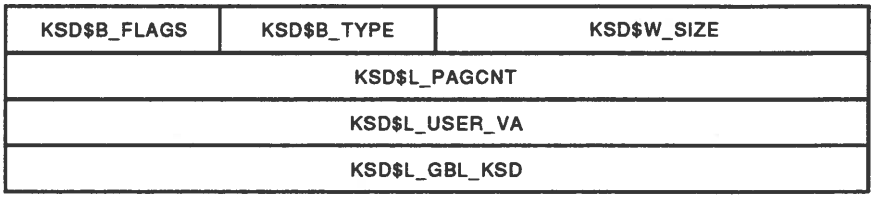
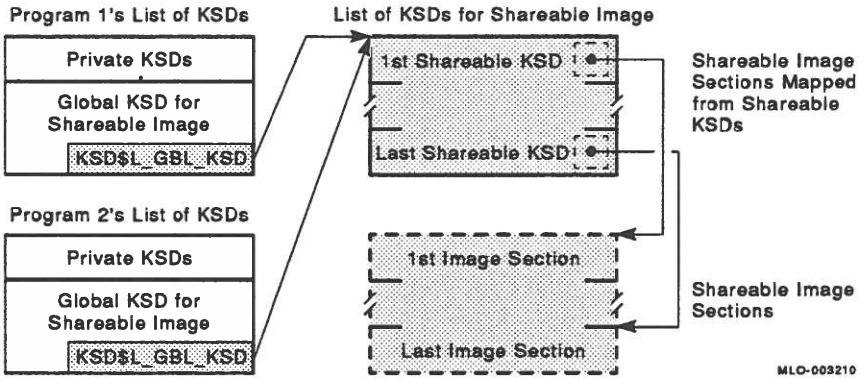


Figure 2–8: A Global KSD Refers to Shareable KSDs



image's list of KSDs and then maps them into the referencing program's address space starting at the address in KSD$L_USER_VA. Figure 2–8 shows how the global KSD refers to the KSDs in the shareable image table.

2.6.1.3 VMS Image Structures Used in Shareable Image Processing

A shareable image has the same general structure and components that an executable image has: a header, a body, and, possibly, a fixup vector section.

While processing a shareable image, the System Builder scans the ISDs in the image header. For shareable images, the System Builder looks for only three of the possible values in the ISD type field:

- ISD$K_SHRPIC, which indicates that the image section is shareable and position-independent
- ISD$K_PRVPIC, which indicates that the image section is nonshareable and position-independent
- ISD$K_PRVFXD, which indicates that the image section is nonshareable and position-dependent (usually a fixup vector image section)

The flags field in the ISD is also examined to determine whether an image section is global, copy-on-reference, fixup, or writeable. The setting of the bits in the flags field determines whether a shareable image is purely shareable or whether it must be mapped into the address space of the image that references it.

Of central importance to the processing of shareable images is the fixup vector image section supplied by the image that references the shareable image. The System Builder finds which shareable images a program image references by analyzing the program's fixup section. A fixup vector image section is indicated by the ISD$V_FIXUPVEC bit set in an ISD's flags field. When the System Builder encounters such an ISD during image processing, it begins to process the shareable images cited in the fixup section.

The fixup vector image section, shown in Figure 2–9, contains three blocks of information that the System Builder uses to incorporate shareable images into the system image:

- A shareable image list (SHL). This list contains an entry describing each shareable image referenced. An SHL entry contains a number of fields that are read and updated during processing of the shareable image. After image processing, these fields contain the following information about a shareable image:
  - The SHL$L_BASEVA field specifies the base virtual address at which the shareable image is mapped into the referencing

program's address space. This address is used for address relocation.

— The SHL$L_SHLPTR field points to the shareable image descriptor for the shareable image that the SHL entry describes. Shareable image descriptors are discussed in Section 2.6.1.1.

— The SHL$L_IDENT field contains data used to detect any mismatch between the shareable image the program linked against and the actual shareable image appearing in the system image.

— The SHL$T_IMGNAM field contains a string descriptor specifying the name of the shareable image.

This list should not be confused with the shareable image table created by the System Builder.

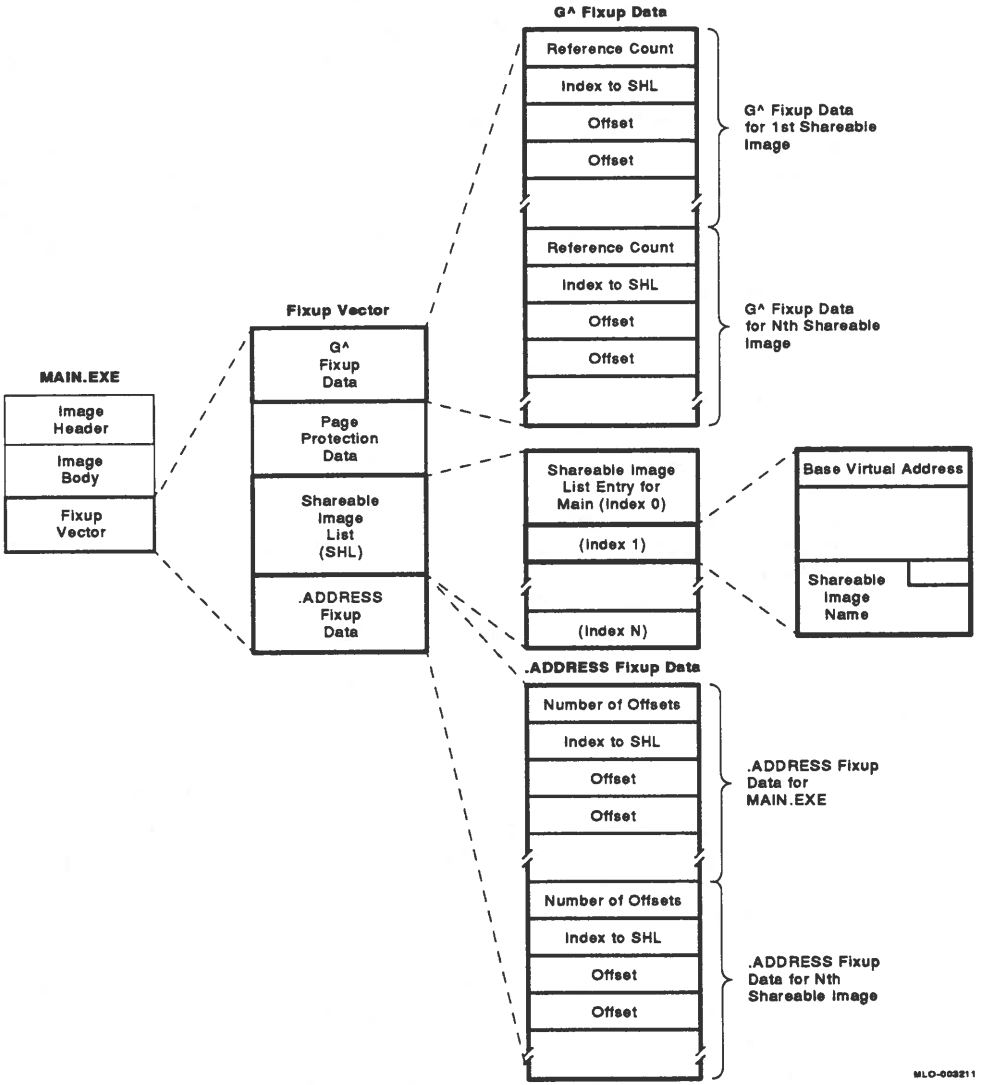
- A table of address relocation information for general-mode (G^) references to shareable images. This information is used by the System Builder to perform address relocation for the program's references to global locations within the shareable images in its SHL.
- A table of address relocation information for .ADDRESS references to shareable images. This information is used by the System Builder to perform address relocation for the program's use of addresses of global locations within the shareable images in its SHL.

Given the information in a program's fixup section, the System Builder can completely integrate the VAXELN or user-written shareable images that a program references.

2.6.2 Processing Shareable Images

The processing of shareable images occurs in two distinct phases. The first phase involves creating shareable image descriptors and KSDs, and copying the image sections to the system image file. The second phase involves performing address relocation to resolve references within the system's program images to locations within the system's shareable images. The following sections describe these two phases of shareable image processing.

Figure 2–9: Structure of an Image Fixup Vector



2.6.2.1 Creating Shareable Image Descriptors and KSDs

As it does for executable images, the System Builder processes shareable images by creating a descriptor for each image, opening the image file, analyzing its image header and ISDs, and creating KSDs to describe its image sections. The descriptors and KSDs (the shareable image table) and image sections for the shareable image are then written to the system image file. The address of the first shareable image descriptor is recorded in the location KER$GA_SHARE_LIST in the kernel's parameter block; this value is copied to the location KER$GA_SHAREABLE_IMAGE_LIST in the kernel data block during system initialization. The shareable images themselves follow the table and comprise the last element in the system image.

The inclusion of implicitly requested shareable images — that is, those shareable images to which another image refers — begins when the System Builder encounters a fixup vector ISD during its translation of an image's ISDs to KSDs (the fixup vector image section may reside in either an executable image or a shareable image). The System Builder then scans the fixup section and includes the shareable images that are listed in the fixup vector's SHL, described in Section 2.6.1.3. (Explicitly requested shareable images — those on the guaranteed image list — are included during a separate phase of processing, but their treatment is otherwise identical. Only those images on the list that are not already in the shareable image table are included.)

The System Builder takes the following steps to process a shareable image:

1. Records the address of the current fixup section in an internal list. This list will be used later to perform address relocation.
2. Extracts the name of the shareable image from the SHL$T_IMGNAM field and determines whether this shareable image has already been processed. If so, processing of the shareable image ceases.
3. If the shareable image has not already been processed, creates a shareable image descriptor for it and adds it to the shareable image table. The contents of the descriptor are described in Section 2.6.1.1.
4. Opens the shareable image file, scans the ISDs in the image header, translates them to KSDs, adds the KSDs to the image descriptor, and copies the shareable image into memory. If the shareable image references any other shareable images, the System Builder

repeats this process for those images, using the current image's SHL to locate the additional shareable images.

5. Repeats the entire process for each shareable image in the image's shareable image list.
6. Copies the completed shareable image table, including all shareable KSDs, and the shareable image sections themselves from the System Builder's dynamic memory to the system image file.

If all shareable images contained only read-only code and data, their processing would be as straightforward as it is for executable images. In that simplest of cases, all programs referencing a given shareable image use the single copy of the image that the System Builder loads into the system image file. At run time, a reference to a location within a "pure" shareable image will point to a system virtual address, a location common to all jobs in the system. That location can be referenced by multiple readers since none will be modifying the shareable image.

The processing of shareable images, however, is potentially quite complex. Complications arise when a shareable image contains a writeable image section. Since any program referencing the writeable image section in the shareable image can alter its contents, data integrity must be assured by providing that each referencing program owns a private copy of the writeable image section. To accomplish this, the System Builder creates a global KSD describing the image sections in the writeable shareable image and inserts the KSD into the referencing program's own list of KSDs.

The ISDs being processed must be of type ISDK_SHRPIC, ISDK_PRVPIC, or ISD$K_PRVFXD. If not, processing stops for the shareable image. An ISD of type ISD$K_SHRFXD generates a warning message, since it cannot be used for address relocation.

For ISDs of the correct type, the System Builder allocates a KSD structure and initializes it with zeros. The value for the KSD's KSD$L_PAGCNT is copied from a comparable field in the ISD. The KSD$L_SHT field is set to the address of the KSD's associated descriptor, and the system virtual address of the image section is written to the KSD$L_SYSTEM_VA field.

The values of the KSD$B_TYPE and KSD$B_FLAGS fields depend on the bit settings in the ISD's flags field. Moreover, much additional processing is required when an ISD has certain flag settings. The following sections describe how the flag settings in a shareable image's ISD influence the creation of KSDs for the shareable image. Table 2–9

summarizes the characteristics of the KSDs that can be created for shareable images.

Table 2–9: Characteristics of Shareable KSDs

ISD Flag	Section	KSD Flag	KSD Type	Run-Time Characteristics
None	Read-only	None	KSD$K_ CODE	Shareable in system space; referenced through S0 page table
ISD$V_WRT ISD$V_CRF	Nonshareable read/write data	KSD$V_CRF	KSD$K_DATA	Physical memory allocated; image section copied and mapped; referenced through P0 page table
ISD$V_WRT	Shareable read/write data	None	KSD$K_ SHARE_DATA	Mapped into program space; referenced through P0 page table
ISD$V_ FIXUPVEC	Fixup vector	None	KSD$K_ FIXUP	Pure shareable image: shareable and referenced through S0 page table; otherwise, private copy of the fixup section is mapped into program space to vector G^ references to shareable image mapped into program space

2.6.2.1.1 No Applicable Flags Set — Shareable Code Sections

If none of the applicable ISD flags is set, then a read-only shareable image section has been encountered. Therefore, the KSD type is set to KSD$K_CODE, and KSD$L_SYSTEM_VA is set to the system virtual address at which the image section will appear in the system image. If this image section follows another private read-only section, the System Builder combines the two sections by adding the current KSD's KSD$L_PAGCNT value to that in the previous KSD. The two image sections are now effectively merged.

A read-only shareable image section is potentially shareable by all programs in a VAXELN system. If the section resides in a pure shareable image (SHT$V_LOCAL_COPY is clear), then no double mapping of the section into the program's address space is necessary, and all programs will reference a single copy of the image section. The process of address relocation ensures that a program's reference to the shareable image is vectored to the correct location in system space through the system (S0) page table.

If the shareable image contains any writeable image sections (excluding fixup sections), then the image's read-only shareable sections must be double mapped into the referencing program's address space, where they will be referenced through the program's P0 page table. However, no physical memory is allocated for the shareable section. Because the shareable image contains a writeable section, its shareable sections are referenced through the job's P0 page table instead of through the S0 page table.

2.6.2.1.2 ISD$V_WRT and ISD$V_CRF Flags Set — Data Sections

If bits ISD$V_WRT and ISD$V_CRF are set, then a nonshareable read/write image section (that is, data) has been encountered. Therefore, the KSD flag KSD$V_CRF is set, the KSD type is set to KSD$K_DATA, and KSD$L_SYSTEM_VA is set to the system virtual address at which the image section will appear in the system image. If this image section follows another data image section, the System Builder combines the two sections by adding the current KSD's KSD$L_PAGCNT value to that in the previous KSD. The two image sections are now effectively merged.

When a nonshareable, writeable image section is encountered, the SHT$V_LOCAL_COPY bit is set in the shareable image descriptor. At run time, physical memory is allocated for the writeable image section, the content of the section is copied to the allocated memory, and the memory is mapped into the referencing program's address space.

2.6.2.1.3 ISD$V_WRT Flag Set and ISD$V_CRF Clear — Shareable Data Sections

If the ISD$V_WRT bit is set without the ISD$V_CRF being set, then a shareable read/write image section has been encountered. Therefore, the type is set to KSD$K_SHARE_DATA, and KSD$L_SYSTEM_VA is set to the system virtual address at which the image section will appear in the system image. If this image section follows another shareable, read/write section, the System Builder combines the two

sections by adding the current KSD's KSD$L_PAGCNT value to that in the previous KSD. The two image sections are now effectively merged.

An image section with the KSD$K_SHARE_DATA attribute represents a global common section that can be read and updated by multiple jobs. At run time, the section is simply mapped from its system address into the referencing program's address space. (However, if the pages containing the shareable section are located in ROM on a MicroVAX I, the kernel allocates physical memory and copies the section there at run time; the physical memory is still shared by all referencers.)

Because the section is writeable, the shareable image that contains it is marked with the SHT$V_LOCAL_COPY flag and is represented in the referencing program's list of KSDs by a global KSD. At run time, KER$CREATE_JOB will begin double mapping the shareable image's KSDs into the program's address space. Because the KSD$V_CRF bit is not set in the shareable read/write section's KSD, the kernel will not create a private copy of the section; only the system page table entries that map the section will be copied into the job's P0 page table. Therefore, each program that references that section will reference and modify the single copy of the section in physical memory. (Even though the writeable section is shareable, it must be mapped into the program's address space, because user-mode programs are unable to write to memory mapped in the system page table.)

2.6.2.1.4 ISD$V_FIXUPVEC Flag Set — Fixup Vector Sections

If bit ISD$V_FIXUPVEC is set, then an image section containing fixup data has been encountered. Therefore, the type is set to KSD$K_ FIXUP.

The presence of a fixup vector ISD means that the current shareable image references yet another shareable image; as a result, the System Builder processes the shareable images in this fixup vector's SHL. If any of the shareable images referenced by this shareable image contain a writeable image section, then the current shareable image's SHT$V_ LOCAL_COPY bit is set, even if it does not itself contain writeable sections.

The processing required by the presence of a fixup vector in a shareable image depends on whether the image contains writeable sections.

2.6.2.1.4.1 Shareable Images Without Writeable Sections

For shareable images that contain no writeable sections or reference no other writeable shareable images, no further processing of the fixup vector — beyond address relocation — is required. Through address relocation, all references to the shareable image point to the single copy of the image in system address space. Address relocation is discussed in more detail in Section 2.6.2.2.

2.6.2.1.4.2 Shareable Images with Writeable Sections

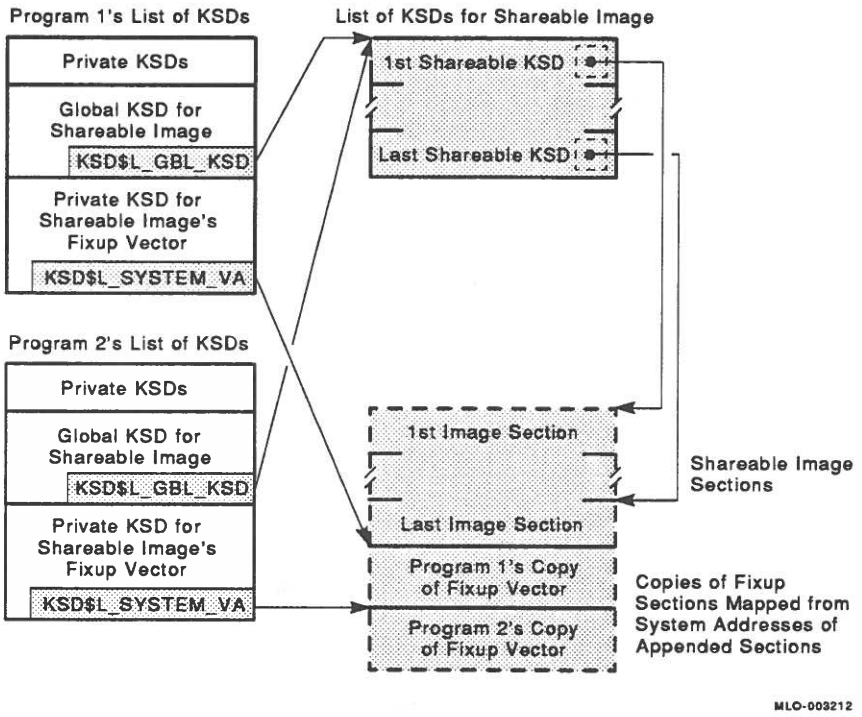
The processing that results from the presence of a fixup section in a writeable shareable image (SHT$V_LOCAL_COPY is set) is more involved than it is for pure shareable images. The following steps are taken:

1. The KSD that describes the shareable image's fixup vector section is *copied* and inserted into the referencing program's list of KSDs as a private KSD.
2. The KSD$L_SYSTEM_VA field in the fixup KSD is set to the system address at which a *private copy* of the fixup section will appear in the shareable image in the system image file.
3. The base address of the fixup section within the program's address space is recorded in the KSD$L_USER_VA field.

A private copy of the fixup section for each referencing program is required for correct address relocation. Address relocation for the referencing program must be calculated using the base virtual address at which the shareable image will be mapped in the referencing program's address space. This base address can be different for every program that references the writeable shareable image. As it processes shareable images, the System Builder calculates how many programs in the program list reference a given shareable image. That number of copies of the shareable image's fixup section is appended to the body of the shareable image when it is copied to the system image file.

When KER$CREATE_JOB encounters a program's shareable image fixup KSD, it maps the fixup image section into the program's address space from the system virtual address of the program's unique copy of the fixup appended to the shareable image. Address relocation, too, is performed on this private copy of the shareable image's fixup vector. Figure 2–10 shows how programs are associated with their private copies of a writeable shareable image's fixup vector.

Figure 2–10: Multiple Fixup Vectors in Writeable Shareable Images



If the writeable shareable image contains no read/write address data (.ADDRESS references), then only two additional KSDs are inserted into the referencing program's list of KSDs: the global KSD representing the shareable image and the private KSD describing the fixup vector section.

If the SHL entry in the fixup vector indicates that the writeable shareable image contains .ADDRESS references, including such references to itself, then *all* the shareable image's KSDs are copied into the referencing program's KSD list as private KSDs; no global KSD is used. The System Builder also sets the SHT$V_RWADDRDATA bit in the shareable image descriptor and sets the KSD$V_RWADDRDATA bit in each duplicate KSD that actually contains a .ADDRESS reference. (Sections that contain .ADDRESS references are writeable since they must be updated during address relocation; therefore, a shareable image that

contains read/write address data always has SHT$V_LOCAL_COPY set as well.)

This duplication of shareable KSDs as private KSDs is really an extension of the System Builder's method for handling fixup sections in writeable shareable images. To fix up the .ADDRESS references, the offset stored in the .ADDRESS cell by the VMS Linker must be replaced by the sum of the offset and the base address at which the shareable image will appear in the referencing program's address space. Since that base address can differ for every referencing program, a private copy of the image sections containing .ADDRESS references must be made for every program that references those sections.

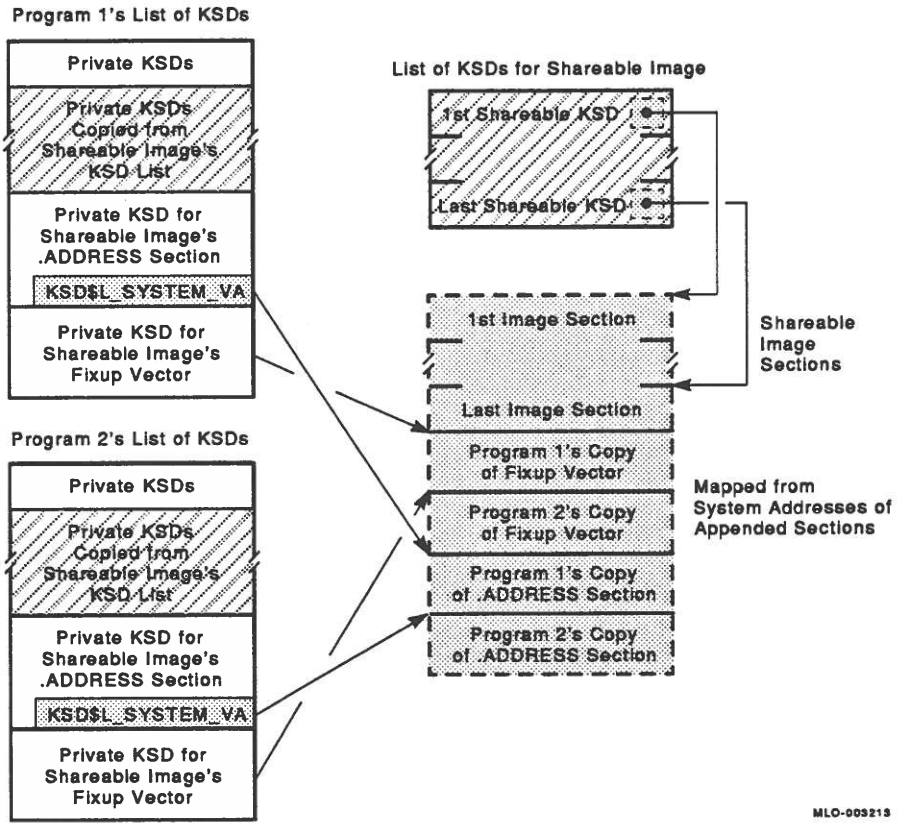
Like fixup vectors, the private copies of the .ADDRESS sections are appended to the end of the shareable image as it is copied to the system image file. The unique system virtual address for each copy is recorded in the KSD$L_SYSTEM_VA field of its associated duplicate KSD. When the System Builder performs address relocation for a referencing program, the .ADDRESS fixups are made to the program's private copies of the sections in the shareable image that contain the actual .ADDRESS cells, using the base virtual address of the shareable image within the program's address space.

Figure 2–11 shows how programs are associated with their private copies of a writeable shareable image's .ADDRESS sections.

At run time, KER$CREATE_JOB treats these duplicate KSDs as it does other private KSDs — they are mapped into the program's address space from the system address in the KSD$L_SYSTEM_VA field to the address in the KSD$L_USER_VA field. Processing a KSD whose KSD$V_RWADDRDATA bit is set differs only in that it is mapped from the system address at which the referencing program's private copy of the .ADDRESS section is appended to the shareable image. Other programs mapping the same shareable image from their own duplicate KSDs will reference unique copies of the sections containing .ADDRESS references.

For dynamically loaded programs, these private copies of fixup and .ADDRESS sections are created in the job's dynamic memory, since they cannot be appended to the body of the shareable image at run time.

Figure 2–11: Multiple .ADDRESS Sections in Writeable Shareable Images



2.6.2.2 Address Relocation Fixup

When the shareable image table is complete and the shareable KSDs and images have been copied to the system image file, the System Builder performs address relocation. Under VMS, the image activator performs this task at image-activation time; similarly, under VAXELN, the program loader utility performs address relocation when a program is loaded from local or remote disk. Address relocation can be performed at system-build time under VAXELN because the System Builder has determined the user and system base virtual addresses of

every shareable image in the system. Once these base virtual addresses are known, address relocation can proceed.

Address relocation, or fixup, is required for images (executable and shareable) that make general-mode (G^) and .ADDRESS references to locations — data or routine entry points — within shareable images. The information the System Builder requires to perform address relocation is stored in an image's fixup vector section — the SHL and the G^ and .ADDRESS vector tables — and in a series of internal structures built during the creation of an image's KSDs. An image's SHL contains a list entry and fixup tables for each shareable image referenced.

The G^ and .ADDRESS vector tables record a series of offsets. For G^ references, the table records the offset from the base of the referenced shareable image to the actual location referenced. The G^ location itself contains the address of the associated G^ vector within the G^ vector table. When processed by the VMS Linker, G^ references are translated to @L^ references (longword relative deferred). When the G^ operand is fetched during program execution, the deferred address of the referenced location in the shareable image is obtained by vectoring the reference through the G^ vector in the vector table.

For .ADDRESS references, the vector table records the offset from the base of the referencing image to the location of the .ADDRESS directive within the referencing image. The .ADDRESS cell itself contains the offset from the base of the referenced shareable image to the actual location referenced.

In general, address relocation fixup involves adding the base address of the referenced shareable image to the G^ or .ADDRESS cell offset. The base address of the referenced shareable image used for the fixup is determined by whether the shareable image contains writeable image sections. If it does — that is, the SHT$V_LOCAL_COPY bit is set in the shareable image descriptor — then the image is mapped into the referencing program's virtual address space. If the shareable image is pure, then its base address is the system virtual address at which the image appears in the system image. (Since shareable images that contain .ADDRESS references are always writeable, they are fixed up using base addresses within the referencing program's address space.)

To perform address relocation, the System Builder scans an internal list of fixup control blocks and processes each entry in the list. Each control block describes the location of a fixup section within the system image file. Other internal information allows the System Builder to locate the image sections that contain .ADDRESS references as well. The process of address relocation proceeds as follows:

1. The System Builder reads a fixup section into an internal buffer and locates the SHL entry for the first shared image referenced. The fixup section can be native to the referencing program (that is, one generated by the VMS Linker), or it can be one of the local copies of a writeable shareable image's fixup section appended to the end of a shareable image in the system image file.
2. The image identification stored in the fixup control block is compared to the identification in the SHT$L_IDENT field in the descriptor for the referenced shareable image (the address of the descriptor is stored in the SHL$L_SHLPTR field in the SHL entry for the image). If the two identifications do not match according to the match control specified for the shareable image (match-all, match-if-equal, or match-if-less-than-or-equal), then a warning message citing the shareable image and the referencing program is generated, and processing stops for this fixup vector. The identification information is checked for each shareable image in the SHL.
3. If the fixup vector contains G^ references, the base address of the referenced shareable image is added to each offset in the G^ fixup table entry, yielding the actual address of the referenced locations within the shareable image.

   If the shareable image is pure, then the system base address for the image, stored in the System Builder's internal copy of the shareable image descriptor, is used for the fixup calculation. If the referenced shareable image is writeable (SHT$V_LOCAL_COPY is set), the value stored in SHL$L_BASEVA is used in the fixup calculation; this address represents the base address of the shareable image within the referencing program's address space.
4. If the fixup section contains .ADDRESS fixups, the System Builder reads into memory the image section or sections in the referencing program containing the .ADDRESS directives. The fixup will be made on the duplicate (private) copy of the .ADDRESS section appended to the end of the shareable image in the system image file. (These sections are described by private KSDs — with the KSD$V_RWADDRDATA bit set — in the referencing program's list of KSDs.)

   A .ADDRESS cell within a section is located by adding the base address of the image containing the .ADDRESS directive to the offset in the .ADDRESS fixup table entry. Finally, the fixup is made by adding the base address of the shareable image to the offset stored in the .ADDRESS cell, yielding the address of the referenced location within the referenced shareable image.

Because shareable images with .ADDRESS references are writeable, the system base address for the image, the value stored in SHL$L_BASEVA, is used in the fixup calculation. This address represents the base address of the shareable image within the referencing program's address space.

The entire process is repeated for every .ADDRESS reference within the referencing program.

5. The process is repeated for the next fixup control block. When the System Builder has exhausted the list of fixup control blocks, address relocation is complete.

2.6.3 A Shareable Image Example

Even a simple program linked against a shareable image library can require elaborate shareable image support from the System Builder. This section presents just such a simple program, written in C, and uses an excerpt from a full System Builder map to examine how the System Builder supports the program's use of shareable images.

Consider the following C program, which writes a single line to the standard output device:

```
#include $vaxelnc

test()
    {
       printf ("Hello, world!\n");
    }
```

Compiling the program, linking it against the CRTLSHARE shareable library, and building it into a VAXELN system image produces the following program entry in a full System Builder map:

```
TEST                    DISK:[USER]TEST.EXE;1

                        No debug, Run, No initialize, Mode = User
                        User stack = 1, Kernel stack = 1
                        Job priority = 16, Process priority = 8
                        Job message limit = 16384
                        Power recovery exception = Disabled
                        Argument(s):
```

```
Image section(s):
  Type           Base VA     Page(s)    Image
  Noshr Write    00000200          1
  Read-only      00000400          1
  Fixup vector   00000600          1
  Read-only      00000800         38    DCIO (1)
  Noshr Write    00005400          1    DCIO (2)
  Fixup vector   00005600          2
  Shareable      00005A00         20    CMSC
  Fixup vector   00007E00          1
Transfer address:  00000400
```

Notice in particular the composition of the image sections. Compare them with the map's ISDs for the shareable images DCIO.EXE (C I/O routines) and CMSC.EXE (general-purpose C routines) referenced by TEST.EXE:

```
DCIO                    SYS$SYSDEVICE:[ELN]DCIO.EXE;2
                        Major Id: 2, Minor Id: 0
                        Map into program region = Yes
                        Image section(s):
                          Type           Base VA     Page(s)
                          Read-only      80008A00         38
                          Noshr Write    8000D600          1
                          Fixup vector   8000D800          2

CMSC                    SYS$SYSDEVICE:[ELN]CMSC.EXE;2
                        Major Id: 2, Minor Id: 0
                        Map into program region = Yes
                        Image section(s):
                          Type           Base VA     Page(s)
                          Read-only      8000E200         20
                          Noshr Write    80010800          1
                          Fixup vector   80010A00          1
```

As both entries show, each shareable image will be mapped into the referencing program's address space at run time — 41 pages from DCIO and 22 pages from CMSC (including their fixup vector pages).

Both images appear in the entry for TEST.EXE because the System Builder has inserted a KSD or KSDs describing the shareable images into the list of KSDs for TEST.EXE. Table 2–10 explains the source of each image section created for TEST.EXE.

Table 2–10: KSDs and Image Sections for TEST.EXE

Section	Pages	Source	Explanation
Noshr Write	1	TEST.EXE	A writeable, private section generated by the VAX C compiler to hold the string constant "Hello, world!\n".
Read-only	1	TEST.EXE	The read-only code for the program.
Fixup vector	1	TEST.EXE	The fixup section generated to record the program's shareable-image references and a G^ fixup vector table entry for its single reference to the **printf** function in the shareable image DCIO.EXE.
Read-only	38	DCIO.EXE	The read-only code in the DCIO shareable image. Because DCIO.EXE contains .ADDRESS references (demonstrable by running ANALYZE/IMAGE on DCIO.EXE), the System Builder copies all its KSDs into TEST.EXE's KSD list.
Noshr Write	1	DCIO.EXE	The page in DCIO.EXE that contains its six .ADDRESS references. The KSD for this section was copied from DCIO.EXE's KSD list into TEST.EXE's KSD list. TEST.EXE's copy of the KSD actually maps a private copy of the page in DCIO.EXE that the System Builder appended to the end of the shareable image. The System Builder performed .ADDRESS fixups on this duplicate fixup section to resolve DCIO.EXE's .ADDRESS references to itself within TEST.EXE's address space. See Section 2.6.2.1.4.2 for a description of how private copies of .ADDRESS sections are made.
Fixup vector	2	DCIO.EXE	A private copy of DCIO.EXE's two-page fixup section, which contains the fixup information for the shareable image's G^ and .ADDRESS references to other shareable images. Because DCIO.EXE is writeable, the System Builder inserts a KSD describing the fixup section into TEST.EXE's KSD list. TEST.EXE's copy of the KSD actually maps a private copy of fixup pages in DCIO.EXE that the System Builder appended to the end of the shareable image. The System Builder performed G^ fixups on this duplicate fixup section to resolve DCIO.EXE's general-mode references within TEST.EXE's address space. See Section 2.6.2.1.4.2 for a description of how private copies of fixup sections are made.

Table 2–10 (Cont.): KSDs and Image Sections for TEST.EXE

Section	Pages	Source	Explanation
Shareable	20	CMSC.EXE	A global KSD inserted in TEST.EXE's KSD list to map CMSC.EXE's image sections. Although 20 of CMSC.EXE's 21 pages are read-only code, its one writeable page means that the shareable image must be mapped into TEST.EXE's address space. Although CMSC.EXE is not referenced directly by TEXT.EXE, it is referenced by DCIO.EXE and is therefore included in the system image.
Fixup vector	1	CMSC.EXE	A private copy of CMSC.EXE's fixup section, which contains the fixup information for the shareable image's G^ references to other shareable images. Because CMSC.EXE is writeable, the System Builder inserts a KSD describing the fixup section into TEST.EXE's KSD list. TEST.EXE's copy of the KSD actually maps a private copy of fixup pages in CMSC.EXE that the System Builder appends to the end of the shareable image. The System Builder performed G^ fixups on this duplicate fixup section to resolve CMSC.EXE's general-mode references within TEST.EXE's address space.

The growth in the size of DCIO.EXE in response to the addition of the duplicate .ADDRESS and fixup sections can be demonstrated by building successive versions of the C program into the system image under different names — call them TEST1.EXE and TEST2.EXE. The number of pages DCIO.EXE occupies in the system image can then be determined by subtracting the base system address of DCIO.EXE from the base system address of the shareable image that follows it in the system image; in this case, CMSC.EXE.

With only TEST.EXE built into the system, the base address of the DCIO.EXE is $80008A00_{16}$. The base address of CMSC.EXE is $8000E200_{16}$, yielding a difference of 5800_{16} or 44 pages. This figure reflects DCIO.EXE's native size of 41 pages plus the three pages added for TEST.EXE's private copies of the one page containing the .ADDRESS cells and the two pages containing the fixup section.

With both TEST.EXE and TEST1.EXE built into the image, the base address of DCIO.EXE is 80009000_{16} and that of CMSC.EXE is now $8000EE00_{16}$, yielding a size of 47 pages for DCIO.EXE. Adding TEST2.EXE to the system image gives DCIO.EXE a base address of 80009800_{16} and CMSC.EXE a base address of $8000FC00_{16}$, giving

DCIO.EXE a size of 50 pages. (The base address of DCIO.EXE increases because the addition of programs forces it to a higher address in the system image.)

As becomes evident from the pattern established here, each time another program references DCIO.EXE, the size of the shareable image grows by three pages as it accommodates the referencing program's private copies of the image's single .ADDRESS page and two fixup pages. The System Builder's address relocation procedure ensures that each program's references are resolved through its private copies of the .ADDRESS and fixup sections.

Chapter 3

System Bootstrap, Kernel Initialization, and Application Start-Up

The goal of the VAXELN Kernel's initialization sequence is to prepare the system for the execution of VAXELN jobs and processes on the target processor. Before the kernel can initialize itself, it must be loaded into processor memory by the VAX primary bootstrap program, VMB.

This chapter first describes, in Section 3.1, the role of VMB in loading the VAXELN system image into memory and transferring control to the code for the kernel's initialization. Section 3.2 then focuses on the initialization process itself, which proceeds through the three distinct phases of the secondary bootstrap:

1. Unmapped initialization. Memory management is disabled while the kernel creates essential system data structures and maps them into system address space.
2. Enabling memory management. This allows the kernel to execute using system virtual addresses.
3. Mapped initialization. The kernel creates and initializes the remainder of its data structures, creates the start-up job, and completes its initialization by invoking the scheduler to begin job execution.

Section 3.3 then describes how the VAXELN start-up job creates system and user application jobs.

3.1 Primary Bootstrap: VMB

The VAX primary bootstrap program, called VMB, provides a general-purpose method for bootstrapping VAX processors. VMB's primary functions are the following:

- To locate and determine the size of physical memory on the system
- To locate the secondary bootstrap program, load it into memory from the boot device, and transfer control to it

  For VAXELN systems, the secondary bootstrap program loaded into memory is the VAXELN system image created by the System Builder and described in Chapter 2, The VAXELN System Image.

The operation of VMB is described in some detail in the text *VAX/VMS Internals and Data Structures*. This discussion will focus on the layout of main memory that VMB establishes before it transfers control to the secondary bootstrap program. This memory state is the starting point for the kernel's initialization.

Figure 3–1 shows the layout of memory as VMB transfers control to the VAXELN Kernel and shows how the locations of those memory structures are transmitted to the kernel. Table 3–1 describes these structures.

Figure 3–1: State of Physical Memory After VMB Executes

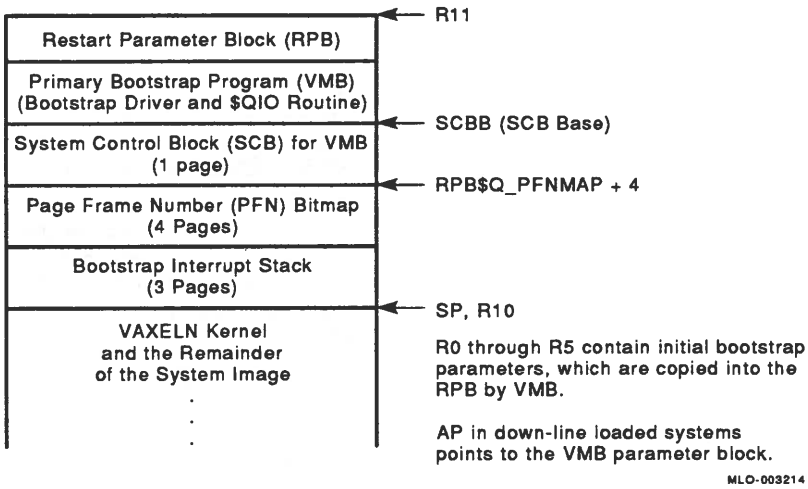


Table 3–1: Bootstrap Elements in Memory After VMB Executes

Element	Description
Restart parameter block (RPB)	A 512-byte structure that stores bootstrap parameters (loaded by VMB), system values (loaded by the kernel), and volatile machine state information (loaded by KER$POWER_FAIL in module POWERFAIL) to enable system restart after a power failure (although restarts are not supported). For example, VMB stores the size and address of the page frame number (PFN) bitmap in the quadword RPB$Q_PFNMAP; the kernel then uses and updates this field when it initializes its own PFN bitmap.
Primary bootstrap program	The VMB image. VMB occupies fewer than 64K bytes of memory. Most of the VMB image is overwritten as the secondary bootstrap program is read into memory. All that remains of VMB at this point is the appropriate bootstrap driver and a skeletal $QIO routine. These remainders of the VMB image are eventually overwritten as the kernel begins to allocate the page frames they occupy.
System control block (SCB)	A one-page structure that contains vectors for interrupt, exception, and machine-check handlers for VMB's execution. The kernel creates its own boot-time SCB at a different location; therefore, VMB's SCB is eventually overwritten.

Table 3–1 (Cont.): Bootstrap Elements in Memory After VMB Executes

Element	Description
Page fram number (PFN) bitmap	A bitmap that contains one set bit for each good page of memory located by the bootstrap's memory-testing sequence. The first longword in the quadword RPB$Q_PFNMAP contains the size in bytes of the bitmap; the second longword contains the physical address of the bitmap. On systems with 8 megabytes or fewer of memory, the PFN bitmap is held in these original four pages; otherwise, the bitmap is located in contiguous page frames at the high end of main memory. When the kernel initializes, it copies and maps the bitmap, and the pages occupied by the original bitmap are eventually overwritten. Initially, the bitmap shows all good page frames as available, including those occupied by the VMB structures. Any frames not marked in the bitmap are eventually overwritten by subsequent allocations of physical memory. Therefore, the kernel marks the page frames occupied by those structures that it will keep — the RPB and the system image — as used.
Bootstrap interrupt stack	The stack used by VMB during its execution. When control is transferred to the kernel, the stack pointer (SP) points to the base of this stack. The kernel uses this stack, which grows toward lower addresses, until memory management is enabled, at which time the kernel's interrupt stack is used. After this, the pages occupied by the original stack are eventually overwritten.
VAXELN Kernel	The secondary bootstrap program. VMB transfers control to the first byte in the VAXELN system image that it has loaded into memory.

3.2 Secondary Bootstrap: Initializing the Kernel

Once the system image has been loaded into physical memory on the target processor, control is transferred to the first byte in the image. That byte is a Branch with Word Displacement (BRW) instruction to transfer control to the start of the kernel initialization code, at label KER$START in module INITIAL. The kernel's execution begins in the following environment:

- Memory management is disabled.
- IPL is set to 31, disabling all interrupts.
- Execution is on the boot interrupt stack.
- Memory is organized as shown in Figure 3–1.

The following sections describe the kernel's initialization, which occurs in three stages:

1. Unmapped initialization (Section 3.2.2)
2. Enabling memory management (Section 3.2.3)
3. Mapped initialization (Section 3.2.4)

Most of the initialization sequence does not depend on the type of processor executing the code. However, certain portions of the sequence, such as the initialization of I/O adapters and the SCB, do require processor- or bus-specific code. The mechanism the kernel uses to support processor-dependent initialization is described in Section 3.2.1.

After initialization completes, system virtual memory is structured as shown in Figure 3–2. Table 3–2 describes the structures mapped into S0 memory during initialization.

Figure 3–2: Mapping of the S0 Region by the Kernel

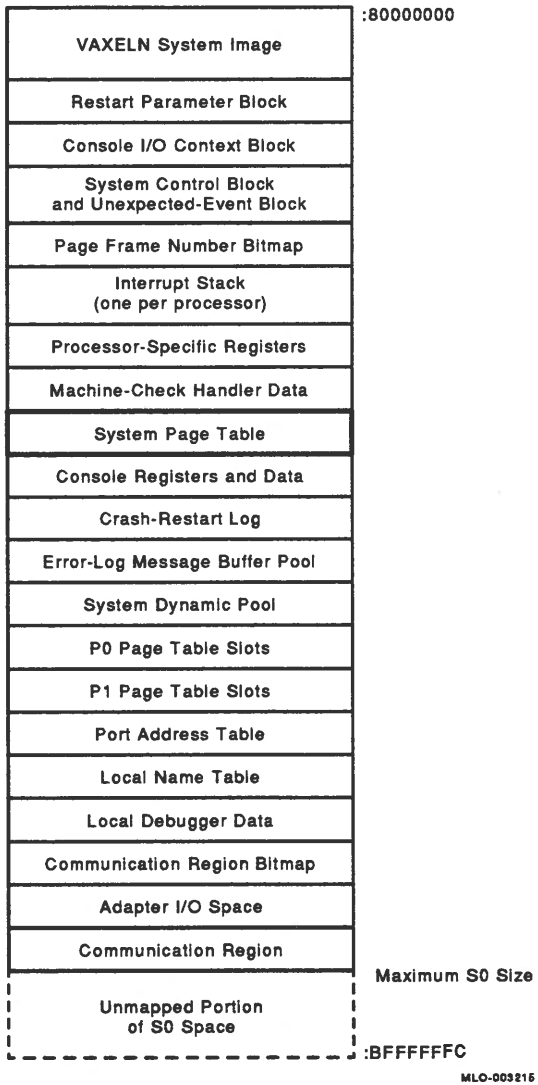


Table 3–2: System Components Mapped into S0 Address Space

Component	Description
System image	The memory-resident image of the system image file produced by the System Builder. The system image is mapped from the base of S0 space — 80000000_{16} — for the number of page frames equal to KER$GW_SYSTEM_SIZE. Chapter 2 describes the components of a system image.
Restart parameter block	The structure created by VMB. See Table 3–1.
Console I/O context block	A block containing pointers to data and flags required by the console I/O subsystem.
System control block (SCB) and unexpected-event dispatch block	A multipage structure containing the addresses (vectors) of routines to handle exceptions and interrupts. The size of the SCB varies from processor to processor. Identical in size to the SCB, the unexpected-event dispatch block contains instructions that transfer control to a handler routine when unexpected interrupts or exceptions occur.
Page frame number (PFN) bitmap	The structure representing available pages of physical memory. See Table 3–1.
Interrupt stack	The stack used when the system is running in system context; for instance, during system initialization and the execution of device interrupt service routines. Code that runs on the interrupt stack can reference only system addresses.
Processor-specific registers	The system virtual address region to which the processor-specific registers (such as the MicroVAX system-extension identification register) are mapped. These registers are accessed with MOV instructions and should not be confused with the architecturally defined internal processor registers accessed with MTPR and MFPR instructions.
Machine-check handler data	The system virtual address region that maps processor-specific data used during machine-check handling.
System page table	The VAX page table that maps the entire system region. This table supports the translation of system virtual addresses to physical addresses.
Console registers and data	The system virtual address region that maps the console I/O registers and data that enable the kernel to write messages directly to the system console.

Table 3–2 (Cont.): System Components Mapped Into S0 Address Space

Component	Description
Crash-restart log	The system virtual address region that maps buffers used during the orderly shutdown of the system after a fatal system bugcheck.
Error-log message buffer pool	The system virtual address region that maps a number of buffers — set by the **Number of buffers** value on the Error Log Characteristics Menu — available for storing posted error log entries before they are written to the error log file. The size of each buffer is determined by the global value KER$GW_EMB_SIZE (1 page).
System dynamic pool	The address range that maps a number of 128-byte blocks — set by the **Dynamic pool** value on the System Characteristics Menu — available for the creation of kernel objects and related system components.
P0 page table slots	A block of system virtual addresses reserved for P0 page tables, their allocation bitmaps, and the P0 page table slot bitmap.
P1 page table slots	A block of system virtual addresses reserved for P1 page tables, their allocation bitmaps, and the P1 page table slot bitmap.
Port address table	A block of system virtual addresses reserved for the addresses of the port objects created by the jobs in the system. The table contains the number of longwords specified as the global parameter value KER$GW_PORT_SIZE.
Local name table	A block of system virtual addresses reserved for the listheads for the local name table, in which the kernel records names associated with local port objects. The table contains the number of table listheads specified by the global parameter value KER$GW_NAME_SIZE (128).
Local debugger data	The system virtual address region that maps a buffer used by the local debugger. This buffer exists only on systems that include local debugging support.
Communication region bitmap	The bitmap used to control the allocation of virtual pages in the communication region.

Table 3–2 (Cont.): System Components Mapped into S0 Address Space

Component	Description
Adapter I/O space	The system virtual address region that maps the VAX physical addresses reserved for I/O adapters, such as VAXBI nodes and UNIBUS adapters, and control/status registers.
Communication region	A block of system virtual addresses reserved for the allocation of device communication regions and dynamic program space, and for job allocation of S0 memory. Accompanying the region is an allocation bitmap. The size of the region (and its bitmap) is determined by the global parameter value KER$GW_IO_SIZE, which combines the values of the **System region size** and **Dynamic program space** entries on the System Characteristics Menu. The word size of the value limits the size of the communication region to 65,535 pages (32 megabytes).

3.2.1 Processor-Specific Factors

When initialization requires a processor-specific operation, the kernel calls generic internal subroutines to perform the operation appropriate for the target processor. The processor-specific initialization operations include:

- Configuring I/O address space
- Mapping I/O address space
- Creating the system control block
- Initializing machine-check data blocks
- Mapping and initializing processor-specific registers

For example, to determine the size of the SCB required by the target processor, the kernel calls the generic subroutine KER$SCB_PAGCNT, which returns the number of SCB pages required for the processor type. The code executed in the subroutine depends on the type of kernel executing; that is, KER$SCB_PAGCNT returns a different value when called in the kernel for Q22-bus-based processors than it does in the kernel for VAXBI-based processors.

The processor-specific version of these subroutines is included when the kernel image for the processors is linked. For example, when the 8NNKER kernel is linked, it includes the processor-specific initialization module INIT8NN. One INIT*nnn* module exists for each processor supported by the kernel. When the kernel calls one of the generic subroutines, such as KER$SCB_PAGCNT, control is transferred to the subroutine in module INIT8NN, which returns the number of SCB pages required by the VAXBI-based processors supported by that module.

In the QBUSKER and UBUSKER versions of the kernel, processor-specific subroutine calls are vectored through an intermediate, bus-specific module, either COMBQ22BUS or COMBUNIBUS, which is also included at link time for QBUSKER and UBUSKER. These modules contain the generic subroutine entry points, such as KER$SCB_PAGCNT, which in turn dispatch the call at run time (through a CASE instruction) to a processor-specific routine based on the exact processor type.

For example, when the kernel calls KER$SCB_PAGCNT while executing on a MicroVAX II-based processor (KA620 or KA630), control is first transferred to the generic subroutine entry point in the COMBQ22BUS module. That routine then determines the processor type and transfers control to the processor-specific subroutine UV2$SCB_PAGCNT in module INITUV2.

The kernel does on occasion determine the processor type in line, when only a few instructions are involved. Usually, the type of processor can be determined from the contents of the global value KER$GB_CPU_TYPE, which is set early in the initialization sequence. Processor type can also be determined from global Boolean values such as KER$GB_RTVAX, which indicates that processor is a KA620.

3.2.2 Unmapped Initialization

When VMB transfers control to the kernel, VAX memory management — mapping — is disabled, and the processor uses the physical addresses of code and data. The major goal of this first, unmapped phase of initialization is to create and initialize the system page table so that memory management can be enabled.

During unmapped initialization, the kernel keeps track of both the current physical and system virtual addresses of the data structures it creates. Once a structure is created, consuming physical and virtual address space, these addresses are updated to reflect the next available page of physical and virtual memory.

Unmapped initialization follows these basic steps:

1. The address of the first available page of physical memory is determined. If the system image resides in ROM, the system data block is copied to writeable memory. See Section 3.2.2.1.
2. The console is initialized for I/O from the kernel, and the console I/O context block is allocated and initialized. See Section 3.2.2.2.
3. The SCB is initialized with the address of kernel boot-time interrupt, exception, and machine-check handlers. See Section 3.2.2.3.
4. The processor type is determined. In tightly coupled symmetric multiprocessing systems, the number of available processors is also determined. See Section 3.2.2.4.
5. Certain values are copied from the system parameter block to the system data block. On down-line loaded systems, the system node name and address are saved in the data block as well. See Section 3.2.2.5.
6. The PFN bitmap is copied to its final location, and its bitmap descriptor is initialized. See Section 3.2.2.6.
7. The physical and virtual memory sizes of the system data structures are computed. See Section 3.2.2.7.
8. The system page table is created, and the system structures are mapped into it. See Section 3.2.2.8.

3.2.2.1 Step 1 — Find the First Writeable Page and Copy ROM Data

The kernel determines the address of the first writeable page of memory by adding the number of pages in the system image (KER$GW_SYSTEM_SIZE) to the number of pages by which the beginning of the system image is offset from the base of memory. This offset is represented by the physical address of the global location KER$VECTOR_START.

If the system is not executing from ROM, this first writeable page is the first page in memory following the system image, and the kernel sets the value of KER$GL_FIRST_WRT_PAGE to 0. Otherwise, the first writeable page is the first page in memory, pointed to by R10 and SP. (The value of KER$GL_FIRST_WRT_PAGE is used during

job creation to determine whether the memory of a shareable, writeable image section must be copied from ROM. A nonzero value for KER$GL_FIRST_WRT_PAGE means that the copy must be made. See Section 2.6.2.1.3, ISD$V_WRT Flag Set and ISD$V_CRF Clear — Shareable Data Sections.)

When a VAXELN system is booted from ROM on a MicroVAX I target, the kernel copies the system data block from its location in read-only memory to the first writeable pages of memory. The base address of the data block, once represented by the value of the global label KER$GR_KERNEL_DATA, is now the physical address of the copied data, and the first writeable page is now the physical address of the page following the new data block. Subsequent allocations of physical memory begin with that page.

Even though the data block is actually copied only on ROM-based MicroVAX I systems, in all cases the kernel subsequently writes to data cells using each cell's relative displacement from the physical base of the data block. This form of reference is required when the data has been copied because the original addresses of the data cells, represented by their symbolic names, have been rendered invalid by the displacement of the data block. Therefore, to write to a data cell, the kernel determines the location of the cell by subtracting the virtual address of the base of the data block (KER$GR_KERNEL_DATA) from the virtual address of the data cell, represented by its KER$ symbolic name, for example, KER$GA_SPT_PHYSICAL.

Throughout the unmapped phase of initialization, the kernel stores the physical address of the data block in a general register (R8). Therefore, the kernel would write the value of KER$GA_SPT_PHYSICAL in the following manner:

```
MOVL    R11,W^KER$GA_SPT_PHYSICAL-KER$GR_KERNEL_DATA(R8)
```

The virtual address of the original location of the data block is subtracted from the virtual address of data cell KER$GA_SPT_PHYSICAL. The result represents the byte offset of the data cell within the block. This value is then used as a byte displacement from the physical base of the block, stored in R8, to yield the physical address of the cell. The data is then written to that location. Subsequent reads of KER$GA_SPT_PHYSICAL must also use this technique, because the updated value must be read from the potentially relocated data block.

Later in unmapped initialization, the kernel determines whether it has copied the data block by comparing the current base address of the block to the value of KER$GR_KERNEL_DATA. If the values are different, the kernel then maps the relocated data block into the S0 page table entries that had mapped the original data block in ROM. When memory management is enabled, the relocated data cells can then be referenced by their symbolic names, because their virtual addresses have been remapped to their new physical locations in the data block.

3.2.2.2 Step 2 — Initialize the Console

The kernel initializes the system console for I/O by calling the internal subroutine KER$CONIO_INITIAL (in module CONSOLIO). This function obtains the byte offset of the console I/O shareable image from the parameter KER$GL_BIOS_OFFSET and adds it to the physical base of the system image, yielding the physical address of the console I/O code. This address is then stored in KER$GA_CONIO_CODE.

The subroutine then calls the console I/O routine to initialize the console registers by vectoring a procedure call through the address in KER$GA_CONIO_CODE. Because this is a physical address, the console will have to be reinitialized during mapped initialization so that the virtual address of the console I/O code can be written to KER$GA_CONIO_CODE. On certain processors, console initialization includes the creation and initialization of a console I/O context block to record the addresses of components such as fonts and graphics controller control/status registers.

The kernel does not normally write to the console during unmapped initialization, but, because the console might receive boot-time error and machine-check messages, it must be initialized during this phase of execution.

3.2.2.3 Step 3 — Initialize the Boot-Time SCB

For the unmapped phase of initialization, the kernel establishes a special SCB to handle unexpected interrupts, exceptions, and machine checks.

To create the SCB, the kernel obtains the size in pages of the SCB for the target processor (by calling the subroutine KER$SCB_PAGCNT) and fills each longword vector in those pages with the address of a boot-time interrupt/exception handler. The second longword in the SCB —

the machine-check vector — is filled with the address of the boot-time machine-check handler.

The first page of the SCB occupies the first page in memory beyond the system image or beyond the console I/O context block (if one is present), and the physical address of that page becomes the value of the SCBB (SCB base) privileged register. (When the system image is in ROM, the SCB begins on the first page after the relocated kernel data block or after the console I/O context block.)

The unexpected interrupt/exception handler and the machine-check handler are the local subroutines BOOT_INTEXC and BOOT_MACHINECHK, respectively. Both routines display an error message on the console and place the processor in an infinite loop to await the manual halt and reboot of the processor.

3.2.2.4 Step 4 — Determine the Processor Type

The kernel must determine early in initialization whether it is compatible with the target processor. This determination is made by calling the processor-specific subroutine KER$CHECK_CPUID (in module INIT*nnn*). This routine reads the contents of the system identification register (SID) and compares the processor identification there with the identifications of the processors supported by the version of the executing kernel. If a mismatch occurs, the routine returns a failure value, causing the kernel to display an error message on the console and enter an infinite loop.

If the general processor type is supported — for example, a Q22-bus-based processor — processor type is narrowed further — for example, a MicroVAX II versus a MicroVAX 3600. Based on the value of the SID register, the kernel sets or clears processor-type flags in the data block. For example, on Q22-bus-based processors, the low bit in KER$B_QBUS is set; on VAXBI-based processors, the KER$B_VAXBI flag is set.

These processor flags are checked by some kernel routines to determine whether certain processor-specific actions should be taken. For example, when creating process page tables, the kernel checks the KER$GB_RTVAX (KA620) flag. If it is set, physical addresses, instead of the usual system virtual base addresses, are used as the values of P0BR and P1BR.

3.2.2.5 Step 5 — Copy Parameters to the Data Block

Values in the kernel parameter block (in module PARAMETER) that are designated "initial" are copied to their respective cells in the kernel data block (see Tables A–2 and A–1). For example, the read-only value KER$GQ_INITIAL_CONNECT_TIMEOUT set by the System Builder is copied to the read/write cell KER$GQ_CONNECT_TIMEOUT in the data block. The transfer of these data items allows their values to be dynamically updated in future versions of the kernel.

If the system image was down-line loaded, the kernel copies the target's and host's node names and addresses from the VMB parameter block to KER$GQ_NODE_ADDRESS and KER$GT_NODE_NAME, and KER$GQ_HOST_ADDRESS and KER$GT_HOST_NAME, respectively.

3.2.2.6 Step 6 — Initialize the PFN Bitmap

The kernel copies the PFN bitmap created during processor bootstrap to the next free pages in memory. The bitmap descriptor located at RPB$Q_PFNMAP in the RPB specifies the size and address of the bitmap. If the **Memory limit** entry on the System Characteristics Menu has been set to a nonzero value (in the system parameter KER$GL_MEMORY_LIMIT), the kernel truncates the bitmap to the specified length, effectively limiting the amount of physical memory that the kernel recognizes.

A zeroed longword is appended to the end of the bitmap as a terminator, and the quadword descriptor in the RPB is updated with the new size and address of the PFN bitmap. The descriptor is used later to map the bitmap and create the kernel's own bitmap descriptor in the data block.

3.2.2.7 Step 7 — Compute the Sizes of System Data Structures

Before the kernel can create the system page table, it must determine the number of pages the system will occupy. Each page will be represented by one longword page table entry (PTE) in the system page table (SPT). The kernel keeps two tallies: the number of physical pages and the number of virtual pages.

The count of physical pages is used to determine the physical address of the base of the SPT and whether the target contains enough physical memory to contain those system structures that are mapped in the system page table before the system page table itself is mapped; the count of virtual pages is used to determine the size of the page table.

Therefore, the kernel counts only the physical sizes of structures that are mapped into system address space before the system page table.

For example, the machine-check handler data block is mapped into system address space before the system page table; therefore, its required number of pages is added to both the counts of physical and virtual pages. By contrast, the error-logging message buffer is mapped after the system page table; therefore, its required number of pages is added only to the count of virtual pages.

The global value KER$GW_SYSTEM_SIZE specifies the number of pages occupied by the system image itself. The total system size is calculated by adding the size of the system image to the size of the following system data structures:

- The restart parameter block. This block occupies a single page.
- The console I/O context block. This block occupies the number of pages required for data to support the console I/O subsystem. The block exists only on systems that support workstation graphics displays.
- The system control block and unexpected-event dispatch block. The size of the SCB is processor-specific. The kernel calls KER$SCB_PAGCNT to return the SCB size for the target processor. The returned value is doubled to accommodate the unexpected-event dispatch block. See Section 3.2.4.3.
- The PFN bitmap. The number of bytes in the PFN bitmap specified in the RPB$Q_PFNMAP descriptor is used to calculate the number of bitmap pages, rounded to the next page.
- The interrupt stack. The size of the interrupt stack is specified by the value of the system parameter KER$GW_ISTACK_SIZE, set on the System Characteristics Menu.

  Since each processor in the system has its own interrupt stack, the number of pages specified (plus one as an invalid-stack page), is multiplied by the number of processors.
- Processor-specific registers. The number of pages required to map process-specific registers, such as the system identification register extension on MicroVAX processors, is determined by calling the process-specific subroutine KER$REGSP_PAGCNT.
- Console registers and data. The number of pages required to map the console's registers and data block is determined by calling console-specific I/O procedures.

- The machine-check handler data block, the crash-restart logs, error-log message buffers, and dump control block and I/O buffer. The size of the machine-check handler data block is determined by calling the processor-specific subroutine KER$MCHK_PAGCNT. Since each processor in the system requires its own machine-check data block, the number of pages is multiplied by the number of processors.

  If error logging is enabled for the system, the value of KER$GW_EMB_COUNT (set on the Error Log Characteristics Menu) is added only to the total of virtual pages. The number of pages required for the dump control block and the dump I/O buffer are read from the code block for the system dump facility.
- The system dynamic pool. The number of pages in the system pool is determined by the value of KER$GW_POOL_SIZE, set on the System Characteristics Menu.
- The P0 and P1 page table slots and associated bitmaps. The size of a process page table slot includes the size of the page table itself plus the size of its page table entry bitmap. The total number of pages required to hold all the P0 and P1 page table slots includes the sizes of the two bitmaps that control the allocation of the page table slots themselves. One bitmap page — the minimum allocation for a bitmap — can represent 4096 (8 * 512) page table slots. Figure 9–3 shows the layout of system memory reserved for the P0 and P1 page table slots.

  Size requirements are determined as follows:

  — The number of bits in the page table slot bitmap is determined by the values of KER$GW_P0_SLOT_COUNT and KER$GW_P1_SLOT_COUNT (set on the System Characteristics Menu), rounded up to the next page.

  — The number of pages required by the table slots is determined by multiplying KER$GW_P0_SLOT_COUNT or KER$GW_P1_SLOT_COUNT by the size in pages of a single P0 or P1 page table and its bitmap descriptor.

  — The size of a page table, in pages, is determined by the value of KER$GW_P0_SLOT_SIZE or KER$GW_P1_SLOT_SIZE (also set on the System Characteristics Menu).

    Since each page table page will map 128 pages of virtual memory, 128 bitmap bits — 16 bytes — is required for each page table page.

Once the size in pages of a bitmap is determined, that value is added to KER$GW_P0_SLOT_SIZE or KER$GW_P1_SLOT_SIZE to yield the value of KER$GW_P0_SLOT_LENGTH and KER$GW_P1_SLOT_LENGTH, respectively.

Since the page table slots are mapped after the system page table, their sizes are added only to the count of virtual pages.

- The port address table. The value KER$GW_PORT_SIZE (set on the System Characteristics Menu) specifies the number of ports the system can support at one time. One page of the port address table is required for each 128 ports requested.
- The local name table. The local name table requires eight bytes for each name listhead. The number of listheads is determined by the value of KER$GW_NAME_SIZE (128). One page is required for every 64 listheads.
- I/O space and the communication region. The total number of pages required to map these regions depends on the number of pages required to map the system's I/O space, the dynamic program region, and the system region. This number of pages is determined by the local subroutine COMPUTE_IO_SIZE.

  COMPUTE_IO_SIZE uses the processor-specific subroutine KER$COMPUTE_IOPAGCNT to obtain the number of pages required for I/O space; this value is added to KER$GW_IO_SIZE (which combines the System Builder values **System region size** and **Dynamic program space**) to yield the number of pages in the communication region. This total is added only to the virtual page count.
- The system page table itself. Each page in the system page table can map 128 pages of system virtual memory. Therefore, the size of the page table is determined by the total number of virtual pages in the system plus the number of page table pages required to map those system virtual pages.

3.2.2.8 Step 8 — Initialize the System Page Table and Map Existing Components

Once the kernel has determined the maximum virtual size of the system, the system page table can be created. The physical page count becomes the page frame number (PFN) of the base address of the system page table, and the virtual page total becomes the length of the table.

The kernel compares the length of the page table with the length of the PFN bitmap. If the bitmap contains fewer bits than the number of virtual pages the page table can map, the system has insufficient physical memory. Therefore, the kernel displays an error message on the console and enters an infinite loop. If sufficient physical memory exists, the physical address of the page table is written to the system base register (SBR) and KER$GA_SPT_PHYSICAL, and the virtual page total is written to the system length register (SLR) and KER$GL_SPT_LENGTH.

With the system page table initialized, the kernel then begins mapping selected system components in the page table. At this point, only structures that already reside in physical memory are mapped. Once they are mapped, memory management can be enabled. The top half of Figure 3–2, down to the system page table, shows the elements mapped at this stage of initialization.

The mapping is performed by calling the local subroutine KER$FILL_SPTE. This routine creates a system page table entry for each page by inserting the current physical page frame number into the SPTE specified by the kernel's system page table entry pointer. After each SPTE is created, the subroutine advances the values of the current page frame number, the current system virtual address, and the address of the next SPTE to be filled.

The routine also marks the page frame in use by setting its corresponding bit in the PFN bitmap (only if the PFN is within the bitmap; for example, physical addresses used to map I/O space are not represented by bits in the PFN bitmap). KER$FILL_SPTE is called once for each page frame mapped in the system page table.

The following system components, which precede the system page table in virtual memory, are mapped in the system page table at this time:

- The system image. The kernel sets the current page frame counter to indicate the start of the system image (KER$VECTOR_START), calls KER$FILL_SPTE for each page specified by KER$GW_SYSTEM_SIZE, and restores the page frame counter. Since the first page of the system image is mapped in the first SPTE, the S0 addresses assigned to components in the system image — based at 80000000_{16} — will be valid when memory management is enabled.
- The kernel data block. If the kernel's data has been relocated from its original location in MicroVAX I ROM, the kernel maps the data into the SPTEs that mapped the original data block.

- The RPB. The current virtual address is written to KER$GA_RPB to set the base of the RPB, and the block is mapped. The kernel also completes the initialization of the RPB at this point.
- The console I/O context block. The current virtual address is written to KER$AA_CONIO_CONTEXT + 4 to set the virtual base of the context block. Each page in the block is mapped.
- The SCB. The current virtual address is written to KER$GA_SCB_BASE. The number of pages in the SCB is returned by calling KER$SCB_PAGCNT and is doubled to account for the size of the unexpected-event dispatch block that follows the SCB. Each page of both blocks is mapped.
- The PFN bitmap. The PFN bitmap descriptor located at KER$GR_PAGE_BITMAP is initialized with the length and virtual address of the bitmap. Each page is mapped.
- The interrupt stack. The interrupt stack for each processor is mapped, and the SPTE representing the end page in each stack is cleared.
- Processor-specific registers. These registers are mapped by calling the processor-specific subroutine KER$REGSP_MAP, which returns the address of a table that describes the registers specific to that processor. This address is then passed to the local subroutine MAP_REGSPACE to perform the actual mapping. The current virtual address becomes the value of KER$GA_CPUREGSP.
- The machine-check handler data block. The number of pages for the machine-check handler data block is determined by calling the processor-specific subroutine KER$MCHK_PAGCNT. That number of pages is mapped for each processor in the system, and the virtual address of each block is written to the array located at KER$GA_MACHINECHK_DATA, indexed by processor number.
- The system page table itself. The SPT is the last component to be mapped before memory management is enabled. The current virtual address becomes the value of KER$GA_SPT_BASE, and the value of KER$GA_SPT_PHYSICAL is used to determine the first page frame to be mapped.

Once these items have been mapped, the kernel begins the process of enabling memory management, as described in Section 3.2.3.

3.2.3 Enabling Memory Management

The kernel prepares to enable memory management by superimposing a temporary P0 page table over the portion of the S0 page table that maps the kernel's initialization code. When memory management is enabled, a single instruction is executed in P0 space to load the program counter (PC) with the S0 address of the next initialization instruction.

The instruction that transfers execution to S0 space is executed in P0 space because its physical address can be mapped to an identical P0 address. The range of P0 space addresses 0 to $3FFFFFFF_{16}$ also describes the maximum range of physical addresses supported by the VAX architecture. This guarantees that a P0 address will match the physical address of the instruction that initiates execution in system space.

Figure 3–3 shows how the kernel overlays the S0 page table with the temporary P0 page table, enables memory management, and transfers control to a system virtual address. The technique depends on displacing the base of the P0 page table from the start of the S0 page table to account for the page offset of the kernel from the start of physical memory. Because the system image is the first element mapped in the S0 page table, the first page of the kernel is mapped by the first SPTE. This displacement means that when the physical address of the first mapped instruction is interpreted as a P0 virtual address, its translation will again yield the physical address of that instruction.

The kernel sets up the temporary P0 page table with the following steps. The sample values presented in the discussion refer to those shown in Figures 3–4 and 3–5, which illustrate the relationship of the S0 and P0 page tables to the physical memory they map. These instructions execute with memory management disabled, so all memory references are to physical addresses.

Figure 3–3: Kernel Code That Enables Memory Management

```
        .
        .
        .
MAPEN_ENABLE:
        MOVAB   W^KER$VECTOR_START,R0   ; get physical address of kernel
        ASHL    #-VA$V_VPN,R0,R0        ; compute its base PFN
        MNEGL   R0,R1                   ; negate the PFN
        BLBS    W^KER$GB_RTVAX--        ; branch if KA620
                KER$GR_KERNEL_DATA(R8),10$
        MOVL    W^KER$GA_SPT_BASE--     ; get virtual address of SPT
                KER$GR_KERNEL_DATA(R8),R2
        BRB     20$   ; rejoin common code
10$:    MOVL    W^KER$GA_SPT_PHYSICAL-- ; get physical address of SPT
                KER$GR_KERNEL_DATA(R8),R2
20$:    MOVAL   (R2)[R1],R1             ; compute equivalent P0PT base address
        MTPR    R1,#PR$_P0BR            ; set base address of P0 page table
        ADDL3   AP,R0,R1                ; compute length of P0 page table
        MTPR    R1,#PR$_P0LR            ; set length of P0 page table
        INVALID_ALL                     ; invalidate entire translation buffer
                mtpr    #0,#PR$_TBIA
        MTPR    #1,#PR$_MAPEN           ; enable memory management
        JMP     @#MAPPED_EXECUTION      ; set PC to a system virtual address

MAPPED_EXECUTION:                       ; begin mapped execution...
        GETCPU  R0                      ; get current processor number
                clrl    R0
        .
        .
        .
```

Figure 3–4: Enabling Memory Management Through the Temporary P0 Page Table, Part 1

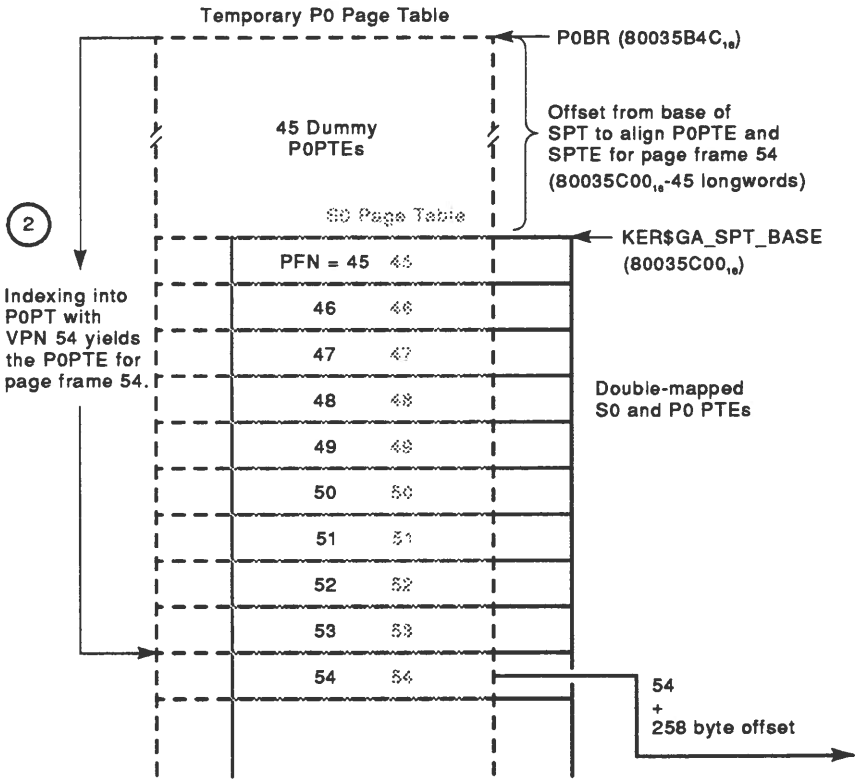
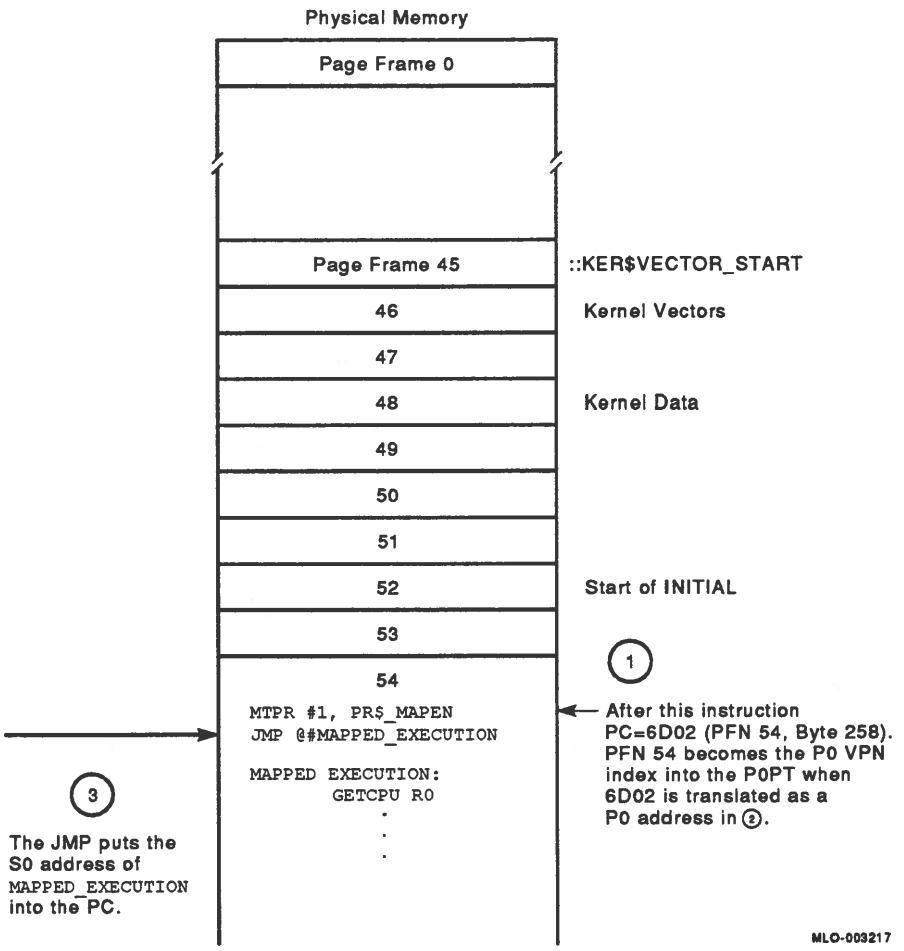


Figure 3–5: Enabling Memory Management Through the Temporary P0 Page Table, Part 2



1. The physical address of the kernel's base — represented by the global label KER$VECTOR_START — is obtained. In Figure 3–5, that physical address is $5A00_{16}$.
2. The PFN field of the physical address is extracted. This value represents the page offset of the base of the kernel image from the start of memory. In Figure 3–5, this means that the kernel starts on the 45th page frame.
3. The page frame offset determined in the previous step is negated.
4. For all processors but the KA620, the virtual address of the base of the S0 page table is obtained from the KER$GA_SPT_BASE field in the kernel data block. This value was calculated during the mapping of the S0 page table. In Figure 3–4, the SPT base is $800035C00_{16}$. On the KA620 processor, the physical address of the S0 page table is obtained instead from KER$GA_SPT_PHYSICAL.
5. The negated page frame offset of the kernel is subtracted from the base address of the SPT to yield the required base virtual address of the temporary P0 page table. In Figure 3–5, this value is $80035B4C_{16}$, 45 longwords — P0PTEs — before the base of the SPT. On the KA620 processor, the offset is subtracted from the physical base of the SPT to yield the physical address of the P0 page table.

   This is the critical step: when the VPN derived from the address of the JMP instruction is added to the base address of the P0 page table during address translation, the resulting P0PTE is also the SPTE holding the PFN of the page frame that contains the JMP instruction.
6. The virtual address of the base of the temporary P0PT is written to the P0BR for use during address translation. On the KA620 processor, the physical address of the P0PT is written to P0BR.
7. The kernel's page offset is added to the present physical size of the system to become the value of the P0LR.
8. The address translation buffer is invalidated.

The next three instructions in the sequence merit closer attention. Each instruction executes in a different address space — physical, P0, and S0 (the list numbers correspond to the numbers shown in Figures 3–4 and 3–5):

❶ The first of the three instructions is accessed by its physical address and enables mapping by writing a 1 to the Map Enable privileged register:

```
MTPR    #1,#PR$_MAPEN
```

After the execution of this instruction, all address references are translated. The program counter now contains the physical address of the next instruction (JMP) — in Figure 3–5, $6D02_{16}$, namely, byte 258 on page frame 54. That physical address is now interpreted as a P0 address when the processor fetches that instruction.

❷ The next instruction —

```
JMP     @#MAPPED_EXECUTION
```

— is accessed through the P0 virtual address in the program counter. The address is translated by extracting the VPN — in the figure, 54 — from the P0 address and using it as a longword index from the P0BR. This offset yields the system virtual address of the P0PTE that maps the page frame containing the JMP instruction. The temporary P0 page table has been superimposed on the S0 page table so that this P0PTE corresponds to the SPTE that maps page frame 54. When the byte offset is added back in, the physical address of the JMP instruction — $6D02_{16}$ — results.

The effect of the JMP instruction is to set the program counter with the virtual address of the next instruction, at the MAPPED_EXECUTION label. That address will be the S0 virtual address for that label, which was calculated when the kernel image was linked.

❸ The instruction at the MAPPED_EXECUTION label —

```
GETCPU  R0
```

— is executed in system address space (and has no connection with enabling memory management). From this point on, system initialization continues with memory management enabled, executing in system address space.

3.2.4 Mapped Initialization

With mapping enabled, the kernel can reference system data structures, such as the RPB, through their virtual addresses. For example, cells in the data block can now be referenced directly instead of through a displacement from the physical base of the data block.

The kernel takes the following steps to complete initialization:

1. Execution is switched to the interrupt stack. See Section 3.2.4.1.

2. The machine-check handler data block or blocks are initialized. See Section 3.2.4.2.
3. The SCB is initialized. See Section 3.2.4.3.
4. I/O address space is configured. See Section 3.2.4.4.
5. The processor-specific registers and console registers are initialized, and data used by the console subsystem is mapped. See Section 3.2.4.5.
6. The remaining system data structures are created, initialized, and mapped. See Section 3.2.4.6.
7. Scheduler and job queues are initialized. See Section 3.2.4.7.
8. The start-up job is created. See Section 3.2.4.8.
9. The system announcement string is displayed. See Section 3.2.4.9.
10. The system interval time is started. See Section 3.2.4.10.
11. The system start-up is logged in the system error log file. See Section 3.2.4.11.
12. Job scheduling begins, and initialization is complete. See Section 3.2.4.12.

3.2.4.1 Step 1 — Switch Execution to the Interrupt Stack

The stack pointer (SP) is loaded with the base address of the interrupt stack for the primary processor. Execution continues on the interrupt stack until the start-up job is created, when execution switches temporarily to the kernel stack.

3.2.4.2 Step 2 — Initialize the Machine-Check Data Block

For each processor in the system, the kernel calls the processor-specific subroutine KER$INIT_MACHINECHK to initialize the processor's machine-check handler data block.

3.2.4.3 Step 3 — Initialize the SCB

Until this point, the SCB has contained physical-address-based vectors to the boot-time exception and interrupt handlers. The SCB is now initialized with its run-time vectors. The VAXELN SCB structure is actually two components: the SCB itself and the unexpected-event dispatch block, which supports the handling of unexpected interrupts and exceptions.

The SCB and unexpected-event dispatch block are initialized in four passes:

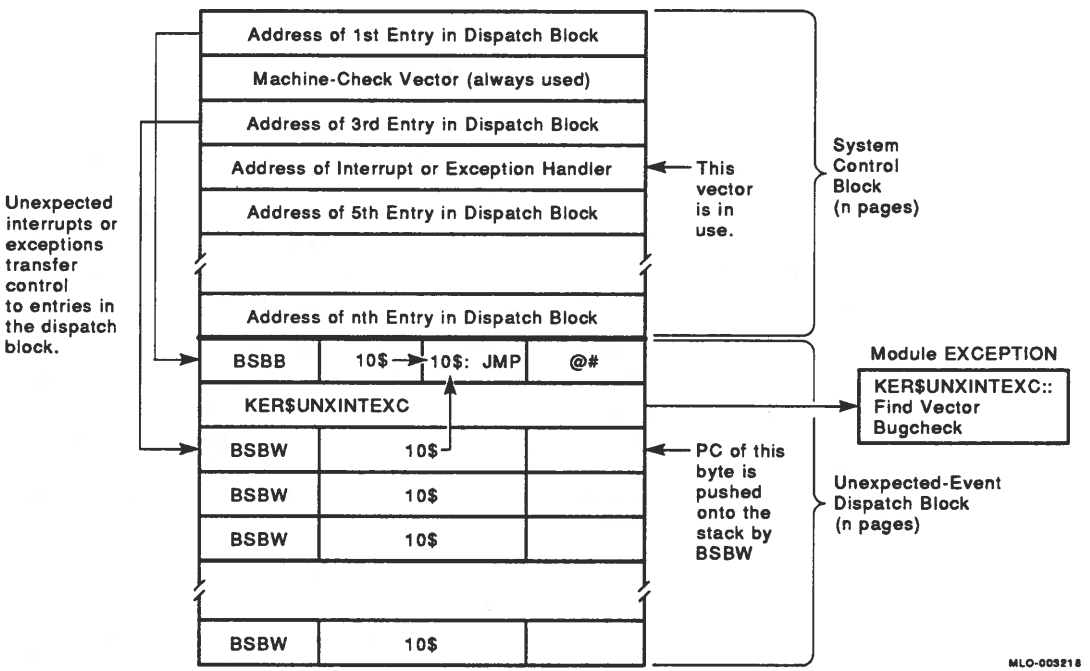
1. Each vector in the SCB is filled with the address of its corresponding entry in the unexpected-event block.
2. The architecturally defined and processor-specific portions of the SCB are filled in with the addresses of the appropriate interrupt and exception handlers. (The general layout of the SCB is described in Chapter 6, Condition Handling.) The architecturally defined vectors are copied from a table at the start of module INITIAL.

   Figure 3–6 shows the relationship between the SCB and the unexpected-event dispatch block. Any SCB vectors not touched on this second pass continue to point indirectly to the unexpected-event handler, KER$UNXINTEXC, in module EXCEPTION.

   Each of these unused vectors in fact contains the address of the corresponding entry in the unexpected-event dispatch block (that is, the *n*th vector points to the *n*th longword in the unexpected-event block). That unexpected-event entry contains a Branch to Subroutine with Word Displacement (BSBW) instruction and the displacement to the six-byte unexpected-interrupt dispatcher, which starts at the third byte of the block.

   When an unexpected interrupt or exception occurs, the SCB vector it uses tranfers control to the BSBW instruction in its corresponding entry in the unexpected-event dispatch block. The BSBW pushes the PC (the address of the byte following the displacement byte) onto the stack and transfers control to the unexpected-event dispatcher at the top of the block. The dispatcher then executes a JMP to the unexpected-event handler in module EXCEPTION, which uses the PC on the stack to determine which dispatch entry executed the BSBW. The offset of this entry in the block corresponds to the SCB vector that received the unexpected interrupt or exception. This result is printed at the top of the stack dump when the handler declares a system fatal bugcheck to halt the system.
3. Processor-specific SCB vectors are initialized. The address of a table containing processor-specific vectors is returned by the internal subroutine KER$SCB_FIXUPS.

Figure 3–6: Relationship Between the SCB and the Unexpected-Event Dispatch Block



4. The addresses of the string and floating-point instruction emulator routines are loaded into the appropriate SCB vectors if emulation was requested.

3.2.4.4 Step 4 — Configure I/O Address Space

The kernel configures I/O address space by calling the processor-specific subroutine KER$CONFIGURE_IOSPACE, which resides in the appropriate INIT*nnn* module for the target processor. For busless and Q-bus-based processors, the subroutine simply returns. On the VAX 62*nn* and VAXBI-based processors, the subroutine probes the I/O bus for I/O adapters, initializes them, and saves information about their configuration to be used during later operations.

3.2.4.5 Step 5 — Initialize Processor-Specific and Console Registers

The kernel initializes the processor-specific registers that were mapped earlier by calling the processor-specific subroutine KER$INIT_PROCREG. The kernel then calls KER$CONIO_INITIAL again to update the value of KER$GA_CONIO_CODE with the virtual address of the console I/O image, replacing the physical address used to call console I/O procedures during unmapped initialization. This call to KER$CONIO_INITIAL also results in the mapping of the console registers and data.

3.2.4.6 Step 6 — Create and Map Remaining System Structures

At this point, the kernel creates and maps the remaining system data structures whose sizes were calculated before the creation of the system page table. To allocate memory and map the structures, the kernel calls the local subroutine GET_FRAME for each page required. GET_FRAME in turn calls the internal subroutine KER$ALLOCATE_FRAME and then maps the returned page frame number into the system page table, advancing the current system virtual and SPTE addresses as each page is mapped.

When I/O address space is mapped, however, no page frames are allocated; rather, the address space is mapped to the appropriate processor-specific physical addresses. In these instances, the local subroutine KER$FILL_SPTE is called.

The lower half of Figure 3–2, below the system page table, shows the elements mapped at this stage of initialization. The kernel creates and maps the following system data structures:

- The crash-restart log. One log area is created and mapped for each processor in the system, and the base address of each log is written to the array located at KER$GA_CRASHLOG (indexed by processor number).
- The pool of error-log buffers. If error logging has been enabled (KER$GB_ERRLOG_ENABLE is 1), the kernel creates and maps the number of error message buffers specified as KER$GW_EMB_COUNT. This process is described in Section 7.1.2.1.1, Error Message Buffers.
- The system dynamic pool. The system pool is allocated and mapped as described in Section 9.4.1, Initializing System Pool.

- The P0 and P1 page table slots. Physical memory is allocated to hold the P0 and P1 page table slot bitmaps, but no page frames are allocated for the page tables themselves. Instead, physical memory is allocated for page tables at the time of job and process creation, as described in Section 9.1.2.1.2, P0 Page Tables, and Section 9.1.2.1.3, P1 Page Tables. The slot bitmap descriptors located at KER$GR_P0_SLOT_BITMAP and KER$GR_P1_SLOT_BITMAP are also initialized, and the SPTEs that map the area reserved for the page table slots are cleared. The virtual addresses of the P0 and P1 slot areas become the values of KER$GA_P0_SLOT_BASE and KER$GA_P1_SLOT_BASE, respectively.
- The port address table. The port address table, whose size in longword table entries is indicated by KER$GW_PORT_SIZE, is initialized as described in Section 10.2.1.2, Port Address Table. The current virtual address becomes the value of KER$GA_PORT_BASE.
- The local name table. The local name table, whose size in quadword entries is indicated by the global parameter KER$GW_NAME_SIZE, is initialized as described in Section B.12. The current virtual address is used to initialize the table descriptor located at KER$GR_LOCAL_TABLE. The NTB$A_ADDRESS1 field in the descriptor contains the address of the first listhead, and NTB$L_LENGTH1 contains the number of listheads in the table (128).
- The local debugger data area. If the local debugger component is present, the kernel maps and zeros the number of page frames specified by KER$GA_KERNEL_DEBUG_DATA to support the debugger's read/write data. The current virtual address then becomes the new value for KER$GA_KERNEL_DEBUG_DATA.

  The kernel next calls the local debugger's initialization code as a subroutine at the address specified by KER$GA_KERNEL_DEBUG_CODE. If the user has requested the initial kernel breakpoint (RPB$V_INIBPT is set), then the subroutine KER$KERNEL_BREAK is called to give the local debugger control in a kernel debugging session.
- I/O space. At this point, the kernel displays the system startup message: "%VAXELN system initializing."

  Next, the kernel calls the processor-specific subroutine KER$MAP_IOSPACE to map the I/O address space. If the target processor has one or more hardware adapters, the subroutine walks the list of system configuration records (SCRs), whose address is calculated using the offset value in KER$GA_DEVICE_LIST, and analyzes them to determine what I/O adapters have been configured for the

system. For each adapter the subroutine encounters, an adapter control block (ADP) is created, initialized, and inserted into the adapter list located at KER$GA_ADAPTER_LIST. The structure of the ADP is described in Table B–2.

The number of virtual pages that are mapped in the system page table depends on the processor and its adapter configuration. For example, on the Q22-bus-based MicroVAX II, the following elements are mapped:

- Q22-bus I/O space
- Q22-bus map registers
- The allocation bitmap for map registers

On a VAXBI-based processor, by contrast, VAXBI nodespace is mapped, and, if a UNIBUS adapter resides on the VAXBI bus, the following UNIBUS elements are mapped as well:

- UNIBUS I/O space
- UNIBUS adapter space
- The allocation bitmap for UNIBUS map registers

No physical memory is allocated during this mapping, and no bits are cleared in the PFN bitmap (physical memory is allocated for any bitmaps that are created). These I/O pages are simply mapped into virtual address space using their bus-specific physical addresses. For example, on many VAX processors, I/O space is mapped into system space at physical address 20000000_{16} or above.

- The communication region. Based on the value of KER$GW_IO_SIZE, the kernel computes the number of pages required to hold the communication region bitmap. It then allocates and maps those page frames and initializes the bitmap, located at KER$GR_REGION_BITMAP (see Section 9.3.1, Allocating System Virtual Memory). The current virtual address then becomes the value KER$GA_REGION_BASE. Finally, the SPTEs that will map the communication region are cleared to show that no pages are yet allocated there. These are the last entries in the system page table to be initialized.

At this point, system virtual memory has been completely mapped.

3.2.4.7 Step 7 — Initialize Scheduler and Job Queues

The queue of ready jobs, located at KER$AQ_READY_HEAD, contains a job listhead for each possible job priority. The kernel initializes the queue listheads by writing the address of each quadword listhead to its forward and backward link fields.

The kernel also initializes the scheduler's mask of idle processors, located at KER$GW_CPU_IDLE, by setting the bit that indicates that the primary processor is idle. This means that the start-up job will be scheduled to run on the primary processor in the system.

Finally, the kernel initializes the listheads for the queue of jobs in the system and the list of area control blocks.

3.2.4.8 Step 8 — Create the Start-Up Job

The start-up job is the first job to run in any VAXELN system. It scans the program list created by the System Builder and creates the jobs in the list that have the **Init required** and **Run** attributes. The execution of the start-up job is described in Section 3.3.

To create the start-up job, the kernel calls the KER$CREATE_JOB procedure. The call is vectored through the change-mode dispatcher (described in Chapter 8), which executes the CHMK instruction to transfer control to the procedure code. Since executing CHMK on the interrupt stack is illegal, the kernel must first switch execution to the kernel stack. Because the kernel executes a Save Process Context (SVPCTX) instruction to return to the interrupt stack, it must create a temporary hardware process context block. The kernel also takes steps to prevent the start-up job from executing immediately after its creation.

The start-up job is created as follows:

1. A temporary process hardware context block (PTX) is allocated on the stack. The minimum number of fields is initialized in the PTX. The address of this PTX is written to the PCBB (hardware context base) internal register. The PTX block is required so that the kernel can later execute the SVPCTX instruction to return execution to the interrupt stack.
2. IPL is lowered from 31 to 3 (IPL$K_DISABLE_SWITCH).

3. A pool block is allocated to act as a dummy, and the minimum number of fields is initialized in the JCB. The JCB address is written to the KER$AA_CURRENT_JCB array entry for the current processor. This prevents the scheduler from allowing the start-up job to run until the kernel removes the dummy JCB from KER$AA_CURRENT_JCB and invokes the scheduler.
4. The current PSL is pushed onto the stack, and the high byte is zeroed, clearing the interrupt stack bit (IS) and setting the current mode field to kernel. The kernel then pushes the PC of the next instruction onto the stack by calling a subroutine that executes only an Return from Exception or Interrupt (REI) instruction. The execution of the REI then causes the PC and PSL on the stack to become current, effectively switching execution to the kernel stack at IPL 3, beginning at the instruction following the subroutine call. The kernel stack pointer is then initialized with the address of the interrupt stack, so that same stack as before is used. The kernel is now executing in kernel mode to create the start-up job.
5. The address of a prepared program descriptor, located at KER$GR_STARTUP (in module STARTUP), is written to the data cell KER$GA_PROGRAM_LIST. This makes the start-up job's descriptor the only one in the program list, where KER$CREATE_JOB will find it. The program descriptor specifies that the start-up job will run in kernel mode with job and master process priorities of 0.
6. The arguments to KER$CREATE_JOB are pushed onto the stack. Arguments for the exit port, a null program name, the job port, and the return status are specified. The null program name in the argument list will match the name in the program descriptor when KER$CREATE_JOB searches the list for the specified program.
7. The KER$CREATE_JOB procedure is called with the CALLS instruction and the procedure entry point KER$STARTUP. The procedure will attempt to create the start-up job and return a completion status. KER$CREATE_JOB is described in Chapter 4.
8. The status returned from KER$CREATE_JOB is examined. If it indicates failure, a fatal bugcheck is raised to halt the system. Otherwise, the start-up job has been created and awaits execution in the ready job queue.
9. The original values for KER$GA_PROGRAM_LIST and KER$AA_CURRENT_JCB are restored, and the dummy JCB pool block is returned to the pool. Removing the JCB address from KER$AA_CURRENT_JCB allows the start-up job to run once the scheduler is called.

10. The SVPCTX instruction is executed to return execution to the interrupt stack for the completion of initialization. The temporary PTX, which contains the unneeded process context information, is cleared from the stack.

3.2.4.9 Step 9 — Announce the System

The kernel calls the generic interface to the console I/O subsystem to display the VAXELN announcement string — for example, "VAXELN V4.0 QBUS" — on the console terminal.

3.2.4.10 Step 10 — Start the Interval Clock

The kernel initializes the timer queue listhead located at KER$GQ_TIME_QUEUE. It then sets the processor's interval clock to interrupt at the interval specified by KER$GL_TIME_INTERVAL and starts the clock by writing to the PR$_ICCS register. Section 5.3.1 describes the function of the interval clock in VAXELN timekeeping.

3.2.4.11 Step 11 — Log the System Start-Up

The kernel calls the internal subroutine KER$COLDSTART. If error logging is enabled for the system, a system start-up entry is posted to the system's error log file. At this time, the processor's cold and warm start flags are also cleared.

3.2.4.12 Step 12 — Begin Job Scheduling

The kernel completes its initialization by setting IPL to 8 (IPL$K_SYNCHRONIZE) and invoking the scheduler by transferring control to internal subroutine KER$SCHEDULE_JOB. The scheduling pass this precipitates allows the start-up job to run. Control will not return to module INITIAL. The execution of the start-up job is described in Section 3.3.

3.3 Application Start-Up: The Start-Up Job

The VAXELN initialization sequence ends by invoking the kernel's scheduler. The scheduler finds only one job to be scheduled: the start-up job. The start-up job (in module STARTUP) performs the following tasks:

- Begins the execution of applications by creating jobs with the **Init required** and **Run** characteristics
- Configures and boots auxiliary processors in a tightly coupled symmetric multiprocessing system
- Initializes the bus-based message port in a closely coupled symmetric multiprocessing system

The first item, since it concerns the start-up of applications on every VAXELN system, is the focus of Section 3.3.1. Section 3.3.2 describes a related kernel procedure, KER$INITIALIZATION_DONE, which allows initialization jobs to inform the start-up job that their initialization is complete.

3.3.1 Creating Jobs Sequentially

When the System Builder created the program list, it arranged the system's program descriptors in the following order:

1. Programs that require initialization at system start-up. These are the programs for which the **Init required** characteristic has been selected in their program descriptions (that is, the PRG$V_SEQ_INITIAL bit is set in the PRG$B_OPTION_FLAGS field). These initialization jobs are sorted by job priority, with the highest priority job first in the list.
2. Programs that must be created at system start-up. These are the programs for which the **Run** characteristic has been selected in their program descriptions (that is, the PRG$V_AUTO_START bit is set in the PRG$B_OPTION_FLAGS field). These jobs are also placed in priority order.
3. All other programs. Programs that require neither initialization nor automatic start-up appear at the end of the program list in the order they were processed by the System Builder.

The start-up job runs in kernel mode at IPL 0, with job and master process priorities of 0. It allows lower priority initialization jobs to run by blocking its own execution. The start-up job's task is to walk the program list and perform the following operations:

1. Create all jobs that require initialization. After each job is created, the start-up job waits until the job calls KER$INITIALIZATION_DONE or exits.
2. Create all jobs that require automatic start-up. These jobs cannot run until the start-up job exits.
3. Exit to allow normal job scheduling to occur.

To pass the necessary arguments to KER$CREATE_JOB, the start-up job reserves the required space on the stack, pushes its arguments there, and calls KER$CREATE_JOB, specifying the address of the top of the stack region as the argument table operand for the CALLG instruction. If a failure status is returned by any procedure call within the start-up job, IPL is raised to 2 (IPL$K_AST_LEVEL), and a system fatal bugcheck is declared to cause the orderly shutdown of the system.

Communication between the start-up job and initialization jobs is through the start-up job's job port. The start-up job first obtains the identifier of this port by calling KER$JOB_PORT, obtains the address of that port by calling the KER$TRANSLATE_OBJECT subroutine, and stores that address in the global cell KER$GA_STARTUP_PORT. The start-up job blocks its execution — allowing an initialization job to run — by waiting on this port for a message. The message signifies that one of two possible events has occurred:

- The job has completed its initialization and called KER$INITIALIZATION_DONE. This message comes directly from KER$INITIALIZATION_DONE, which has obtained the identifier for the start-up job's port from KER$GA_STARTUP_PORT.
- The job has exited. This message comes from KER$CREATE_JOB (through a call to KER$DELETE), which obtains the port identifier as the exit port argument passed to it by the start-up job.

Before the start-up job exits, it removes the address of its job port from KER$GA_STARTUP_PORT. This will signify, to subsequent calls to KER$INITIALIZATION_DONE, that the start-up job no longer exists.

After obtaining the value of its job port, the start-up job next walks the program list from beginning to end looking for program descriptors that have both the PRG$V_SEQ_INITIAL and PRG$V_AUTO_START bits set in the PRG$B_OPTION_FLAGS field. Notice that a program must have both the **Run** and **Init required** characteristics before it can actually run. If such a job is found, control branches to a local subroutine to create the job and allow it to run. The search for initialization jobs completes at the end of the program list.

Next comes a second pass through the program list. This time, the start-up job is looking for descriptors with the PRG$V_AUTO_START bit set and the PRG$V_SEQ_INITIAL bit clear. If it finds such a job, control again transfers to the local subroutine to create the job. This time, however, the newly created job is not allowed to run. The search for auto-start jobs completes at the end of the program list.

The local subroutine creates initialization jobs and auto-start jobs as follows:

1. The CALLG argument table for KER$CREATE_JOB is built on the stack as follows:
   a. The job parameters in the program descriptor are extracted from their parameter blocks and pushed onto the stack as string descriptors, as KER$CREATE_JOB expects them.
   b. If this is an initialization job, the address of the start-up job's port is pushed onto the stack as the exit port argument. Otherwise, 0 is pushed as a null argument for auto-start jobs, since these jobs do not communicate with the start-up job.
   c. The program name and size are extracted from the program descriptor. They are then fashioned into a string descriptor, whose address is pushed onto the stack as expected by KER$CREATE_JOB.
   d. The address for the status value is pushed onto the stack.
   e. The final argument count is pushed onto the stack.
2. The KER$CREATE_JOB procedure is called. If a success status is returned, execution continues. Otherwise, a system fatal bugcheck occurs.
3. If an auto-start job was created, the subroutine returns. The auto-start job will not be able to run until the start-up job exits.

4. The identifier for the job port is pushed onto the stack, and KER$WAIT_ANY is called. By explicitly blocking its own execution, the start-up job allows the initialization job to run until it completes or until it calls KER$INITIALIZATION_DONE and, possibly, blocks itself. The start-up job will not run again until a message arrives on its job port.
5. The KER$RECEIVE procedure is called to receive the message. The reply port argument is included so that the message's sender can be identified.
6. If the identifier of the reply port is the same as the initialization job just created, the job initialization is done, and the subroutine exits. The message arrived because the job exited or called KER$INITIALIZATION_DONE.
7. If the port identifiers do not match, the message signifies the termination of an earlier initialization job that did not call KER$INITIALIZATION_DONE. Therefore, the start-up job waits again on its job port for a message from the job it just created. No new jobs will be created until such a message arrives.

3.3.2 Job Initialization and KER$INITIALIZATION_DONE

When a job completes its initialization, it can call the KER$INITIALIZATION_DONE procedure to inform the start-up job and allow other jobs to be created. KER$INITIALIZATION_DONE sends a message to the start-up job, which is waiting on its job port. When the message arrives, the start-up job unblocks and continues its search of the program list for other initialization jobs.

KER$INITIALIZATION_DONE executes as follows:

1. The address of the caller's JCB is obtained.
2. The address of the caller's program descriptor is obtained from JCB$A_PROGRAM.
3. The PRG$V_SEQ_INITIAL bit in the PRG$B_OPTION_FLAGS field is tested. If it is clear, the calling job does not have the **Init required** attribute, and the failure status KER$_NO_INITIALIZATION is returned to the caller.
4. The value of KER$GA_STARTUP_PORT is obtained. If that value is not an S0 virtual address — the address of the start-up job's job port — KER$_SUCCESS is returned to the caller. This means that KER$INITIALIZATION_DONE was called after application start-up was completed and therefore has no function.

5. The KER$CREATE_MESSAGE procedure is called to create a zero-length message. If the procedure fails, that failure status is returned to the caller.
6. The KER$SEND procedure is called to send the null message to the start-up job's port. The arrival of this message unblocks the start-up job. If the send procedure fails, that failure status is returned to the caller.

Chapter 4

Job and Process Creation and Deletion

This chapter describes the creation of VAXELN jobs and processes, the environment established for program execution when a job is created, and the deletion of jobs and processes.

For each executable program image included in a VAXELN system image at build time or loaded dynamically at run time, one or more jobs can be created. A job represents the activation of a program in the system. Each job is logically independent of other jobs (including other activations of the same program image) and executes concurrently with other jobs in the system. This is the VAXELN multiprogramming environment.

If the program is marked with the *Run* characteristic on the Program Description Menu, the kernel creates a job to run the program at system start-up; otherwise, a job is created by an explicit run-time procedure call from a running program or interactive utility (the debugger or the VAXELN Command Language).

Creating a VAXELN job establishes an environment for execution of a program image by one or more processes. At job creation, the kernel creates a master process to execute the program's main code, beginning at its transfer address. Subsequently, subprocesses can be created as needed to execute portions of the program code (process blocks or routines, compiled as VAX procedures). The job's master process and subprocesses concurrently execute the same or different portions of a program image, with each process representing a logically independent thread of execution. This is the VAXELN multitasking environment.

During job creation, the kernel creates a framework of interconnected control blocks and other data structures that represent the state of the job and its processes at any given time. Among the significant structures that support program execution are the job control block (JCB), representing jobwide elements of process software context, shared among all processes in a job; the process control block (PCB), representing process-specific elements of process software context, private to a process; the process hardware context block (PTX), representing process hardware context; and the page tables that map job and process components to the program (P0) and control (P1) regions of process address space. These structures are described in Section 4.2. The kernel procedure that creates a job, KER$CREATE_JOB, is described in Section 4.4.

Creating a process establishes an independent thread of program execution within the system. A VAXELN process is a VAX process, as defined by the VAX architecture, with the added characteristic that it shares the P0 region of virtual address space, where the program image is mapped, with all other processes executing in the job.

During process creation, the kernel creates process-specific control blocks and other data structures that are added to the job's framework of data structures. These structures include a PCB, a PTX, and a page table mapping the new process's components and resources to its private P1 address space.

A subprocess is created with a call to the kernel procedure KER$CREATE_PROCESS, described in Section 4.5. The master process is created implicitly when a job is created; see Section 4.4.

Deleting a process deactivates an execution thread within the system. If the deleted process is the master process, the entire job is terminated, its subprocesses are deleted, and its system resources are deallocated. If the process's termination began with an implicit exit (the process reached the end of its code) or with an explicit call to KER$EXIT from program code, additional orderly cleanup is performed before the process is deleted.

The procedure that deletes object-related kernel resources, KER$DELETE, is described in Chapter 10. Section 4.6 describes how KER$DELETE deletes processes; it also describes the kernel procedure that provides orderly process and job termination, KER$EXIT.

4.1 Process Execution Environment

The environment in which a process executes is defined by the states and contents of its supporting data structures, its address space, and its registers. Figure 4–1 summarizes this context for a process's execution.

The control structures established by the kernel at job and process creation reside in the system (S0) region of virtual address space, where they can be modified directly only by the kernel (which manages them for the user) or by a process executing in kernel access mode. The control structures occupy system pool blocks, page table slots, or pages allocated from the system's communication region.

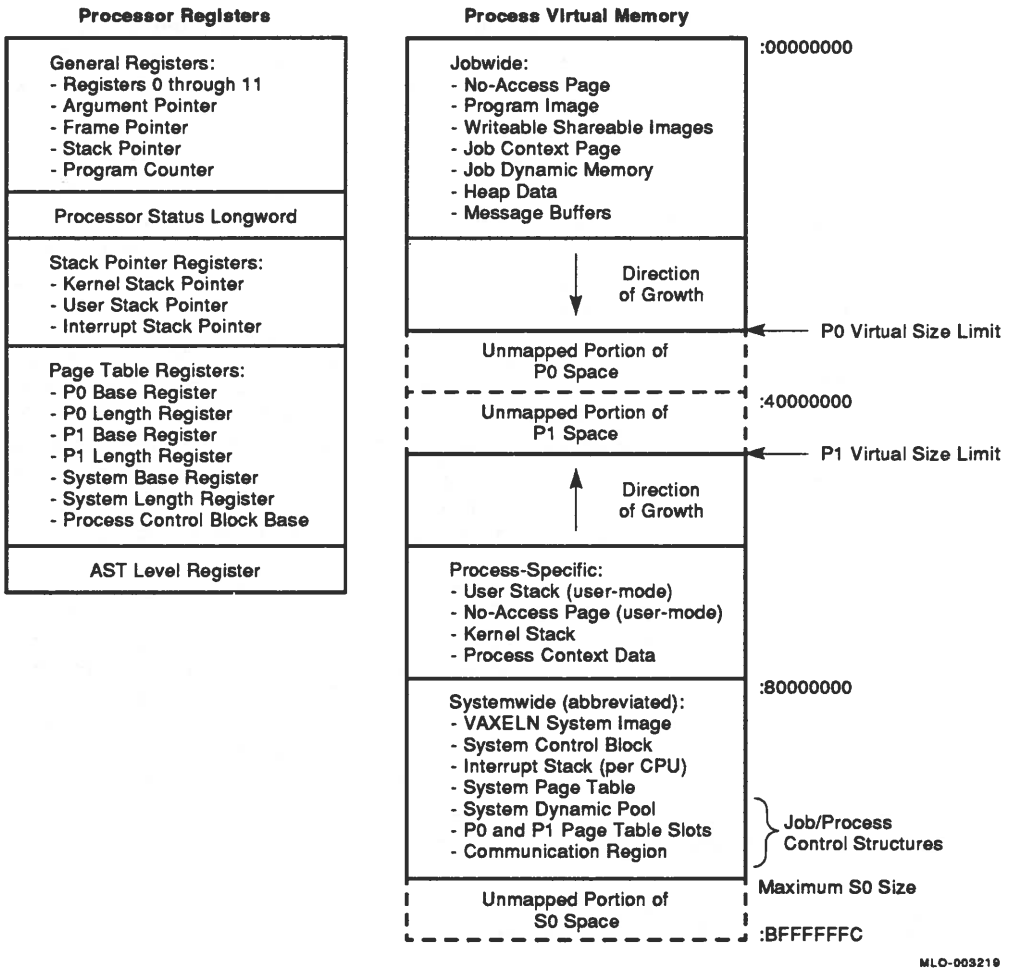
Job components, including program code and global data, are mapped into the P0 region of process address space, for jobwide access. Process components, including the process stacks, are mapped into the P1 region of process address space, for private access. Job and process resources can reside in process address space — for example, dynamically allocated P0 or P1 virtual memory — or in system address space — for example, kernel objects or dynamically allocated S0 virtual memory.

When a process is selected by the kernel's scheduler to run, a LDPCTX (Load Process Context) instruction is executed to load the process hardware context, as defined by the process's PTX, into the processor registers that support process execution. Among those registers are the P0 base register (P0BR), the P0 length register (P0LR), the P1 base register (P1BR), and the P1 length register (P1LR), which receive the base addresses and effective lengths of the job's P0 page table and the process's P1 page table. These registers define the executing process's P0 and P1 virtual address space, and are accessed to translate virtual addresses referenced by the program.

4.2 Job and Process Data Structures

At job creation, the kernel creates a framework of interconnected data structures to represent the components, resources, attributes, and state of the job and its processes at any given time. For each process subsequently created by the job, the kernel adds data structures specific to that process to the framework.

Figure 4–1: Execution Context of a Process



The job data structures established at job creation include the following:

- Job control block (JCB). This structure represents the attributes and state of the job and contains pointers to its components and resources. The JCB also is linked into the kernel's queue of all jobs in the system and, if it is in the ready state, into the ready

job scheduling queue for its priority. The JCB is described in Section 4.2.1.

- Job parameter blocks (JPBs). These structures store caller-specified job arguments during the first phase of job creation; ultimately job arguments are copied to P0 virtual memory, where all the job's processes can access them. JPB format is described in Section 4.4.1.2.
- Job object tables. These structures comprise a two-tiered arrangement of address tables — an object base table and one or more object pointer tables — used to create, locate, and delete dynamically created kernel objects, representing job resources. Job object tables are described in detail in Chapter 10.
- P0 page table (P0PT) and related memory-management structures such as allocation bitmaps and page table entries (PTEs). In conjunction with the processor page table registers (when the job is running) or the page table fields of the job's PTXs (when the job is not running), the P0PT defines and maps the P0 virtual memory for the job. Page tables and related structures are described in Chapter 9; the job components and resources mapped into P0 virtual address space are described in Section 4.3.1.
- Master process data structures. These process structures are private to the master process.

The process data structures established at process creation (for master and subprocesses alike) include the following:

- Process control block (PCB). This structure represents the attributes and state of the process and contains pointers to its components and resources. The PCB is entered into the job's object database as the object representing the process. The PCB also is linked into the job's queue of all its processes and, if it is in the ready state, into the ready process scheduling queue for its priority. The PCB is described in Section 4.2.2.
- Process hardware context block (PTX). This structure represents the process's hardware context, consisting in part of the saved values of the internal registers that support process execution. A LDPCTX instruction loads these register values from the PTX when a process is scheduled to run, and a SVPCTX (Save Process Context) instruction returns these values to the PTX when the process is removed from execution. The PTX is described in Section 4.2.3.
- Wait control blocks (WCBs). These structures allow a process to wait for kernel objects. WCBs are described in Chapter 11.

- Process argument block. This structure is a standard VAX argument list containing the arguments specified when the process was created.
- P1 page table (P1PT) and related memory management structures such as allocation bitmaps and page table entries (PTEs). In conjunction with the processor page table registers (when the process is running) or the page table fields of the PTX (when the process is not running), the P1PT defines and maps a private P1 virtual address space for the process. A unique P1PT is created for each process in a job. Process components and resources mapped into P1 memory are described in Section 4.3.2.

4.2.1 Job Control Block

A job control block (JCB) is created by the KER$CREATE_JOB kernel procedure for each job created in a VAXELN system. The JCB represents and associates the attributes, state, resources, and components of a job and its family of processes. As described in Section 4.4, the JCB is created at the beginning of two pages allocated for job and process control blocks in the communication region of S0 address space. The following information is maintained in its fields:

- Links into the job scheduling queue for its job priority and into a linked list of all the system's jobs
- Pointers to the job's process queues, port queue, and object tables
- Pointers to the program descriptor for the program the job was created to execute, and to a list of program arguments
- Pointer to the job's P0 page table, and the size in pages that the page table maps

Figure 4–2 shows the structure of the JCB, and Table 4–1 describes its fields.

Figure 4–2: Structure of a Job Control Block

JCB$A_SCHEDULE_FLINK			
JCB$A_SCHEDULE_BLINK			
JCB$B_STATE	JCB$B_PRIORITY	JCB$B_READY_PRIORITY	JCB$B_TYPE
JCB$A_CURRENT_PCB			
JCB$A_NEXT_PCB			
JCB$A_PROCESS_QUEUES			
JCB$A_PROCESS_FLINK			
JCB$A_PROCESS_BLINK			
JCB$A_JOB_FLINK			
JCB$A_JOB_BLINK			
JCB$A_PORT_FLINK			
JCB$A_PORT_BLINK			
JCB$W_READY_SUMMARY		JCB$W_DISABLE	
JCB$W_OBJECT_FREE		JCB$W_CPU_MASK	
JCB$A_OBJECT_TABLE			
JCB$A_PROGRAM			
JCB$L_RW_DATA_PTE			
JCB$L_MESSAGE_PTE			
JCB$A_PARAMETER_LIST			
JCB$A_INITIAL_STACK			
JCB$W_PROCESS_GENERATION		JCB$W_GENERATION	
JCB$L_PORT_ID			
JCB$A_P0_BASE			
JCB$L_P0_LIMIT			
JCB$A_P0_BITMAP (12 bytes)			
JCB$B_EXIT_PORT_ID (16 bytes)			
JCB$W_CONTEXT_COUNT		JCB$B_CPU_NUMBER	JCB$B_MODE

MLO-003220

Table 4–1: Job Control Block Fields

Field	Meaning
JCB$A_SCHEDULE_FLINK JCB$A_SCHEDULE_BLINK	The forward and backward links to next and previous jobs in the ready queue for this job's priority.
JCB$B_TYPE	The structure type: OBJ$K_JOB.
JCB$B_READY_PRIORITY	The priority of the highest priority ready process in the job.
JCB$B_PRIORITY	The job's priority: 0 (highest) to 31 (lowest).
JCB$B_STATE	The job's state: JCB$K_READY (0), JCB$K_RUNNING (1), or JCB$K_WAITING (2).
JCB$A_CURRENT_PCB	The address of the PCB of the job's current process — the process placed in the running state for this job. The process is actually running only if the job is running.
JCB$A_NEXT_PCB	The address of the PCB of a process that has been selected to become the running process for this job, preempting the current process; if an address is present (a nonzero value), this field indicates that a rescheduling is pending.
JCB$A_PROCESS_QUEUES	The address of a set of 16 ready process queues, ordered by process priority (0–15), containing the PCBs of all the job's ready processes; this address is indexed to insert or remove a process from its priority queue.
JCB$A_PROCESS_FLINK JCB$A_PROCESS_BLINK	The listhead for the job's queue of processes. Each process in the job is inserted at process creation and removed at process (or job) deletion. Because the first entry is the job's master process, the JCB$A_PROCESS_FLINK address is used by the kernel to locate or identify the master process PCB. The queue is walked during P0 page table expansion, to update the P0 length register value in each process's PTX, and when certain asynchronous exceptions are raised against a process in the job.
JCB$A_JOB_FLINK JCB$A_JOB_BLINK	The forward and backward links to next and previous jobs in the system's job queue; each job is inserted at job creation and removed at job deletion.
JCB$A_PORT_FLINK JCB$A_PORT_BLINK	The listhead for the job's port queue. Ports are inserted and removed from the queue as they are created and deleted by the job. All ports are deleted when the job is deleted.

Table 4–1 (Cont.): Job Control Block Fields

Field	Meaning
JCB$W_DISABLE	The process-switching disable count, which is increased by 1 for each disable request and decreased by 1 for each enable request; 0 indicates that process switching is enabled.
JCB$W_READY_SUMMARY	The summary mask of the job's ready process queues. Each set bit in the mask indicates a nonempty ready process queue; bit *n* represents the ready queue for process priority *n*.
JCB$W_CPU_MASK	The complement mask of processors on which the job is eligible to run; set bits indicate ineligible processors in a tightly coupled multiprocessor configuration. Bit *n* represents processor *n*.
JCB$W_OBJECT_FREE	The encoded value indicating the location of the next available entry in the object pointer tables. The value is updated during each kernel object creation and deletion.
JCB$A_OBJECT_TABLE	The address of the job's object base table, which is indexed, when locating up a kernel object, to get an object pointer table address.
JCB$A_PROGRAM	The address of the job's program descriptor (PRG), used to look up program characteristics.
JCB$L_RW_DATA_PTE	The prototype page table entry for the creation of PTEs for the job's read/write data. The valid, protection (PTE$C_UW), owner (the program mode), and type (PTE$K_RW_DATA) fields are present in the prototype PTE. During P0 memory allocation (global data, heap, and KER$ALLOCATE_MEMORY) and P1 user stack allocation, the allocated page frame number is inserted into the PFN field to create the actual PTE for the page.
JCB$L_MESSAGE_PTE	The prototype page table entry for the creation of PTEs for the job's message and area data buffers. The valid, protection (PTE$C_UW), owner (the program mode), and type (PTE$K_MESSAGE) fields are present in the prototype PTE. During the allocation of message and area buffers, the allocated page frame number is inserted into the PFN field to create the actual PTE for the page.

Table 4–1 (Cont.): Job Control Block Fields

Field	Meaning
JCB$A_PARAMETER_LIST	A pointer to a linked list of the job's program arguments, as specified in the KER$CREATE_JOB procedure call. (If the job is created at system start-up, the supplied arguments are from the Program Description Menu.) Early in job creation, each program argument is entered into a separate job parameter block (JPB) and linked into a list; subsequently the arguments are copied to P0 memory to allow jobwide access. JPB format is described in Section 4.4.1.2.
JCB$A_INITIAL_STACK	The address of the initial stack top — user or kernel — matching the job's program mode. The initial kernel stack top is at P1 location P1$K_KERNEL_STACK_INIT ($7FFFFDF0_{16}$); the initial user stack top is at the P1 location –512 bytes offset from the end of the kernel stack. This field is also used for resetting the stack pointer in case of stack errors.
JCB$W_GENERATION	The job generation number, recorded when the job is created; the number is generated from the kernel value KER$GW_JOB_GENERATION. Jobs created in the system are numbered consecutively upwards from 1 in the order created.
JCB$W_PROCESS_GENERATION	The process generation number; incremented each time a process is created and recorded in the new process's PCB (PCB$W_GENERATION). Processes created in the job are numbered consecutively upwards from 1 in the order of their creation.
JCB$L_PORT_ID	The identifier for the job port.
JCB$A_P0_BASE	The system (S0) virtual address of the P0 page table, set during job creation when the page table is created. This value is used to locate the P0 page table during page table expansion. The value corresponds to the value of the P0BR.
JCB$L_P0_LIMIT	The count of PTEs in the P0 page table. This value is updated during the expansion of the P0 page table and corresponds to the value of the P0LR.
JCB$A_P0_BITMAP	The descriptor for the P0 virtual page allocation bitmap. This descriptor is used to locate and update the P0 bitmap during the allocation of P0 virtual memory. The format of the bitmap descriptor (BMP) is described in Chapter 9.

Table 4–1 (Cont.): Job Control Block Fields

Field	Meaning
JCB$B_EXIT_PORT_ID	The identifier for the job's exit port, if one was specified; otherwise 0.
JCB$B_MODE	The access mode of the program, as specified in the job's program descriptor: 0 for kernel mode, 3 for user mode. This value becomes the value of the ownership field in PTEs allocated for the job.
JCB$B_CPU_NUMBER	The number of the target processor, in a tightly coupled multiprocessor configuration, on which the job is running or last ran. If a job is selected to run and the processor on which it last ran is idle, the job is selected to run on the same processor.
JCB$W_CONTEXT_COUNT	The number of times a job enters the run state.

4.2.2 Process Control Block

A process control block (PCB) is created by the KER$CREATE_PROCESS kernel procedure for each subprocess created in a VAXELN system. In addition, a PCB is created by the KER$CREATE_JOB kernel procedure for a job's master process.

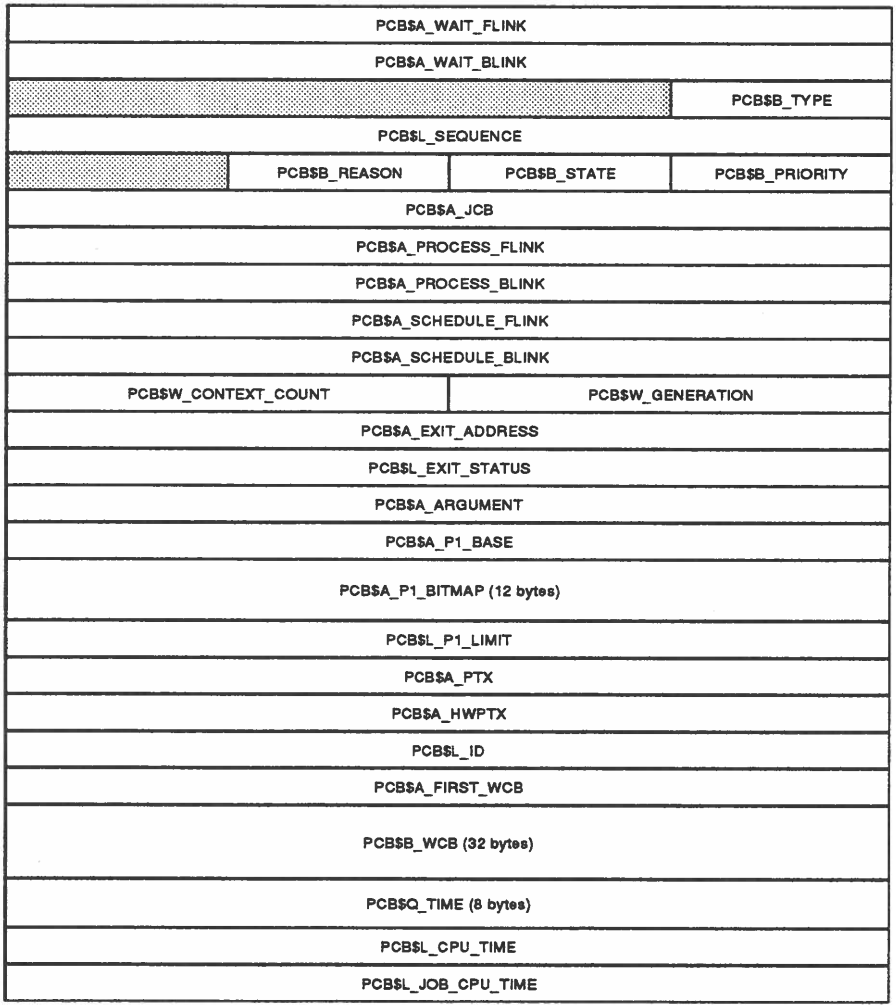
The PCB represents the process-specific portion of a process's software context, as distinct from the jobwide portion represented in the JCB. The PCB ties together the attributes, state, resources, and components that are private to the process. As described in Section 4.5, the PCB is created at offset 128 within a page allocated for the PTX and PCB in the communication region of S0 address space. The information maintained in its fields includes the following:

- Links into the scheduling queue for its process priority and into the job's process queue
- Pointers to a linked list of wait control blocks (WCBs), headed by a timer WCB, and to a block of process arguments
- Pointers to the job's JCB and to the process's PTX (a virtual address)
- Pointer to the process's P1 page table, and the number of page table entries (at the low-address end of the page table) that correspond to inaccessible pages

- Fields for accumulating process statistics

Figure 4–3 shows the structure of the PCB, and Table 4–2 describes its fields.

Figure 4–3: Structure of a Process Control Block



MLO-003223

Table 4–2: Process Control Block Fields

Field	Meaning
PCB$A_WAIT_FLINK PCB$A_WAIT_BLINK	The listhead for the queue of WCBs representing processes waiting for this process's termination. This queue is walked by the KER$DELETE procedure during process deletion to mark each process wait as satisfied and determine whether any waiting processes can be unblocked as a result.
PCB$B_TYPE	The structure type: OBJ$K_PROCESS.
PCB$L_SEQUENCE	The object sequence number for this PCB.
PCB$B_PRIORITY	The process's priority, 0 (highest) to 15 (lowest). Initial process priority is specified on the Program Description Menu.
PCB$B_STATE	The process's state: PCB$K_READY (0), PCB$K_RUNNING (1), PCB$K_SUSPENDED (2), or PCB$K_WAITING (3).
PCB$B_REASON	A bit mask representing asynchronous exceptions pending against the process. The fields are listed and described in Section 6.5.1.4.
PCB$A_JCB	The address of the process's associated JCB.
PCB$A_PROCESS_FLINK PCB$A_PROCESS_BLINK	The forward and backward links to next and previous processes in the job's process queue; the JCB$A_PROCESS_FLINK and JCB$A_PROCESS_BLINK fields in the JCB represent the listhead for the queue.
PCB$A_SCHEDULE_FLINK PCB$A_SCHEDULE_BLINK	The forward and backward links to next and previous processes in the ready queue for this process's priority.
PCB$W_GENERATION	The process generation number, recorded when the job is created; the value is generated in JCB field JCB$W_PROCESS_GENERATION.
PCB$W_CONTEXT_COUNT	The number of times the process switches to the running state; cleared at process creation and subsequently incremented for each switch to the running state.
PCB$A_EXIT_ADDRESS	The user-specified P0 address to which exit status is to be returned on process termination; 0 if no exit address was specified by the creator.
PCB$L_EXIT_STATUS	The exit status value.

Table 4–2 (Cont.): Process Control Block Fields

Field	Meaning
PCB$A_ARGUMENT	The address of the process argument block. If the creator of the process specified arguments, the block contains a longword argument count and a block of contiguous longwords containing the arguments; otherwise the block contains an argument count of 0.
PCB$A_P1_BASE	The system virtual address of the P1 page table. This is the address of the first existent PTE (PTE corresponding to an accessible page) in the P1 page table. The page table grows toward this address from the high end. This value is not the same as PTX$A_P1BR, the value used in P1 address translation, which reflects the base address of the nonexistent portion (PTEs corresponding to inaccessible pages) of the P1 page table.
PCB$A_P1_BITMAP	The descriptor for the P1 virtual page allocation bitmap. This descriptor is used to locate and update the P1 bitmap during the allocation of P1 virtual memory. The format of the bitmap descriptor (BMP) is described in Chapter 9.
PCB$L_P1_LIMIT	The number of nonexistent PTEs (PTEs corresponding to inaccessible pages) in the P1 page table. This value is updated during the expansion of the P1 page table and corresponds to the value of the P1 length register (P1LR).
PCB$A_PTX	The virtual address of the process hardware context block (PTX).
PCB$A_HWPTX	The physical address of the PTX. The physical address is placed in the hardware process control block base register (PCBB) when the process is scheduled to run, prior to the loading of the process's context with the LDPCTX instruction.
PCB$L_ID	The object identifier for this PCB.
PCB$A_FIRST_WCB	The address of the first WCB in the process's current (active) wait list. For a timed wait, this field will contain the address of the timer WCB; for waits without a time value, it contains the address of the first WCB for the first object specified in the wait. The process wait list is initialized by the KER$WAIT procedure and is walked by kernel subroutines that test and satisfy wait conditions.

Table 4–2 (Cont.): Process Control Block Fields

Field	Meaning
PCB$B_WCB	The timer WCB, used to handle timeout values for waits requested by the process; it is also the head of a singly linked list of the process's other allocated WCBs, representing objects (resources or events) the process waits on. The list is walked by the KER$DELETE procedure on process deletion to free the pool blocks occupied by a process's WCBs. When a wait is initiated, a WCB is linked into other lists as well: the timer WCB and its time value are linked into the kernel's timer queue, and the WCB for each object specified in the wait is linked into a queue of processes waiting for that object.
PCB$Q_TIME	A 64-bit binary time value associated with the timer WCB in the PCB$B_WCB field; this value represents the absolute system time at which the wait expires.
PCB$L_CPU_TIME	An accumulator for the process's execution time.
PCB$L_JOB_CPU_TIME	The total accumulated CPU time of all deleted processes in the job (recorded in the master process PCB only).

4.2.3 Process Hardware Context Block

A process hardware context block (PTX) is created by the KER$CREATE_PROCESS kernel procedure for each subprocess created in a VAXELN system. In addition, a PTX is created by the KER$CREATE_JOB kernel procedure for a job's master process.

The PTX stores the hardware context of a process when it is not executing. The first 96 bytes provide the information a VAX processor requires when it loads process context to place a process in the running state or when it saves process context to remove a process from execution. Additional fields (specific to VAXELN and not loaded or saved) provide the username, UIC, and name block address associated with the process.

As described in Section 4.5, the PTX is created at the beginning of a page allocated for both it and the PCB in the communication region of S0 address space. The information maintained in the fields of the PTX includes the following:

- Contents of the general purpose registers, stack pointer registers, and page table registers

- User information for the process: username, UIC, and process name

When a process's context is loaded, the majority of the PTX's fields are moved into processor registers. When a process's context is saved in order for it to be removed from execution, the updated contents of these registers is stored back into the PTX.

Figure 4–4 shows the structure of the PTX, and Table 4–3 describes its fields.

Figure 4–4: Structure of a Process Hardware Context Block

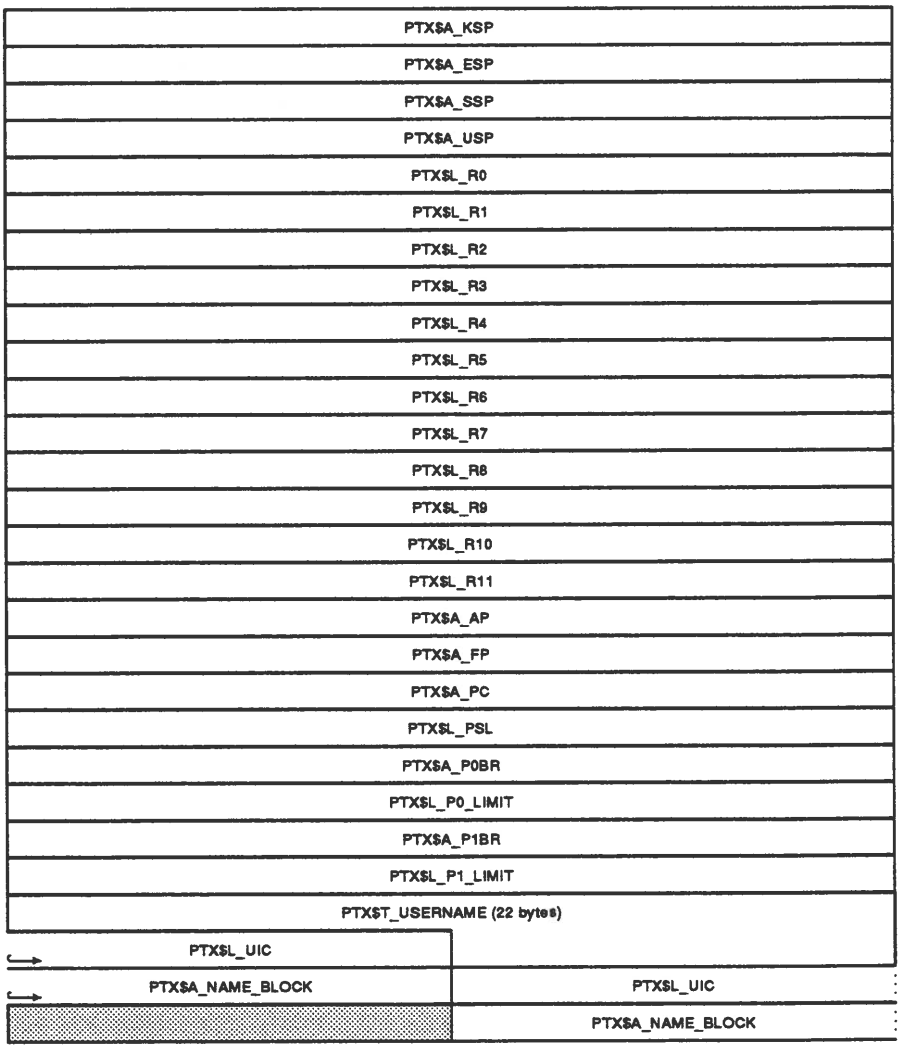


Table 4–3: Process Hardware Context Block Fields

Field	Meaning
PTX$A_KSP	The kernel stack pointer — used when the process is executing in kernel mode
PTX$A_ESP	The executive stack pointer — not used
PTX$A_SSP	The supervisor stack pointer — not used
PTX$A_USP	The user stack pointer — used when the process is executing in user mode
PTX$L_R0	The contents of general register R0
PTX$L_R1	The contents of general register R1
PTX$L_R2	The contents of general register R2
PTX$L_R3	The contents of general register R3
PTX$L_R4	The contents of general register R4
PTX$L_R5	The contents of general register R5
PTX$L_R6	The contents of general register R6
PTX$L_R7	The contents of general register R7
PTX$L_R8	The contents of general register R8
PTX$L_R9	The contents of general register R9
PTX$L_R10	The contents of general register R10
PTX$L_R11	The contents of general register R11
PTX$A_AP	The contents of the argument pointer
PTX$A_FP	The contents of the frame pointer
PTX$A_PC	The contents of the program counter
PTX$L_PSL	The contents of the processor status longword
PTX$A_P0BR	The contents of the P0 base register for this process
PTX$L_P0_LIMIT	A bit field containing the following relevant fields:
	PTX$V_P0LR — Bits <21:0>: the low-order 22 bits of the P0LR for this process
	PTX$V_ASTLVL — Bits <26:24>: the value for the AST level register (ASTLVL); contains the access mode number of the most privileged mode for which an asynchronous exception is pending

Table 4–3 (Cont.): Process Hardware Context Block Fields

Field	Meaning
PTX$A_P1BR	The contents of the P1BR for this process
PTX$L_P1_LIMIT (PTX$V_ P1LR)	A bit field (bits <21:0>) containing the low-order 22 bits of the P1LR for this process
PTX$T_USERNAME	The process's username, stored as a word-length character count followed by a username of up to 20 characters; defaults to username USER if not explicitly set by a KER$SET_USER procedure call
PTX$L_UIC	The process's UIC; if not explicitly set by KER$SET_USER, defaults to the UIC set at build time on the Program Description Menu (kernel value KER$GL_DEFAULT_UIC) or to the System Builder's default, [1,1]
PTX$A_NAME_BLOCK	The address of the process's name block, if a procedure call has been issued to name a process; otherwise 0

4.3 Job and Process Virtual Memory

A job's P0 page table maps job components and resources into the program region of process virtual address space for jobwide access. For each process in the job, a separate P1 page table maps process components and resources into the control region of process virtual address space for private access by the process. Section 4.3.1 describes the job components and resources mapped into P0 memory, and Section 4.3.2 describes the process components and resources mapped into P1 memory.

4.3.1 Job Virtual Address Space

The P0 address space of a process is created during the creation of its job, as described in Section 4.4, and is shared among all processes in the job. The job components and resources mapped into P0 memory are shown in Figure 4–5 and described in detail in Table 4–4.

Figure 4–5: Structure of P0 Virtual Memory

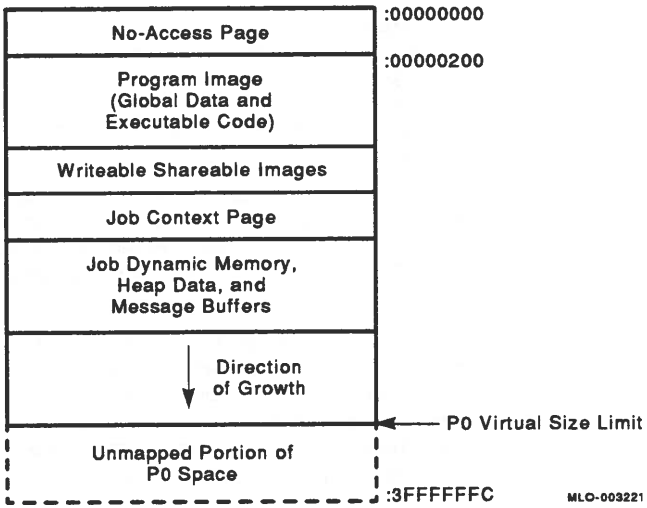


Table 4–4: Job Components Mapped into P0 Address Space

Component	Purpose
No-access page	A single page whose bit is cleared in the P0 virtual memory bitmap to ensure that it is never allocated. The page is made inaccessible because the VMS Linker does not normally allocate virtual addresses below 200_{16}.
Program image	The program's global data and executable code. At job creation, the kernel maps into virtual memory the global data and executable code program image sections created by the VMS Linker and described by kernel section descriptors (KSDs) in the program descriptor. KSDs for program images are described in Section 2.4.1.3.
Writeable shareable images	The program's writeable-shareable image sections, corresponding to the shareable images it references. At job creation, the kernel maps into virtual memory the writeable-shareable program image sections created by the VMS Linker and described by kernel section descriptors (KSDs) in the program descriptor. KSDs for program images are described in Section 2.4.1.3.

Table 4–4 (Cont.): Job Components Mapped into P0 Address Space

Component	Purpose	
Job context page	A page used to record jobwide information for run-time library routines and the debugger. Maintained within this page is a job context block (JCX) containing the P0 address of the job's program arguments and context values used by the heap management routines, the debugger, and the file system. Sample fields include the following:	
	JCX$A_JOB_PARAMETERS	The address of the job's program arguments (copied from JPBs into job dynamic memory during job creation), valid for the life of the job.
	JCX$A_HEAP_LISTHEAD	The listhead for the heap-storage queue, the first of three heap storage fields.
	JCX$L_DEBUG_BPT_MASK	The first of several debugger context fields.
	JCX$A_FILE_LISTHEAD	The first of four open-file context fields, including a listhead for file rundown on job exit.
Job dynamic memory	A region of memory used to map heap data (allocated, for example, by the Pascal routines NEW and DISPOSE), message buffers created by KER$CREATE_MESSAGE, shared memory buffers created by KER$CREATE_AREA, and memory allocated at P0 virtual addresses by KER$ALLOCATE_MEMORY. This dynamic region can expand from the base of job dynamic memory — one page beyond the last page mapped by the program image — to the P0 virtual size limit established by the **P0 virtual size** value on the System Characteristics Menu. Allocating job dynamic memory is discussed in Section 9.3.2.2.	

4.3.2 Process Virtual Address Space

The P1 address space of a process is mapped during its creation, as described in Section 4.5, and is inaccessible to other processes in the job and in the system. The process components mapped into P1 memory are shown in Figure 4–6 and described in detail in Table 4–5.

NOTE

Figure 4–6 and Table 4–5 do not represent or describe dynamic memory allocated at P1 virtual addresses with the KER$ALLOCATE_MEMORY procedure, because such allocations are not typical and require careful attention. For example, an allocation in kernel or user stack space would require a guarantee that the allocated area not be used for normal stack activity while allocated. KER$ALLOCATE_MEMORY typically is used to allocate P0 dynamic memory, as reflected in the corresponding P0 figure and table in Section 4.3.1.

Figure 4–6: Structure of P1 Virtual Memory

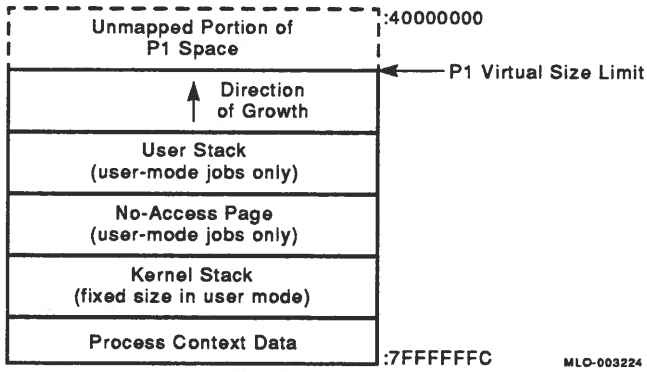


Table 4–5: Process Components Mapped into P1 Address Space

Component	Purpose
User stack	The dynamic memory region used by user-mode processes, when executing in user mode, to store automatic (local) variables and procedure call frames and argument lists (in stack-based languages such as VAX C and VAXELN Pascal). The initial size of the user stack is set by the **User stack** entry on the Program Description Menu and is transmitted to the kernel through the program descriptor value PRG$W_USER_STACK. The same initial stack size is used for every process created in a job. As stack requirements increase at run time, the kernel expands the user stack. A limit for expansion is specified as the **P1 virtual size** value on the System Characteristics Menu. The user stack begins at a –512 byte offset from the end of the kernel stack. This stack exists only for user-mode programs.
No-access page	A guard page marked inaccessible by the kernel to separate the user and kernel stacks. Attempts to access this page inform the kernel that the kernel stack has overrun its limit. This page exists only for user-mode programs.
Kernel stack	The dynamic memory region used by kernel-mode processes to store automatic (local) variables and procedure call frames and argument lists. This stack is also used by user-mode processes executing in kernel mode, as they do when executing most kernel procedures (see Chapter 8). The size of the kernel stack is set by the **Kernel stack** entry on the Program Description Menu and is transmitted to the kernel through the program descriptor value PRG$W_KERNEL_STACK. Overrunning the stack limit causes a fatal kernel stack exception in the process. Kernel-stack overrun can be circumvented through the use of the ELN$ALLOCATE_STACK utility procedure, used by kernel-mode programs to explicitly expand the stack. The kernel stack begins at fixed location P1$K_KERNEL_STACK_INIT — $7FFFFDF0_{16}$ — at a -10_{16} byte offset from the end of the first page allocated for kernel stack. The offset allows for the presence of 4 process context longwords (part of the process context data area) at the end of the page.

Table 4–5 (Cont.): Process Components Mapped into P1 Address Space

Component	Purpose
Process context data	Process context longwords and process debugging information. The process context longwords occupy the high-order 16 bytes of the first page allocated for the kernel stack, immediately following the kernel stack base (P1$K_KERNEL_STACK_INIT) in P1 memory. If debugging was requested, the process debugging information is maintained at the beginning of the next virtual page, which is the last page of P1 memory. The process context longwords provide fixed locations for storing context addresses, as follows: P1$GA_JCX — The jobwide context address; the job context data area is shown in Figure 4–5 and summarized in Table 4–4. P1$GA_ADA_CTX — Ada run-time context address. P1$GA_ADA_DATA — Ada data run-time context address. P1$GA_CRTL_CTX — C run-time library context address. If debugging was requested for a program, process-specific information for use by the debugger is maintained in a process context block (PCX) at location P1$GR_CONTEXT — $7FFFFE00_{16}$ — the beginning of the last page of P1 memory. The PCX is initialized by the debugger bootstrap at the conclusion of job or process creation.

4.4 Job Creation

The goal of job creation is to establish an environment for the execution of a program. At system start-up, when the first jobs are created in a system, the kernel's initialization sequence has already prepared the system for the creation and execution of jobs and processes. When the KER$CREATE_JOB kernel procedure (in module CREATEJOB) is invoked, the kernel creates a job and a master process to execute a program's main code thread, beginning at its linker-specified transfer address. Jobwide and process-specific contexts and address spaces are established and a scheduling pass is made. When the job is scheduled to run, execution of the program's main code by the master process will begin.

This section focuses on the job creation sequence, which proceeds through three phases:

1. The initial KER$CREATE_JOB procedure call is handled. This phase creates the minimal job and process context that will allow the job's master process to continue creation of the job. Thus the creating job can be decoupled from the created job and continue executing its own code.

   At the conclusion of this phase, a scheduling pass is made, which can result in the job and master process being selected to run immediately or being placed in the ready job and process queues for their priorities. Saved in the PC field of the master process's PTX is the address of its continuation point within the CREATEJOB module, the FINISH_JOB subroutine. The initial procedure call exits by executing an REI instruction, returning status to its caller. Section 4.4.1 describes this phase.

2. When the system state permits, the master process of the new job is placed in the running state by the scheduler and exercises the FINISH_JOB subroutine, which completes the creation of the job environment.

   This phase concludes when the kernel executes an REI instruction to transfer control to another CREATEJOB module subroutine, KER$ENTER_PROCESS, in the base program mode, user or kernel, specified in the program's descriptor. Section 4.4.2 describes this phase.

3. Job creation is completed in the user-specified program mode. This phase sets up entry to the program's transfer address. (If debugging was requested and the debugger is present, control is first transferred to the debugger bootstrap.) Section 4.4.3 describes this phase.

4.4.1 Phase 1: Creating Minimal Job and Master Process Context

Job creation is initiated by a call to the KER$CREATE_JOB kernel procedure. The KER$CREATE_JOB procedure can be called implicitly by system software — such as system initialization and start-up modules, the debugger, ECL, and LAT — or explicitly by application software.

The main objective of the initial phase of job creation is to construct the minimal job and process context necessary to allow job creation to continue under the control of the job's master process. By returning from the initial KER$CREATE_JOB procedure call as quickly as possible, the kernel decouples the creating job from the created job so that each job can continue executing code in the normal scheduling environment. When the job's master process has been provided with enough context to be scheduled, an REI instruction is executed to exit the system call and return control to the caller.

The kernel creates a minimal job and master process context as follows:

1. KER$CREATE_JOB call arguments are verified.
2. The JCB is created and initialized.
3. Structures are created for managing the job's objects.
4. JCB fields are initialized for P0 memory mapping.
5. The master process is created and initialized: PCB and PTX structures are created for it, its PCB is entered into the job's first object pointer table, its P0 and P1 page tables are allocated, and a page of kernel stack is allocated for it.
6. The job's message port is created.
7. For KA620-based systems only, the entire P0 page table is allocated.
8. A scheduling pass is made, which can result in the job and master process being selected to run or being placed in ready job and process queues.

4.4.1.1 Step 1 — Verify Call Arguments

The KER$CREATE_JOB procedure begins by verifying the arguments passed to it on the stack. The checks made are as follows:

- Job message port address. The location to receive the job port identifier must be writeable; if not, the procedure exits with KER$_NO_ACCESS status.
- Program name. The program name must be readable. If it is, then the system's program descriptor table is scanned for a program list entry matching the caller-specified program name. If no match is found, the procedure exits with KER$_NO_SUCH_PROGRAM status. If the program is found, and it's a dynamically loaded program, the program reference count, variable PRG$W_REF_COUNT in the program descriptor, is incremented. The kernel

maintains this variable for each dynamically loaded program to record how many currently active jobs are running the program, to prevent premature unloading of the program.

- Exit port address. The port identifier argument, if specified, must be readable; if not, the procedure exits with a KER$_NO_ACCESS status.
- Job parameter strings. Each string argument specified must be readable and of the correct size.

4.4.1.2 Step 2 — Create the Job Control Block

After verifying the arguments passed to it, the KER$CREATE_JOB procedure allocates the pool blocks (3, plus 1 for each job argument) and the communication region pages (2) it will need to create job data structures.

The KER$ALLOCATE_POOL subroutine, described in Section 9.4, is used for the pool allocation. Each pool block allocated by KER$ALLOCATE_POOL must subsequently be removed from the pool before it can be used. Pool blocks are allocated for the following job and master-process control structures:

- Ready-process queue header block, containing listheads for the job's 16 priority-ordered ready process queues
- Initial allocation of an object pointer table, to hold pointers to the first 32 objects created dynamically within the job — beginning with the process object for the master process
- Initial allocation of four WCBs to be used when the master process issues a KER$WAIT call
- A pool block for each job argument; a JPB is created in each pool block

The KER$ALLOCATE_REGION subroutine, described in Section 9.3.1.1, is used for memory allocation from the communication region. The first page of the allocated memory holds the job's JCB; the second page holds the master process's PTX and PCB.

The kernel clears 128 bytes for the JCB at the beginning of the 2 pages from the communication region. The kernel proceeds to create other structures to be linked into or otherwise referenced from the JCB.

The kernel copies any caller-specified program arguments (as verified in step 1) into a linked list of JPBs. (If the job is created at system start-up, the supplied arguments are from the Program Description Menu.) For each job argument, a pool block is removed from the allocated pool and a JPB created in the pool block. Each JPB is linked into a singly-linked list whose head is the JCB$A_PARAMETER_LIST field in the JCB. The fields in the parameter block are described in Table 2–4.

When the job's P0 address space has been established, later in job creation, the program arguments are copied there in a format that allows faster jobwide access to them. The argument strings themselves are preceded in P0 memory by a standard VAX argument list containing the argument count and the addresses of each parameter string descriptor. At that point, the JPB pool blocks are returned to the system.

The kernel proceeds to initialize other portions of the JCB as follows:

- The structure type field, JCB$B_TYPE, is set to value OBJ$K_JOB, indicating a JCB structure.
- The job priority field, JCB$B_PRIORITY, is set to the value contained in the associated program description (field PRG$B_JOB_PRIORITY).
- The JCB field containing the priority of the highest priority ready process in the job, JCB$B_READY_PRIORITY, is set to the initial process priority of the master process, which will be the job's only process when the job is first eligible for scheduling. The value is copied from the process priority field (PRG$B_PROCESS_PRIORITY) of the PRG.
- The job state field, JCB$B_STATE, is set to value JCB$K_READY, indicating a ready state.
- The current process field, JCB$A_CURRENT_PCB, is set to the address of the master process PCB. The current process is the process placed in the running state for this job; the designated process is actually running only if its job is running.
- The listhead for the job's process queue, fields JCB$A_PROCESS_FLINK and JCB$A_PROCESS_BLINK, is initialized to an empty queue.
- The listhead for the job's port queue, fields JCB$A_PORT_FLINK and JCB$A_PORT_BLINK, is initialized to an empty queue.

- The job's process-switching disable count, JCB$W_DISABLE, is initialized to 0, indicating process switching is enabled for the job. A disable request received at run time increases this count by 1, and an enable request decreases the count by 1. A count greater than 0 indicates process switching is disabled.
- The job's context switch count, JCB$W_CONTEXT_COUNT — the number of times the job has entered the running state — is cleared.
- The job's process-scheduling ready summary mask, JCB$W_READY_SUMMARY, is cleared, indicating all ready-process queues in the priority-ordered set (pointed to by JCB$A_PROCESS_QUEUES) are initially empty.
- The job's ready-process queue listheads are created at the beginning of a pool block from the earlier pool allocation. A set of 16 quadword listheads, one for each process priority level (0–15, in order), is initialized. The JCB$A_PROCESS_QUEUES field is set to the address of the pool block; this address is indexed by priority value when the kernel needs to locate a particular priority ready-process queue.
- The job's processor eligibility mask, JCB$W_CPU_MASK, is initialized with the eligibility mask from the PRG. In a tightly coupled symmetric multiprocessing system, each bit set in the eligibility mask indicates eligibility for a processor; bit *n* indicates eligibility for processor *n*. Under the current design, the mask copied from the program description is always 0, indicating the job initially is eligible to run on any processor; however, a device driver on a VAX 8800-series multiprocessing system is by default made eligible only for the processor that handles the device's interrupts, when it issues a KER$CREATE_DEVICE call. This mask can be altered with the KER$SET_JOB_ELIGIBILITY procedure. In single processor systems, altering this mask has no effect.
- The job's generation number, JCB$W_GENERATION, is set; the kernel's count of generated jobs, KER$GW_JOB_GENERATION, is increased by one and copied into the JCB field.
- The job's seed for generating process generation numbers, JCB$W_PROCESS_GENERATION, is initialized to 0. For each process subsequently created in the job, the JCB count is increased by 1 and copied into the PCB field PCB$W_GENERATION.
- If the job creator specified an exit port for the job — a port to receive the master process's completion status when the job terminates — the exit port's identifier is placed in JCB field JCB$B_EXIT_PORT_ID.

- The program's access mode, kernel or user, in which the job's processes are to begin their execution, is copied from the PRG into the base access mode field of the JCB, JCB$B_MODE.

4.4.1.3 Step 3 — Create Object Management Structures

During job creation, the kernel creates data structures for managing the objects created dynamically for or by the job:

- The object base table, which contains the addresses of dynamically-created object pointer tables. This table forms the first tier in a two-tiered arrangement of address tables and can hold the addresses of up to 128 object pointer tables. The base table occupies one 512-byte page in the communication region.

  The KER$CREATE_JOB procedure creates a single base table (with the internal routine KER$ALLOCATE_REGION in module ALLOCATE) and stores its address in the JCB$A_OBJECT_TABLE field of the JCB. This field is used when looking up a job object to locate the job's object base table. The table exists until the job is deleted.

- The object pointer tables, which contain the addresses of dynamically created kernel objects. These tables form the second tier of address tables. A pointer table occupies one 128-byte pool block and can contain the addresses of up to 32 kernel objects. (Since there can be up to 128 pointer tables and each can point to up to 32 objects, a job can have up to 4096 objects.)

  The KER$CREATE_JOB procedure creates the job's initial object pointer table (with the internal routine KER$ALLOCATE_OBJECT in module ALLOCATE) and places its address in the job's object base table; subsequent tables will be allocated as each pointer table fills up. (Once a pointer table is allocated, it exists until the job is deleted, even though many of the objects it points to may no longer exist.) KER$CREATE_JOB places in the JCB$W_OBJECT_FREE field of the JCB an encoded value representing the location of the next available entry in the object pointer tables. This field is used and updated when creating or deleting job objects.

The first object represented in a job's object tables is the PCB of its master process.

Chapter 10 describes the object base table and the object pointer tables in more detail.

4.4.1.4 Step 4 — Initialize JCB Fields for P0 Memory Management

To prepare for the mapping of the job's executable program code and global data into P0 address space in the second phase of job creation, the KER$CREATE_JOB procedure initializes three fields in the JCB:

- The JCB$A_PROGRAM field receives the address of the job's program descriptor, used to look up program characteristics.
- The JCB$L_MESSAGE_PTE field receives a prototype page table entry (PTE) for creating and filling in PTEs for the job's message and area buffers. The kernel creates a PTE with the valid, protection (PTE$C_UW), owner (program access mode), and type (PTE$K_MESSAGE) fields present. During the allocation of message and area buffers, the allocated page frame number is inserted into the PFN field to create the actual PTE for the page.
- The JCB$L_RW_DATA_PTE field receives a prototype page table entry (PTE) for creation of PTEs for the job's read/write data. The kernel creates a PTE with the valid, protection (PTE$C_UW), owner (program access mode), and type (PTE$K_RW_DATA) fields present. During P0 memory allocation (global data, heap, and KER$ALLOCATE_MEMORY) and P1 user-mode stack allocation, the allocated page frame number is inserted into the PFN field to create the actual PTE for the page.

4.4.1.5 Step 5 — Create the Master Process

A major part of job creation is establishing a master process to execute the program's main code thread. However, in the first phase of job creation, the more immediate goal is to create a master process with enough context to assume control of job creation in the second phase.

For the most part, the actions taken by the kernel to create a master process in phase 1 mimic the corresponding phase of subprocess creation, as described in Section 4.5.1. The kernel proceeds as follows; detail is provided only for steps that differ significantly from subprocess creation:

1. The master process's PCB is created and initialized. This step matches step 2 of subprocess creation (Section 4.5.1.2), with one difference: the master process state, field PCB$B_STATE, is initialized to indicate a running state (PCB$K_RUNNING). This marks the master process to run when the job is selected to run, since the master process is initially the only ready process in the job.

2. The PTX for the master process, representing its hardware context, is initialized. This step matches step 3 of subprocess creation (Section 4.5.1.3), except as follows:
   - The name block field, PTX$A_NAME_BLOCK, of the PTX is cleared; a master process is given no name. Unlike a subprocess, which can be named and displayed with the given name by VAXELN utilities, a master process generally remains unnamed. For display purposes, VAXELN utilities use the job's program name for the master process; any name established for a master process by a run-time call to the KER$CREATE_NAME or KER$NAME_OBJECT procedure is disregarded.
   - The PC and PSL fields of the PTX are prepared for transfer to the next phase of process creation. The PTX$A_PC field receives the address of the FINISH_JOB internal subroutine. When the new process is first scheduled to run, it will reenter the CREATEJOB module at FINISH_JOB and execute the second phase of job creation. The PTX$L_PSL field is cleared to initialize access mode to kernel (0) and IPL to 0 for when the process is first scheduled to run.
   - The P0LR processor register subfield of PTX$L_P0_LIMIT is cleared, since the master process initially executes a CREATEJOB module routine, FINISH_JOB, out of S0 address space. The master process's P0LR value is set up for program code execution later, as the program's P0 code and global data are mapped or allocated and mapped. (The P0BR field is initialized in step 4 of master-process creation.)

     The ASTLVL subfield of PTX$L_P0_LIMIT receives the value PSL$C_USER + 1 (4), indicating no asynchronous exception is pending for the master process.
3. The PCB for the master process, as the kernel object representing the master process, is entered into the job's object tables. An object pointer table entry is allocated and receives the address of the PCB. This step matches step 6 of subprocess creation (Section 4.5.1.6). The master process PCB is the first object entered in the job object tables.
4. The job's P0 page table is allocated. A P0 page table slot is allocated with a call to the KER$ALLOCATE_P0_SLOT subroutine, following which the page table's base address (equal to the slot's base address) is placed in the PTX (PTX$A_P0BR) and the JCB (JCB$A_P0_BASE). (The P0LR field of the PTX was initialized in an earlier step of master process creation.)

Also, both address subfields of the P0 allocation bitmap descriptor in the JCB (JCB$A_P0_BITMAP) are initialized with the address (within the slot) of the P0 allocation bitmap. Length fields in the descriptor remain clear. This zero-length bitmap will cause the allocation of a page of PTEs for the first 128 pages of virtual memory for the job when the kernel first attempts to allocate P0 memory to map the job's image sections.

Chapter 9 describes P0 page tables and their handling in detail.

5. The master process's P1 page table is allocated. This step matches step 4 of subprocess creation (Section 4.5.1.4). Chapter 9 describes P1 page tables and their handling in detail.
6. The master process's username and UIC are written into fields PTX$T_USERNAME and PTX$L_UIC of the PTX. The username and UIC are inherited from the creating process, unless the job being created is the system start-up job. In the case of the start-up job, the master process is given the username USER and the default UIC specified on the Program Description Menu.
7. One page of kernel stack is allocated for the master process, part of the minimal context the master process needs to assume control of the second phase of job creation. This step corresponds to step 5 of subprocess creation (Section 4.5.1.5). Remaining process stack allocation is done in the second phase.

4.4.1.6 Step 6 — Create the Job's Job Port

After creating the master process, the KER$CREATE_JOB procedure creates the job port with a call to the KER$CREATE_PORT kernel procedure. The message limit set for the job port is the job port message limit specified on the Program Description Menu. Because this phase of job creation is executed by the caller of KER$CREATE_JOB rather than the newly created job, the port object allocated by KER$CREATE_PORT resides in the creating job's port queue; so the kernel rewrites the port object's owner field (PRT$A_OWNER) with a pointer to the new job's JCB, removes the object from the creating job's port queue, and inserts it into the created job's port queue. The port's local identifier is stored in the JCB$L_PORT_ID field of the JCB.

4.4.1.7 Step 7 — Allocate the P0 Page Table for KA620-Based Systems

If the new job is being created for a KA620-based system, such as the rtVAX 1000, page table entries for the entire P0 page table are allocated. P0 and P1 page tables on the KA620 processor are physically contiguous and referenced with physical addresses, which saves virtual address translations for process-space memory references. The process page tables on a KA620 processor are not expanded dynamically in one-page increments up to the specified limit for the process; rather, they are completely allocated during job (P0) and process (P1) creation. A mock P0PTE allocation and deallocation is performed to force immediate allocation of the entire P0 page table, and the P0BR field of the PTX is updated accordingly.

4.4.1.8 Step 8 — Initiate a Scheduling Pass

At the conclusion of the first phase of job creation, the job's JCB (at field JCB$A_JOB_FLINK) is linked into the queue of all jobs within the system (accessed through global listhead KER$GQ_SYSTEM_JOB). Then the kernel calls the KER$READY_JOB subroutine, which initiates a scheduling pass. The job and its master process might be selected to run immediately — for example, if the created job has a higher combined job and process priority than the creating job in a single-processor system — or might be placed in the ready-job and ready-process queues for their priorities to await a change in system state that allows the job to run.

Finally, the KER$CREATE_JOB procedure returns to its caller with an REI instruction. When the newly created job is selected to execute by the scheduler and begins running, it will run in kernel mode and execute the FINISH_JOB subroutine to finish creating the job environment.

4.4.2 Phase 2: Finishing Creation of the Job Environment

The main objective of the second phase of job creation is to complete construction of the job environment. At the end of the second phase, a REI instruction is executed that transfers control to the user-specified program mode (kernel or user) and to a subroutine in the CREATEJOB module that sets up entry to the program's transfer address.

The kernel completes creation of the job environment as follows:

1. The process stacks are allocated.

2. The job's image sections are mapped to P0 memory.
3. Program arguments are stored in P0 memory for jobwide access.
4. Transfer to program mode and to the KER$ENTER_PROCESS subroutine is set up.

4.4.2.1 Step 1 — Allocate the Process Stacks

To finish allocating the master process's stacks, the kernel calls the CREATEJOB subroutine KER$ALLOCATE_PROCESS_STACK, which is called for both master process and subprocess creation. Using the stack sizes specified in the program description for the job's program, KER$ALLOCATE_PROCESS_STACK allocates the remainder of the kernel stack (beyond the page already allocated) and, if the program mode is user, the user stack. If the user stack is allocated, an extra guard page is also allocated between the user stack and the kernel stack, to mark the end of the fixed-length kernel stack. The KER$ALLOCATE_PROCESS_STACK subroutine is described in detail in Section 4.5.2.1, which covers the corresponding step of subprocess creation.

4.4.2.2 Step 2 — Map the Job's Image Sections

In the second phase of job creation, the kernel maps into job (P0) address space the job's image sections, as specified in lists linked to the job's program descriptor. The job's image sections contain the executable code and global data of the program and of object library routines it references, as well as the code and data of shareable images referenced by the program.

Each image section is described by a kernel section descriptor (KSD), which resides in a list of KSDs pointed to by the program descriptor. The KSD describes the image section's characteristics and virtual memory requirements; diagrams of KSDs are provided in Chapter 2.

The program's KSD list includes private KSDs, which describe the program's executable code and global data image sections, and global KSDs, which describe a virtual address range to which the image sections of a shareable image are to be mapped. Each global KSD points to a sublist of shareable KSDs, describing the code and data image sections of a writeable shareable image referenced by the program.

The job's program descriptor, list of KSDs, and image sections all reside in the system image (described in Chapter 2), which was created by the System Builder and mapped into S0 address space during system initialization. In this step of job creation, some of the program's image sections, such as read/write data sections, are copied from S0 space into newly allocated P0 memory pages and mapped, while others, such as read-only code or data sections, are mapped without creating a P0 copy.

To map the program's image sections, the kernel walks the program's KSD list and maps each section as it goes; when it encounters a global KSD (a KSD with the type value KSD$K_GBL in the KSD$B_TYPE field), it completely processes the sublist of shareable image KSDs pointed to by the global KSD before returning to the main list. The end of the main list or a shareable KSD sublist is marked by a KSD with a size field of 0. Image mapping is complete when the kernel reaches the end of the program's KSD list.

For each KSD encountered in the KSD list or a shareable image sublist, the kernel calls the internal subroutine MAP_SECTION (in the module CREATEJOB) to map the section. The MAP_SECTION subroutine checks the KSD's fields to determine the image section type, and then maps the image section accordingly:

1. If the KSD indicates no pages are to be mapped (KSD$L_PAGCNT field equals 0), the MAP_SECTION routine exits with success status.

   Otherwise, P0 page table entries are allocated to map a correct number of pages (specified in the KSD's page count field) starting at the image section's user virtual address (calculated during system building and represented in the KSD). This is accomplished by calling the internal routine KER$ALLOCATE_P0_PTE in module ALLOCATE. When the call returns, the kernel updates the processor P0 limit register, P0LR, based on the JCB$L_P0_LIMIT field of the created job's JCB.

2. If the KSD describes a read/write data section (the KSD type is KSD$K_DATA and the KSD flag KSD$V_CRF is set), MAP_SECTION allocates the required number of P0 page frames with the internal routine KER$ALLOCATE_FRAME and fills their PFNs into P0 page table entries; other fields are inserted into each PTE (from the prototype data PTE in JCB field JCB$L_RW_DATA_PTE) that set the page characteristics. Each allocated P0 page receives a copy of a page of the program's global (static) data from where it resides in system space.

3. If the KSD describes a demand-zero image section (the KSD type is KSD$K_DZRO and the KSD flag KSD$V_CRF is set), the mapping algorithm is the same as for read/write data sections. However, each allocated P0 page is zeroed.
4. If the KSD describes a read-only code or data section (the KSD type is KSD$K_CODE and no KSD flags are set), MAP_SECTION double maps the image section from S0 memory to P0 memory, by copying the system page table entries for the image section into the job's P0 page table. No new physical memory is allocated; multiple jobs can double-map and execute the system copy of the program code. The page characteristics are set to indicate user code.
5. If the KSD describes a global common section, such as a FORTRAN common, MAP_SECTION double maps the image section from S0 memory to P0 memory, again by copying the system page table entries for the image section into the job's P0 page table. No new physical memory is allocated. However, the type and access fields of the PTEs are set to indicate user read/write data, and each page is marked as a system page to prevent its deletion upon master-process exit.

   However, if the system copy of the global common image section is in ROM (not writeable), it must be recreated in RAM. The kernel therefore allocates an identical global common section in P0 space, adjusts each of the global-common section's PTEs in the SPT to point to the P0 copy, and enters (double maps) each page also in the P0PT, marked as a system page to prevent its deletion upon master-process exit. All data is copied from the S0 pages to the P0 pages. (The copying is done the first time a job is created that references the global common section; subsequent jobs that reference the global common section simply map it.)

After mapping each image section, the MAP_SECTION subroutine returns with success status and an updated value for the next P0 virtual address to be mapped.

After all the program's image sections have been mapped in P0 address space, the kernel clears the bitmap bit for page 0 of P0 space, to ensure it cannot be allocated by subsequent allocation requests. If page 0 was already allocated for the program's code or data, the bit was already clear and remains clear.

4.4.2.3 Step 3 — Store the Job's Program Arguments for Jobwide Access

In the second phase of job creation, the job program arguments, if any, that were stored in the job parameter list in the first phase of job creation are copied to the job's P0 address space, so that they can be accessed quickly by all processes in the job and so that the pool blocks occupied by the job parameter list can be freed.

The kernel calculates the number of bytes required to hold the argument count (4 bytes), the argument descriptors (12 bytes per descriptor), and the job arguments (as indicated by field JPB$L_TOTAL_SIZE in the job parameter listhead). The kernel then allocates the required number of bytes by calling the KER$ALLOCATE_MEMORY routine. The allocated memory's starting address is returned to location JCX$A_JOB_PARAMETERS in the job context page; this location retains the address of the job's program arguments for the life of the job.

Using the returned pointer to the allocated memory, the kernel first creates a standard VAX argument list for the job. The argument list contains the argument count and the address of the argument descriptor for each argument in the list of JPBs. Following this argument block in memory come the actual argument descriptors, in the order their addresses appear in the argument list. The components of the descriptors are the argument string size and the string itself. As the contents of each JPB are copied into P0 memory, the pool block occupied by the JPB is returned to the system pool.

4.4.2.4 Step 4 — Begin Program Execution

The final step in the second phase of job creation is to transfer control to the user-specified program mode and to the KER$ENTER_PROCESS subroutine, which will set up entry to the program's main code at its transfer address, indicated by the PRG$L_TRANSFER field in the program descriptor.

Prior to the execution of the REI instruction that triggers the possible change of access mode, general registers are set up with addresses of the program's arguments, the master process PCB, the JCB, the job context page, the job's program descriptor, and the program's entry point.

Next the kernel sets up for the REI instruction mechanism. The user-specified program mode, from the PRG$B_MODE field of the program descriptor, is placed on the stack. And, if the specified mode is user, the program PSL on the stack is initialized to specify user access mode. Finally, the address of the KER$ENTER_PROCESS subroutine is

pushed on the stack as the address to which control is to be transferred. The kernel then executes an REI instruction, which pops these PSL and PC values from the stack into their respective registers. The process is now executing in its base access mode at location KER$ENTER_PROCESS.

4.4.3 Phase 3: Entering the Program Code

The main objective of the final phase of job creation is to set up entry to the program at its transfer address so that program execution can begin.

When the master process continues executing following the REI instruction executed in the second phase, the KER$ENTER_PROCESS subroutine executes. This routine, which executes at the close of both job and process creation, causes the process, in this case the master process, to begin executing its actual code (when the process next runs).

If debugging was requested and the debugger is present in the system, the debugger bootstrap, subroutine BOOTSTRAP_PROCESS (in module [DEBUG]LCLNUC), is entered before the new job's master process executes program code. The debugger bootstrap calls a debugger subroutine, INIT_PROCESS_CONTEXT, to establish the process-specific debug context for the process in the last page of P1 address space; see Figure 4–6. INIT_PROCESS_CONTEXT also initializes debugger-related portions of the job context block (JCX) in P0 memory, and sends a message to VAXELN$DEBUG_PORT to announce the presence of the master process. Furthermore, unless the user requested that this job's processes start without debugger intervention, the debugger bootstrap calls the debugger's first-chance exception handler as if a KER$_DEBUG_SIGNAL exception occurred, which causes the process to await a debugger command. If a debugger command starts execution (for example, the GO command), control returns to the debugger bootstrap, which then executes an REI instruction to start program execution.

Similarly, if performance collector PC coverage activity was requested, the performance collector utility is entered before the master process executes program code.

The KER$ENTER_PROCESS subroutine checks for build-time selection of the debugger or performance collector to run in conjunction with this job. A CALLG instruction is executed to transfer control to the debugger, to the performance collector utility, or directly to the program entry point.

While executing, if the master process reaches the end of its code without issuing a KER$EXIT procedure call (or a KER$DELETE for itself), the RET instruction generated by the compiler is executed. When this happens, control returns to the KER$ENTER_PROCESS subroutine, at the instruction after the CALLG used to invoke the process code. KER$ENTER_PROCESS proceeds to initiate the orderly termination of the master process, as follows:

1. The process exit status and the user-supplied status value address are pushed onto the stack.
2. The KER$EXIT kernel procedure is called. The call does not return; the KER$EXIT kernel procedure completes with a call to the KER$DELETE kernel procedure, which deletes the master process, the job, and all the job's subprocesses. After job deletion, the kernel branches to the internal routine KER$SCHEDULE_JOB to reschedule the system.

The KER$EXIT and KER$DELETE kernel procedures are described in Section 4.6.

4.5 Process Creation

The goal of process creation is to create a context for a single thread of execution of a VAXELN program — either a master process to execute the program's main code or a subprocess to execute a routine or process block in the program. When the KER$CREATE_PROCESS kernel procedure (in module CREATEPRO) is invoked, the kernel creates a process with a unique context, represented in its PCB and PTX, and a private P1 virtual address space, mapped by its P1 page table. Additionally, a process implicitly shares in the environment that has been created for it by system initialization and by the creation of its job, including the S0 address space common to every process in the system, the P0 address space (containing the program code) common to every process in the job, and the JCB into which it and all the job's other processes are linked. Once the process is created, a scheduling pass is made. When the process is scheduled to run, execution of program code by the new process begins.

This section focuses on the creation of subprocesses, as initiated by calls to the KER$CREATE_PROCESS kernel procedure. For a description of the creation of a master process, see Section 4.4. Subprocess creation, like job creation, proceeds through three phases:

1. The initial KER$CREATE_PROCESS procedure call is handled. This phase creates the minimal process context that will allow the new subprocess to continue its own creation. Thus the creating process can be decoupled from the created subprocess and continue executing its own code.

   At the conclusion of this phase, a scheduling pass is made, which can result in the process being selected to run immediately or being placed in the ready process queue for its priority. Saved in the PC field of the new process's PTX is the address of its continuation point within the CREATEPRO module, the FINISH_PROCESS subroutine. The initial procedure call exits by executing an REI instruction, returning status to the caller. Section 4.5.1 describes this phase.

2. When the system state permits, the new subprocess is placed in the running state by the scheduler and executes a CREATEPRO module subroutine, FINISH_PROCESS, that finishes creating the process environment.

   This phase concludes when the kernel executes an REI instruction to transfer control to a CREATEJOB module subroutine, KER$ENTER_PROCESS, and to the base program mode, user or kernel, specified in the program's descriptor. Section 4.5.2 describes this phase.

3. Process creation is completed in the user-specified program mode. This phase sets up entry to the routine or process block transfer address. (If debugging was requested and the debugger is present, control is first transferred to the debugger bootstrap.) Section 4.5.3 describes this phase.

Complete descriptions of the process-specific data structures established by process creation are provided in Section 4.2. The same section describes the jobwide data structures established by job creation. Chapter 3 describes the S0 region, implicitly shared by all VAXELN processes.

4.5.1 Phase 1: Creating Minimal Process Context

Subprocess creation is initiated by a call to the KER$CREATE_PROCESS kernel procedure. The KER$CREATE_PROCESS procedure can be called implicitly, such as when a CREATE PROCESS command is issued from the debugger, or explicitly by application software.

The main objective of the initial phase of subprocess creation is to construct the minimal process context necessary to allow subprocess creation to continue under the control of the subprocess. By returning from the initial KER$CREATE_PROCESS procedure call as quickly as possible, the kernel decouples the creating process from the created process so that each process can continue executing in the normal scheduling environment. When the subprocess has been provided with enough context to be scheduled, an REI instruction is executed to exit the system call and return control to the caller.

The kernel creates a minimal process context for the subprocess as follows:

1. KER$CREATE_PROCESS call arguments are verified.
2. The PCB, representing software context, is created and initialized.
3. The PTX, representing hardware context, is created and initialized.
4. The process's P1 page table is allocated.
5. The first page of kernel stack is allocated.
6. The PCB representing the subprocess is entered in the job's object tables.
7. A scheduling pass is made, which can result in the new process being selected to run or being placed in the ready-process queue for its priority.

4.5.1.1 Step 1 — Verify Call Arguments

The first phase of process creation begins with KER$CREATE_PROCESS argument checks. The kernel checks the process-argument list and exit-status address passed to KER$CREATE_PROCESS as follows:

- If the count of process arguments included in the KER$CREATE_PROCESS argument list exceeds 31, the kernel procedure exits with a KER$_BAD_COUNT status.
- The location designated by the caller to receive status when the created process exits is examined. If an address was specified that is not in the P0 region, the kernel procedure exits with KER$_BAD_VALUE status. P0 address space is shared among the job's processes (and survives subprocess deletion).

4.5.1.2 Step 2 — Create the Process Control Block

After verifying call arguments, the kernel prepares to build process structures by allocating an initial set of resources out of S0 address space. First the kernel allocates two pool blocks that will be used to link process arguments and wait control blocks (WCBs) into the PCB. A call is made to the internal kernel subroutine KER$ALLOCATE_POOL, described in Chapter 9; the kernel procedure exits with KER$_NO_POOL status if insufficient pool space is available.

Next, the kernel allocates a page from the communication region of S0 address space that will hold the PTX and the PCB. A call is made to the kernel routine KER$ALLOCATE_REGION, described in Chapter 9; the kernel procedure exits with KER$_NO_MEMORY status if insufficient memory is available.

The kernel then begins building the PTX and PCB, representing the hardware and software context of the new process, and other structures that link into the PTX and PCB. The PTX begins at the beginning (offset 0) of the system page shared by the PTX and PCB; the PCB begins at offset 128 from the beginning of the same page.

The kernel begins creating the PCB as follows:

1. The PCB is cleared.
2. The listhead for the process's wait queue — the queue of WCBs representing processes waiting for this process's termination — is initialized to an empty queue. The link fields are PCB$A_WAIT_FLINK and PCB$A_WAIT_BLINK.
3. The structure type field, PCB$B_TYPE, is set to the value OBJ$K_PROCESS, indicating that this kernel structure represents a process.
4. The process priority field, PCB$B_PRIORITY, is set to the initial process priority, 0 to 15, that was specified for this program in the Program Description menu and stored in the PRG$B_PROCESS_PRIORITY field of the program descriptor. The created process begins executing at this priority; a process's priority can be altered with KER$SET_PROCESS_PRIORITY.
5. The PCB$A_JCB field is set to point to the JCB of the creating job.
6. The process's PCB is inserted at the end of the job's queue of all its processes, linked through the PCB$A_PROCESS_FLINK and PCB$A_PROCESS_BLINK fields of the PCB. The queue listhead resides in fields JCB$A_PROCESS_FLINK and JCB$A_PROCESS_BLINK of the JCB.

7. The process generation number in the JCB (field JCB$W_PROCESS_GENERATION) is increased by 1, and the resulting value is recorded in the PCB's generation number field, PCB$W_GENERATION. A value of *n* indicates that this process is the *n*th created in this job.
8. The process's context switch count in field PCB$W_CONTEXT_COUNT is cleared; this records how many times the process has been switched into the running state by the scheduler.
9. The caller-specified exit address is copied to the PCB$A_EXIT_ADDRESS field of the PCB. If a valid P0 address (as verified in step 1) was supplied, process status will be returned to the specified address when or if the process terminates.
10. A standard VAX argument list is created for the caller-specified process arguments and linked into the PCB. The kernel uses one of the two pool blocks previously allocated from system pool. If no arguments were specified, only the longword count (0) is copied to the pool block.
11. A timer wait control block (WCB) is created in the PCB$B_WCB field of the PCB, with a pointer (at offset PCB$B_WCB + WCB$A_LIST) to a list of four other WCBs residing in a pool block. (The pool block used is the second of the two previously allocated.)

    The WCB is the kernel structure for handling KER$WAIT procedure calls that synchronize process execution with events or the availability of resources. Each process has a timer WCB and at least four additional WCBs, each representing an element potentially involved in satisfying a wait request. WCBs and the role of KER$CREATE_PROCESS in initializing them are described in detail in Chapter 11.
12. The PTX address is calculated and placed in the PCB$A_PTX field of the PCB. (The PTX resides at the start of the system page shared by the PTX and PCB.)

4.5.1.3 Step 3 — Create the Process Hardware Context Block

The kernel next begins filling in the PTX as follows:

1. The name block field, PTX$A_NAME_BLOCK, of the PTX is cleared; initially the subprocess has no name. A process can name itself or another process in the same job with a KER$NAME_OBJECT (in VAXELN Pascal) or KER$CREATE_NAME procedure call.

2. The physical address of the PTX is derived and placed in the PCB$A_HWPTX field of the PCB. The physical address of the PTX is used (rather than its virtual address) when the kernel inserts its address into the hardware process control block base register (PCBB) when the process is scheduled to run.

3. The four stack pointer address fields in the PTX are set equal to P1$K_KERNEL_STACK_INIT — $7FFFFDF0_{16}$ — the P1 address of the initial top of the kernel stack, which equals the base of the kernel stack. Kernel and user stack allocation occurs later in process creation. The executive and supervisor stack fields (PTX$A_ESP and PTX$A_SSP) are never used.

   Note that the initial kernel stack top, P1$KERNEL_STACK_INIT, is –16 bytes offset from the bottom (the high-address end) of the kernel stack page, indicated by constant P1$KERNEL_STACK_TOP. Intervening are four P1 context longwords — P1$GA_JCX, P1$GA_ADA_CTX, P1$GA_ADA_DATA, and P1$GA_CRTL_CTX — that hold jobwide, Ada, and C RTL context addresses. Although the four longwords share a page with the beginning of the kernel stack, they are not part of the kernel stack. This is illustrated in Figure 4–6.

4. In preparation for the second phase of process creation, the general register fields of the PTX are set up with values used in that phase. One of the saved values is the transfer address for the process's code, as provided in the initial call to KER$CREATE_PROCESS.

5. The PC and PSL fields of the PTX are prepared for transfer to the next phase of process creation. The PTX$A_PC field receives the address of the FINISH_PROCESS subroutine within module CREATEPRO. When the new process is first scheduled to run, it will reenter the CREATEPRO module at FINISH_PROCESS.

   The PTX$L_PSL field is cleared to initialize access mode to kernel (0) and IPL to 0 for when the process is first scheduled to run.

6. The P0BR and P0LR processor register fields in the PTX are set up. These values will be loaded into the P0BR and P0LR registers when the process runs. The P0BR and P0LR processor registers define the P0 address space of the process and help the processor locate the corresponding physical memory. The PTX$A_P0BR and PTX$L_P0_LIMIT fields are established as follows:

   - The PTX$A_P0BR field receives the P0BR value, indicating the virtual base address of the process's P0 page table, copied from the JCB (field JCB$A_P0_BASE).

If the target VAX is KA620-based (such as an rtVAX 1000), the JCB's P0BR value is translated to a physical base address before it is stored in the PTX. (P0 and P1 page tables in a KA620 system are physically contiguous and accessed by physical address, which saves virtual address translations for process-space memory references.)

- The P0LR and ASTLVL subfields of the PTX$L_P0_LIMIT field are set up respectively with the P0LR value, indicating the effective length of the P0 page table, copied from the JCB field JCB$L_P0_LIMIT; and the value PSL$C_USER + 1 (equal to 4), indicating no asynchronous exception is pending.

4.5.1.4 Step 4 — Allocate a P1 Page Table

The process's P1 page table is established. A P1 page table slot is allocated that holds the process's P1 page table and P1 region allocation bitmap. In addition, the P1BR and P1LR processor register fields in the PTX are set up. The P1BR and P1LR values will be loaded into the P1BR and P1LR registers when the process runs. These registers define the P1 address space of the process, by describing the inaccessible portion of it. The P1BR register contains the virtual address of what would be the page table entry (PTE) for the first page of P1 memory (location 40000000_{16}. The P1LR register contains a value indicating the number of nonexistent PTEs (corresponding to inaccessible pages), following which is the first existent PTE (corresponding to the first accessible page).

The page table structures and related PTX fields are set up as follows:

1. The subroutine KER$ALLOCATE_P1_SLOT, in module ALLOCATE, is called to allocate and initialize a P1 page table slot. The KER$ALLOCATE_P1_SLOT routine is described in Chapter 9. The subroutine allocates a P1 page table slot (using the P1 slot allocation bitmap) and returns the system virtual addresses of the two items of interest in the slot — the P1 page table and the P1 region allocation bitmap.
2. The PTX$A_P1BR field receives the base address of the process's P1 page table. The value placed in this field is the page table base address that will be used in address translation — the base address of the nonexistent portion of the P1 page table. The value is calculated by converting the system virtual address of the memory allocated for the P1 page table (as stored in field PCB$A_P1_BASE of the PCB) to reflect the base address of the P1PTE that would map virtual address 40000000_{16}. If the target VAX is a

KA620-based system (such as an rtVAX 1000), the P1BR value is translated to a physical address.

3. The P1LR subfield of the PTX$L_P1_LIMIT field is set up with the P1LR value, indicating the length in PTEs of the nonexistent portion of the P1 page table (corresponding to inaccessible memory pages).

4.5.1.5 Step 5 — Allocate the First Page of Kernel Stack

The next step performed in the initial phase of subprocess creation is allocation of one page of the subprocess's kernel stack. The remainder of the kernel stack and a user stack (if the process is to execute in user mode) will be allocated in the second phase of subprocess creation. The initial page of kernel stack is part of the minimal context the subprocess needs to be scheduled to execute the second phase of subprocess creation.

4.5.1.6 Step 6 — Enter the PCB into the Job's Object Table

The PCB, as the kernel object representing the process, is entered into the job's object tables. The kernel allocates an object pointer table entry by calling the internal subroutine KER$ALLOCATE_OBJECT. The allocated pointer table entry receives the address of the PCB. The process object identifier returned by KER$ALLOCATE_OBJECT is placed in the PCB$L_ID field of the PCB, and its sequence number is placed in the field PCB$L_SEQUENCE.

4.5.1.7 Step 7 — Initiate a Scheduling Pass

At the conclusion of the first phase of process creation, the kernel calls the KER$READY_PROCESS kernel subroutine, which initiates a scheduling pass. The process might be selected to run immediately — for example, if the created process has a higher priority than the creating process (and the job continues in the running state) — or might be placed in the ready-process queue for its priority to await a change in system state that allows the job and process to run.

Finally, the KER$CREATE_PROCESS procedure returns to its caller with an REI instruction. When the newly created process is selected to execute by the scheduler and begins running, it runs in kernel mode and executes the FINISH_PROCESS subroutine to finish creating the subprocess.

4.5.2 Phase 2: Finishing Creation of the Process Environment

The main objective of the second phase of subprocess creation is to complete construction of the subprocess environment so that control can be transferred to the user-specified program mode, kernel or user, for the final phase, which sets up entry to the routine or process block transfer address.

The kernel takes the following steps to complete creation of the process environment:

1. The remaining portion of the kernel stack is allocated and, if the job's program mode is user, a user stack and guard page are allocated.
2. Transfer to program mode and to the KER$ENTER_PROCESS subroutine is set up.

4.5.2.1 Step 1 — Allocate the Process Stacks

To finish allocating the subprocess's stacks, the kernel calls the subroutine KER$ALLOCATE_PROCESS_STACK (in module CREATEJOB). Using the stack sizes specified in the program descriptor, KER$ALLOCATE_PROCESS_STACK allocates the remainder of the kernel stack (beyond the page already allocated) and, if the program mode is user, the user stack. If the user stack is allocated, an extra guard page is also allocated between the user stack and the kernel stack, to mark the end of the fixed-length kernel stack.

The KER$ALLOCATE_PROCESS_STACK subroutine executes as follows:

1. Field JCB$A_INITIAL_STACK of the JCB is set to the value P1$K_KERNEL_STACK_INIT, the P1 initial kernel stack address. (The kernel stack entry in Table 4–5 further discusses the initial address.) The initial-stack JCB field will be overwritten with the initial user stack address in a subsequent step, if the job's program mode is user.
2. The kernel stack size, in pages, is extracted from the program descriptor (field PRG$W_KERNEL_STACK) and is reduced by one to account for the page already allocated. If more pages are needed, the number of P1 page table entries required to map the additional kernel-stack pages is allocated with a call to kernel subroutine KER$ALLOCATE_P1_PTE. The PTEs are then filled in with page frame numbers (generated by successive calls to the

KER$ALLOCATE_FRAME subroutine) and with the following bits set in the remaining fields: PTE$M_VALID (valid), PTE$C_URKW (access), PTE$C_KOWN (owner), and PTE$K_RW_DATA (type).

3. If the job's program mode (PRG$B_MODE) is kernel, process stack allocation is complete. Steps 4 and 5 are bypassed.
4. If the job's program mode is user, the user stack size, in pages, is extracted from the program descriptor (field PRG$W_USER_STACK); if 0, a value of 1 is substituted, to guarantee 1 page of user stack for the process. Also, 1 is added to the count to allow for a guard page to separate the kernel and user stacks. Field JCB$A_INITIAL_STACK is reset to the initial user stack address, which is calculated by offsetting the address of the end of the kernel stack by the byte length of the guard page, –512. The initial user stack address is then written to the processor's internal user stack-pointer register (PR$_USP), in preparation for entering user mode in the third phase of process creation.
5. The number of P1PTEs required to map the user stack and the guard page are allocated with a call to kernel subroutine KER$ALLOCATE_P1_PTE. Using the prototype read/write data PTE in field JCB$L_RW_DATA_PTE of the JCB, the user-stack PTEs are then filled in with page frame numbers (generated by successive calls to the KER$ALLOCATE_FRAME subroutine). Finally, the PTE for the guard page is filled in to indicate the page is inaccessible.
6. P1LR is updated from the PCB$L_P1_LIMIT field of the PCB (updated during the P1 mapping allocations), the P1 context addresses (except P1$GA_JCX) are cleared, and KER$ALLOCATE_PROCESS_STACK returns.

4.5.2.2 Step 2 — Begin Program Execution

The final step in the second phase of subprocess creation is to transfer control to the user-specified program mode and to the KER$ENTER_PROCESS subroutine, which will set up entry to the process entry point specified in the KER$CREATE_PROCESS call.

Prior to the execution of the REI instruction that triggers the possible change of access mode, general registers are set up with the addresses of the process's arguments, the PCB, the JCB, the job context page, the job's program descriptor, and the routine's entry point.

The kernel sets up for the REI instruction mechanism. The user-specified program mode is pushed onto the stack. If the specified mode is user, the program PSL on the stack is initialized to specify user access privileges. Finally, the address of the KER$ENTER_PROCESS subroutine (in module CREATEJOB) is pushed on the stack as the address to which control is to be transferred. The kernel then executes an REI instruction, which pops these PSL and PC values from the stack into their respective registers. The process is now executing in its base access mode at location KER$ENTER_PROCESS.

4.5.3 Phase 3: Entering the Process Code

The main objective of the final phase of subprocess creation is to set up entry to the process code at its transfer address (specified in the initial KER$CREATE_PROCESS call), so that execution can begin. This phase is identical to the final phase of job creation, in which the master process is set up to enter the program at its transfer address.

When a process continues executing following the REI instruction executed in the second phase, it executes the KER$ENTER_PROCESS subroutine. This routine, which executes at the close of both job and process creation, causes the process to be entered and program execution to begin (when the process next becomes eligible to run).

If debugging was requested and the debugger is present in the system, the debugger bootstrap, routine BOOTSTRAP_PROCESS (in module [DEBUG]LCLNUC), is entered before the new subprocess executes process code. The debugger bootstrap calls a debugger subroutine, INIT_PROCESS_CONTEXT, to establish the process-specific debug context for the subprocess in the last page of P1 address space; see Figure 4–6. INIT_PROCESS_CONTEXT also sends a message to VAXELN$DEBUG_PORT to announce the presence of the subprocess. Furthermore, unless the user requested that this job's processes start without debugger intervention, the debugger bootstrap calls the debugger's first-chance exception handler as if a KER$_DEBUG_SIGNAL exception occurred, which causes the subprocess to await a debugger command. If a debugger command starts execution, control returns to the debugger bootstrap, which then executes an REI instruction to start subprocess execution.

Similarly, if performance collector PC coverage activity was requested, the performance collector utility is entered before the subprocess executes program code.

The KER$ENTER_PROCESS subroutine checks for build-time selection of the debugger or performance collector to run in conjunction with this job. A CALLG instruction is executed to transfer control to the debugger, to the performance collector utility, or directly to the process's routine, function, or process block entry point.

While executing, if the process reaches the end of its code without issuing a KER$EXIT procedure call (or a KER$DELETE for itself), the RET instruction generated by the compiler is executed. When this happens, control returns to the KER$ENTER_PROCESS subroutine, at the instruction after the CALLG used to invoke the process code. KER$ENTER_PROCESS proceeds to initiate the orderly termination of the process, as follows:

1. The process exit status and the user-supplied status value address are pushed on the stack.
2. The KER$EXIT kernel procedure is called. The call does not return; the KER$EXIT kernel procedure completes with a call to the KER$DELETE kernel procedure, which deletes the process. After process deletion, the kernel branches to the internal routine KER$SCHEDULE_PROCESS to schedule the next process.

The KER$EXIT and KER$DELETE kernel procedures are described in Section 4.6.

4.6 Job and Process Exit and Deletion

Deleting a VAXELN process with the KER$DELETE kernel procedure deactivates an execution thread within the system. If the process undergoing deletion is a master process, the entire job is terminated, its subprocesses are deleted, and its system resources are freed. (If the job was executing a dynamically loaded program, deletion may also unload the program from the system.) If the process's termination began with an implicit exit — due to the process reaching the end of its block or routine — or with an explicit call to KER$EXIT from program code, additional orderly cleanup is performed before the process is deleted.

A VAXELN process is deleted under the following circumstances:

- Implicit exit from program code. The process reaches the end of the procedure code it is executing. A RET instruction in the compiler-generated code is executed, returning control to the KER$ENTER_PROCESS subroutine (in module CREATEJOB) that initiated the

process's execution. The KER$ENTER_PROCESS subroutine calls the KER$EXIT procedure on behalf of the process. KER$EXIT performs orderly cleanup and then invokes the KER$DELETE kernel procedure to delete the process.

- Explicit exit from program code. The process executes a KER$EXIT procedure call in the program code. KER$EXIT performs its cleanup and then invokes KER$DELETE.
- Explicit deletion from program code. The process executes a KER$DELETE procedure call to delete itself or is the object of a deletion call by another process. (The process also may be the target of a DELETE PROCESS request from the debugger, or be forced into deletion by a kernel or RTL module that detects a fatal error.)
- Unhandled exception. An exception is raised in the process and is not handled by the process or from the debugger. (This includes unhandled asynchronous exceptions, such as those raised by the KER$SIGNAL and KER$RAISE_PROCESS_EXCEPTION procedures.) The kernel forces the process to terminate by calling KER$EXIT with the exception (signal) name as the exit status. For more details on exception handling, see Chapter 6.
- Fatal process-level bugcheck. Some serious error in the process's context has caused the kernel to issue a fatal bugcheck for the process. The subroutine KER$BUG_CHECK (in module BUGCHECK) forces the process to exit by calling KER$EXIT with KER$_BUGCHECK as the exit status.
- Subprocess's master process is deleted. Deleting a master process causes KER$DELETE to be invoked for each subprocess in the job.

The VAXELN procedure that deletes object-related kernel resources, KER$DELETE (in module DELETE), is described in Chapter 10. Section 4.6.1 describes the actions KER$DELETE takes to delete a VAXELN subprocess. Section 4.6.2 describes the additional job object rundown, memory deallocations, and dynamic program unloading involved when a master process is deleted.

4.6.1 Process Deletion

This section describes the steps in process deletion that are common to both subprocess and master process deletion. The additional actions that are performed for master-process deletion — deletion of the job, deallocation of job resources, and dynamic program unloading — are described in Section 4.6.2.

The actions taken by the KER$DELETE procedure to delete a process are as follows:

1. If no exit status has been set for the process — as indicated in field PCB$L_EXIT_STATUS of the PCB — the status value KER$_NO_STATUS is placed in that field.
2. The process's accumulated CPU time, in field PCB$L_CPU_TIME, is added to the accumulated job CPU time in field PCB$L_JOB_CPU_TIME of the master process PCB. The resulting value represents the total accumulated CPU time of all deleted processes in the job.
3. The object table entry allocated for the process PCB is freed with a call to the internal subroutine KER$FREE_OBJECT.
4. If the process creator requested exit status, and the location specified for it is writeable in the program's mode, the kernel moves the exit status from the PCB$L_EXIT_STATUS field of the PCB to the caller-specified location.
5. The waits of all processes waiting on this process deletion are potentially satisfied. All the process's WCBs are dequeued and processed in turn. For each WCB linked into the process's PCB, the wait state is set to satisfied (WCB$B_SATISFIED), then (unless the process was waiting on itself) the internal routine KER$TEST_WAIT is called to determine whether the wait on the process being deleted is completely satisfied as a result. If so, the internal routines KER$SATISFY_WAIT and KER$UNWAIT, described in Chapter 11, are called to formally satisfy the wait. The wait completion status returned is KER$_SUCCESS.
6. If the current process is deleting a process other than itself (that is, a process not in the running state), the specified process is removed from the appropriate state queue or JCB pointer field — the ready queue for its priority if it is in a ready state, the wait queues it resides in if it is in a waiting state, or the JCB$A_NEXT_PCB slot in the JCB if it is the designated next process to run in the job.

If the current process is deleting itself, an SVPCTX instruction is executed to get onto the interrupt stack and off the process stack.

7. The process's P1 page table and page table entries are freed with a call to the local subroutine FREE_PTE and the internal routine KER$FREE_P1_SLOT.
8. The control blocks associated with the process are freed. A series of calls to the internal routine KER$FREE_POOL frees the process argument block, pointed to by the PCB$A_ARGUMENT field of the PCB, and all pool blocks occupied by the process's WCBs, pointed to by the PCB$B_WCB field of the PCB. The pointer to the PCB is cleared from the process's name block — if the process was named. The process's PCB is unlinked from the job's list of processes. Finally, the S0 page containing the PCB and the PTX is freed with a call to the internal routine KER$FREE_REGION.
9. If the current process is deleting itself, a new process must be scheduled to run, so the procedure branches to KER$SCHEDULE_PROCESS, which leads to an exit through the REI instruction.
10. If deleting the last process in the job's list — the master process — a new job must be scheduled to run on the current processor, so the procedure branches to KER$SCHEDULE_JOB, which leads to an exit through the REI instruction.
11. The KER$DELETE call exits with an REI instruction, returning successful completion status KER$_SUCCESS.

4.6.2 Master Process Deletion

When a master process is deleted, the kernel deletes all processes in the job (following the steps listed in Section 4.6.1 for each process), deletes the job, and frees the job's resources. Additionally, if the program executed by the job is a dynamic program and a request has been received to unload it, the program is unloaded, provided no other job is executing it.

The actions taken by the KER$DELETE kernel procedure (in module DELETE) to delete a master process are all the steps listed for a subprocess in Section 4.6.1, along with the following additional steps:

1. The KER$GQ_PREV_JOB_TIME global variable, which accumulates the CPU time used by deleted jobs, is updated. The accumulated job CPU time in field PCB$L_JOB_CPU_TIME of the master process PCB is converted to standard VAX time units (by multiplying times the time interval value in global longword KER$GL_

TIME_INTERVAL), and the quadword result is added to KER$GQ_PREV_JOB_TIME.

2. The kernel procedures KER$CREATE_MESSAGE and KER$SEND are used to send a job termination message to the job exit port, if one was specified by the job's creator.
3. If the current process is a subprocess and is deleting the master process, the current process is made the master process; this allows the KER$DELETE procedure to be called from the current process to delete all the job's objects.
4. The kernel loops through the job's linked list of ports, pointed to by the JCB$A_PORT_FLINK field of the JCB, and calls KER$DELETE to delete each one.
5. If the program executed by the job is a dynamic program, as determined by referencing the program description pointed to by the JCB$A_PROGRAM field of the JCB, the program reference count field, PRG$W_REF_COUNT, is decreased by 1 to indicate that a job executing the program has been deleted. If the program descriptor indicates a request has been received to unload the program on completion, and if PRG$W_REF_COUNT indicates no more jobs are executing it (a zero value), the internal subroutine KER$DELETE_PROGRAM is called to remove the program from the system.
6. All job-created objects are deleted. The kernel walks the tables of job objects, pointed to by the JCB$A_OBJECT_TABLE field of the JCB, and deletes each object. The pool space occupied by the object tables is freed, and the S0 page containing the object base table is freed. The object tables are described in Chapter 10.
7. The job's P0 page table and all its entries are freed. The internal subroutine KER$FREE_P0_SLOT, described in Chapter 9, is called to free the P0 slot containing the page table and the P0 region and slot allocation bitmaps.
8. The job's process queue-listhead pool block, pointed to by field JCB$A_PROCESS_QUEUES of the JCB, is freed with a call to internal subroutine KER$FREE_POOL.
9. The job's JCB is removed from the system job list and, if appropriate, its bits in the KER$AW_CLASS_MASK array and the global active summary, KER$GL_ACTIVE_SUMMARY, are cleared.
10. Finally, the S0 page containing the JCB and the S0 page containing the PTX and the PCB are freed with calls to the internal subroutine KER$FREE_REGION. A final branch is taken to the internal subroutine KER$SCHEDULE_JOB to schedule the next job.

Chapter 5

Software Interrupts, Kernel Synchronization, and Time Support

Software interrupts, synchronization, and time services are critical to the operation of a real-time system. Software interrupts, which invoke specific service routines, allow vital system functions to occur asynchronously to the execution of jobs and processes. To maintain the integrity and consistency of its internal data bases, the VAXELN Kernel synchronizes its operations to enforce exclusive access to critical data and code sections. In addition, the kernel maintains a system clock and a timer mechanism and provides a set of procedures to support time-based synchronization.

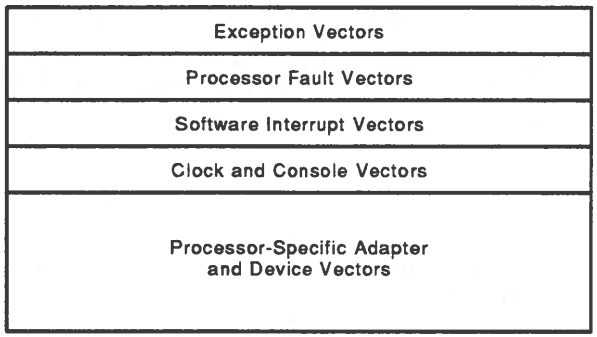
This chapter describes the kernel operations related to these topics:

- Software interrupts and their service routines are surveyed in Section 5.1.
- The kernel's internal synchronization techniques are described in Section 5.2.
- The time support provided by the kernel is described in Section 5.3.

All three of these mechanisms depend to some degree on architecturally defined vectors in the system control block (SCB). Figure 5–1 shows the general layout of a VAX SCB. The first four blocks of vectors appear on the architecturally defined first half-page of the SCB and are shared by all VAX processors. The size and contents of the remainder of the SCB vary with processor type.

In particular, the software interrupt vectors on the first page of the SCB support the VAXELN services described in this chapter. (Chapter 6 describes the role of the exception vectors in VAXELN condition handling.)

Figure 5–1: General Layout of a VAX SCB



5.1 Software Interrupts

The software interrupt mechanism supported by the VAX hardware plays a key role in the kernel's management of system events, such as job and process scheduling and device I/O. Software interrupt service routines running at interrupt priority levels (IPLs) 2 through 8 perform a number of the kernel's most important functions.

This section describes how software interrupts are requested and granted and summarizes their use by the kernel.

5.1.1 Software Interrupt Mechanism

A software interrupt is an interrupt requested by a write to the software interrupt request register (SIRR) rather than through an interrupt from an external device. The kernel requests a software interrupt to invoke system functions as they are needed, without having to test periodically whether each function must be performed.

The VAX interrupt microcode responds to software interrupt requests as it does to hardware interrupts: it dispatches through the appropriate SCB vector, which contains the address of the interrupt service routine (ISR). The setting of the lowest two bits in the vector specifies whether the interrupt will be serviced on the kernel or the interrupt stack.

The VAX architecture provides 15 vectors in the SCB for software interrupts at IPLs 1 through 15. IPLs are assigned to software interrupt services on the basis of two factors: their relative importance and their need to synchronize access to shared data. A software interrupt at a particular IPL is requested by writing that IPL into the SIRR (PR$_SIRR). The kernel generally uses symbolic values and macros to request software interrupts. For example, to reschedule the system, the kernel invokes the RESCHEDULE macro:

```
RESCHEDULE
        mtpr    #IPL$K_RESCHEDULE,#PR$_SIRR
```

Some hardware interrupt service routines, such as the interval timer ISR, request software interrupts as well. In addition, the REI instruction requests IPL 2 software interrupts to deliver asynchronous exceptions.

5.1.2 VAXELN Software Interrupt Service Routines

Table 5–1 shows the software interrupt service routines used by the kernel and its subsystems. None of these routines is described in this section. Instead, as indicated in the table, the routines are discussed in the larger context of the kernel services they perform. For example, the IPL 2 ISR plays a central role in the delivery of asynchronous exceptions and is therefore discussed in Chapter 6, Condition Handling.

Table 5–1: VAXELN Software Interrupts and Service Routines

IPL	Purpose	Stack	Service Routine
9–15	Unused	N/A	None
8	Servicing device queue	Interrupt	KER$DEVICE_SIGNAL in module SIGNALDEV. When the KER$SIGNAL_DEVICE procedure is called, the kernel places the device object in the device queue and requests a software interrupt at IPL 8. This service routine removes device objects from the queue and unblocks the processes waiting on them. See Section 11.3.2.
7	Software timer	Interrupt	KER$SOFTWARE_TIMER in module TIMERINT. The ISR for the hardware timer requests this software interrupt when the first entry in the timer queue has expired. The software timer ISR removes expired entries from the queue and unblocks the processes waiting on them. Section 5.3.5 describes software timer ISRs.
6	Secondary processor service	Interrupt	KER$SOFTWARE_AMP in module BIPORT. This service routine provides fork dispatching for KA800 interprocessor interrupts on closely coupled symmetric multiprocessing systems.
5	Kernel debugger	Interrupt	KER$DEBUG_INTERRUPT in module INITIAL. This service allows entry into the kernel debugger by executing a Breakpoint (BPT) instruction within the kernel. This software interrupt can be requested by the console or the SET SESSION/KERNEL command in the local debugger.
4	Job-level rescheduling	Kernel	KER$RESCHEDULE in module SCHEDJOB. This service routine saves the hardware context of the current process, then finds another job and/or process to run. Process rescheduling within a job is requested through an IPL 2 interrupt.
3	Unused	N/A	None
2	Asynchronous exception delivery	Kernel	KER$AST_INTERRUPT in module ASTDELIVR. This service routine delivers an exception asynchronously into the current process. See Section 6.5.

The kernel treats all software interrupts, except the asynchronous-exception delivery and rescheduling interrupts, as systemwide events

that are serviced outside the context of a specific process. The rescheduling interrupt is taken on the kernel stack of the current process. The interrupt service routine immediately executes a SVPCTX instruction, saving the process's context and switching execution to the interrupt stack. The asynchronous exception interrupt is the only interrupt that is serviced in the context of a specific process.

5.2 Kernel Synchronization

Within the kernel, synchronization involves blocking all but one of two or more events when their simultaneous occurrence might disrupt the proper operation of the system. Most often, such synchronization is required to ensure the integrity of shared data, so that a single thread of execution has exclusive access while reading or writing sensitive data structures. At the level of job and process execution, synchronization involves the use of kernel objects — areas, events, semaphores, and ports — and the KER$WAIT and KER$SIGNAL procedures to synchronize execution with real-time events and to control access to shared data and critical regions of code. Chapter 11 is devoted solely to the topic of job and process synchronization.

The kernel relies on a combination of the following software techniques and VAX hardware features to synchronize access to shared data structures:

- Interlocked instructions. These VAX instructions, such as INSQHI and REMQHI, ADAWI, and BBSSI, synchronize multiprocessor access to shared queues, aligned words, and bit fields. See Section 5.2.1.
- Elevated IPL. Elevating processor IPL on single-processor systems blocks all further system activity that occurs at that IPL and below. See Section 5.2.2.
- Multiprocessor spinlocks. On tightly coupled symmetric multiprocessing systems, elevated IPL is inadequate to synchronize access to system structures. Spinlocks — special-purpose bits that represent specific system resources — are obtained with the interlocked test-and-set instructions (for example, BBSSI) to enforce exclusive access by a single processor. See Section 5.2.3.

- Interprocessor interrupts. Occasionally, on tightly coupled symmetric multiprocessing systems, one processor must inform another processor of an event; this is a form of synchronization. The kernel provides a mechanism to allow interprocessor synchronization through interprocessor interrupts. See Section 5.2.4.

5.2.1 Interlocked Instructions

When a VAX interlocked instruction accesses a structure, it inhibits a similar interlocked access to the same structure by any other processor in the system. The VAX architecture provides the following interlocked instructions:

- ADAWI (Add Aligned Word, Interlocked)
- BBCCI (Branch on Bit Clear and Clear, Interlocked)
- BBSSI (Branch on Bit Set and Set, Interlocked)
- INSQHI and INSQTI (Insert into Queue Head/Tail, Interlocked)
- REMQHI and REMQTI (Remove from Queue Head/Tail, Interlocked)

These instructions are used throughout the kernel where a single access to a shared structure is required. Here are some examples of such usage:

- The global value KER$GW_ERRSEQ, the error log entry sequence number, is incremented with the ADAWI instruction.
- Multiprocessor spinlocks (see Section 5.2.3) are obtained with the BBSSI instruction and released with the BBCCI instruction.
- System pools blocks are obtained from the list of free blocks with a single REMQHI instruction and are returned with a single INSQTI instruction.

5.2.2 Elevated IPL

The primary purpose for raising IPL is to block interrupts at the selected IPL value and all lower values. The kernel uses specific IPL values to synchronize access to certain structures.

The IPL, stored in the processor status longword (PSL) register bits <20:16>, is altered by writing the desired IPL value to the privileged register PR$_IPL. This change in IPL is usually accomplished with the SETIPL macro:

```
.MACRO  SETIPL NEWIPL
        mtpr    NEWIPL,#PR$_IPL
.ENDM   SETIPL
```

This macro changes IPL to the value specified by NEWIPL. Other macros are defined to set IPL to specific levels; for example, the DISABLE_SWITCH macro sets IPL to 3, preventing delivery of the IPL 2 interrupt, which enables one process to preempt another.

To synchronize successfully, IPL must be raised — but not lowered — to the appropriate synchronization level. Lowering defeats any attempt at synchronization and also runs the risk of a reserved operand fault when an REI instruction is later executed (an REI instruction that attempts to elevate IPL causes the fault).

Table 5–2 shows several IPLs that are used for synchronization within the kernel.

Table 5–2: Common IPL Values Used by the Kernel for Synchronization

Name	Value (decimal)	Meaning
IPL$K_KERNEL_DEBUG	31	Disable all interrupts.
IPL$K_POWER	30	Disable all interrupts.
IPL$K_INTERPROCESSOR	23	Block interprocessor interrupts.
IPL$K_SYNCHRONIZE	8	Synchronize access to kernel data structures.
IPL$K_TIMER	7	Block software timer software interrupts.
IPL$K_RESCHEDULE	4	Block job rescheduling.
IPL$K_DISABLE_SWITCH	3	Block process context switching.
IPL$K_AST_LEVEL	2	Block asynchronous exception software interrupts.

The most common instances of IPL synchronization are the uses of IPL$K_DISABLE_SWITCH and IPL$K_SYNCHRONIZE. The kernel's procedure-dispatching code (described in Section 8.2) raises IPL to IPL$K_DISABLE_SWITCH before branching to the kernel procedure

code. Raising IPL to this level prevents the delivery of the IPL 2 interrupt, which initiates a context switch to preempt a process. Process switching within a job must be inhibited during execution of most kernel procedures, because another process, which shares P0 address space with the current process, could delete or corrupt memory required for the execution of the procedure.

IPL$K_SYNCHRONIZE (8) is the IPL at which the device and timer queues are serviced. Before most kernel data structures are accessed (for example, the scheduler database), IPL must be raised to this level. By raising IPL to 8, all other processes that might access the same systemwide data structure are blocked from execution until IPL is lowered. While the processor is executing at IPL 8, certain systemwide events, such as scheduling and timer and device queue servicing, are blocked. More important operations, however, such as hardware interrupt servicing, can continue.

Within the kernel, IPL is elevated to IPL$K_SYNCHRONIZE with the SYNCHRONIZE macro:

```
.MACRO SYNCHRONIZE
        setipl  #IPL$K_SYNCHRONIZE
.ENDM SYNCHRONIZE
```

The SETIPL macro call within SYNCHRONIZE generates the MTPR instruction that writes IPL 8 to the PR$_IPL register.

5.2.3 Spinlocks

In a tightly coupled symmetric multiprocessing system, each processor has its own interrupt priority level, independent of the others. On these systems, then, raising IPL ensures synchronization on a single processor but not across the entire system. Therefore, the kernel employs a mechanism called the spinlock to provide synchronization on multiprocessor systems. Anywhere the kernel synchronizes by raising IPL on a single-processor system, it must also acquire a spinlock on a multiprocessor system.

A spinlock is a bit that represents a system resource or critical section of code. When the bit is clear, the resource or code section is available. To acquire the lock, a processor sets the bit. When another processor finds the bit set, it loops — spins — on the lock bit until the lock's owner releases it by clearing the bit; the other process can now acquire the lock and access the resource.

The bits that constitute the kernel's spinlocks reside in a single longword in the kernel data block, KER$GL_MULTIPROCESSOR_LOCK. Table 5–3 shows the spinlocks currently defined in KER$GL_MULTIPROCESSOR_LOCK and the resources they protect.

Table 5–3: Kernel Spinlocks

Spinlock	Bit Position	Function
KER$V_GENERAL	0	Protects most kernel resources, such as the scheduler and memory management databases.
KER$V_BUGCHECK	1	Allows only one processor to bring down the system during a fatal system bugcheck.
KER$V_CREATE_DEVICE	2	Protects the device database during device creation by KER$CREATE_DEVICE.
KER$V_VIRT_CONSOLE	3	Ensures that a processor has exclusive access to the virtual console.

In the kernel, spinlocks are usually acquired by use of the LOCK and SEIZE macros. The LOCK macro normally appears where IPL would be raised to IPL$K_SYNCHRONIZE and, in fact, performs that operation as well:

```
.MACRO LOCK LOCK_NAME
        synchronize
        seize   LOCK_NAME
.ENDM SYNCHRONIZE
```

The call to the SYNCHRONIZE macro raises IPL to IPL$K_SYNCHRONIZE on the current processor. The call to the SEIZE macro then generates the code to acquire the specified spinlock.

The SEIZE macro is conditionalized to generate spinlock instructions only for multiprocessing versions of the kernel; therefore, on single-processor systems, synchronization remains a matter of raising IPL to IPL$K_SYNCHRONIZE. The SEIZE macro uses the BBSSI test-and-set instruction to acquire the specified spinlock. If the spinlock bit is clear, the BBSSI instruction sets it to acquire the lock, and execution continues with the next instruction. If the bit is already set, meaning that the lock is in use, the BBSSI instruction is reexecuted. Thus this call to the SEIZE macro —

```
SEIZE LOCK=GENERAL
```

— would generate code like this:

```
10$:    BBSSI   #KER$V_GENERAL,W^KER$GL_MULTIPROCESSOR_LOCK,10$
NEXT_INSTRUCTION:
        .
        .
        .
```

A spinlock is relinquished with the RELEASE macro. Like the SEIZE macro, RELEASE is conditionalized to generate code only for multiprocessing versions of the kernel. For those systems, RELEASE generates the requisite BBCCI instruction to clear the specified spinlock bit. Since RELEASE is called only after a call to SEIZE, the processor should not need to spin to clear the lock bit.

The UNLOCK macro is the complement of the LOCK macro — it relinquishes the previously acquired spinlock and lowers IPL from IPL$K_SYNCHRONIZE to IPL$K_DISABLE_SWITCH:

```
.MACRO UNLOCK LOCK_NAME
        release LOCK_NAME
        disable_switch
.ENDM SYNCHRONIZE
```

UNLOCK is called only from kernel code that was executing at IPL$K_DISABLE_SWITCH before calling the LOCK macro. This is the case for the majority of kernel procedures, which execute at least at IPL 3. Kernel code that wishes to relinquish a lock without affecting IPL calls the RELEASE macro directly.

5.2.4 Interprocessor Interrupts

On tightly coupled symmetric multiprocessing systems, the kernel provides a way for one processor to synchronize its activities with one or more other processors by requesting an interprocessor interrupt. Interprocessor interrupts are generated on the VAX 6000 and VAX 8800 series processors at IPL 23.

Interprocessor interrupts are generated by the INTERRUPT_CPU and INTERRUPT_ALL_CPUS macros (in modules MP8800HDR and MP6CCHDR). An argument to the INTERRUPT_CPU macro specifies the number of the processor to be interrupted. Both macros take an argument that specifies the reason for the interrupt. The reason corresponds to a bit in the global bit field KER$AB_REASON in the kernel data block. Table 5–4 shows the reason bits defined in KER$AB_REASON and describes their meanings to the processors that receive the interrupt.

Table 5–4: Interprocessor Interrupts

Reason	Bit Position	Meaning
KER$V_JOB_SCHEDULE	0	Initiate job scheduling. This interprocessor interrupt is requested by the scheduler when it discovers that a job running on another processor requires preemption.
KER$V_FLUSH_TB	1	Flush the entire address translation buffer. This interprocessor interrupt is requested by kernel memory allocation routines whenever system page tables entries are altered. The interrupt informs all processors that their translation buffers may be invalid.
KER$V_CROSS_JOB_SIGNAL	2	Request an IPL 2 interrupt to deliver a debugger halt signal to a process. This interprocessor interrupt is requested by the debugger when it discovers that a user has requested that a process running on another processor be halted. The IPL 2 interrupt on the target processor allows the halt request to be delivered as an asynchronous exception.
KER$V_REQUEST_SHUTDOWN	3	Perform an orderly processor shut-down. This interprocessor interrupt is requested by a processor undergoing a fatal system bugcheck to bring down the other processors in the system.

The macros set the appropriate bit in the KER$AB_REASON mask and then request the interprocessor interrupt. The interrupt service routine, KER$INTERPROCESSOR_INTERRUPT (in modules GENMP8800 and GEN6CC), executing at IPL 23 on the interrupt stack, scans the reason mask until it finds the set reason bit. It then clears the bit, performs the requested action, and dismisses the interrupt.

5.3 Time Support

Support for activities that must occur at an absolute date and time or must measure an interval of time is implemented in both the VAX hardware and in the VAXELN Kernel. This chapter describes the kernel's support for these time-based operations.

A hardware component called the interval clock interrupts the processor at regular intervals. The kernel uses this clock to keep time and to service time-dependent wait requests. It is the key to all time-dependent activities and is described in Section 5.3.1.

A single time is maintained under VAXELN, the current date and time (the system time). Another time, the time elapsed since the system was bootstrapped (the system uptime) is fabricated and returned to users, such as the debugger and the display utility, on request.

Keeping time and servicing time-dependent requests require both a hardware interrupt service routine (ISR) for the interval clock and a software interrupt service routine. The hardware ISR, described in Section 5.3.4, maintains the system time and requests the software timer interrupt as necessary. The software timer ISR, described in Section 5.3.5, supports time-dependent waits by examining a time-ordered queue of requests and unblocking their associated processes as their expiration times occur.

The kernel also provides a number of procedures to service time-related requests: KERSET_TIME, KERGET_TIME, KER$GET_UPTIME. These procedures are described in Section 5.3.6.

5.3.1 Interval Clock

The VAX hardware clocks are updated regularly by timing circuitry. Under VAXELN, only the interval clock is used to maintain the system time; no use is made of the internal time-of-day clock.

All VAX processors implement an interval clock, which can interrupt at interrupt priority level (IPL) 22 or 24 at intervals of at least ten milliseconds. In processors that employ the VAX subset architecture, this timer is implemented as a single bit, in the internal register PR$_ICCS, whose setting enables interrupts every ten milliseconds. The MicroVAX, VAXstation, and VAX 6000 series processors implement this minimum interval clock and can therefore interrupt only at ten-millisecond intervals.

On VAX processors that implement the full VAX architecture, such as the VAX–11/750 and VAX 8820, two additional processor registers, PR$_ICR and PR$_NICR, allow further control of the interval clock.

The interval clock is updated at one-microsecond intervals with an accuracy of at least 0.01 percent (an error of fewer than nine seconds per day). The frequency at which the interval clock causes an interrupt is determined by the value in PR$_NICR.

In the full implementation, the three interval clock registers are used as follows:

- The interval clock control/status register (PR$_ICCS) controls the interrupt status of the interval clock. This register is set at system initialization, then reset by the interval clock ISR to indicate that the interrupt has been serviced and to reenable interrupts (see Section 5.3.4).
- The next interval count register (PR$_NICR) defines how often the interval clock will cause a hardware interrupt—a clock "tick." At system initialization of processors that support intervals other than ten milliseconds, this processor register is set to the value specified by the user as the **Time interval** entry on the System Characteristics Menu. This interval defines the minimum granularity for time-related operations. For example, the smallest amount of time a process can wait is one clock tick — the time interval.

  The interval value is stored, in units of 100 nanoseconds, in the kernel parameter KER$GL_TIME_INTERVAL. Before the kernel initializes PR$_NICR, it converts this value to microseconds. The default interval value is –10000, which specifies an interval clock interrupt period of ten milliseconds (10,000 microseconds). On subset processors, attempts to set PR$_NICR are ignored; the clock will interrupt only at ten-millisecond intervals.
- Every microsecond, the hardware increments the interval count register (PR$_ICR). When the interval clock is initialized, the processor copies the negated value of PR$_NICR to PR$_ICR. As each microsecond passes, the value of PR$_ICR is incremented from the PR$_NICR value toward zero. When PR$_ICR becomes zero, the register overflows, with the following results:
  1. The hardware copies the contents of PR$_NICR into PR$_ICR to define the next interval.
  2. The hardware sets a bit in PR$_ICCS to indicate the overflow condition. This causes an interval clock interrupt.

In VAX subset processors, which contain only the single-bit version of PR$_ICCS, the value written to PR$_NICR during system initialization is ignored. Only the original setting and subsequent resetting of the single interrupt-enable bit in PR$_ICCS to enable ten-millisecond interrupts is significant.

The IPL at which the hardware interrupt occurs is either 22 or 24, depending on the processor type. Earlier VAX processors, such as the VAX–11/750, use IPL 24. The VAX architecture now defines 22 as the IPL associated with the interval clock.

5.3.2 Timekeeping Under VAXELN

Timekeeping under VAXELN involves maintaining the system time and servicing time-dependent waits. The system time is stored in the 64-bit global value KER$GQ_SYSTEM_TIME. This value represents the number of 100-nanosecond intervals since 00:00 hours, November 17, 1858, the base time for the Smithsonian Institution astronomical calendar. KER$GQ_SYSTEM_TIME is updated, by default, every ten milliseconds by the interval clock ISR. On processors that implement the full interval clock, the interrupt interval can be set by the user in the System Builder (see Section 5.3.1).

Because the value of KER$GQ_SYSTEM_TIME is set to 0 when the kernel is assembled and is not changed by system initialization, the system time on VAXELN systems begins at the Smithsonian base time. The KER$SET_TIME procedure allows the system time to be set under program control to a correct value, as described in Section 5.3.6.1.

The kernel maintains other global values for time-related operations. These are shown in Table 5–5. (All the values except KER$GL_TIME_INTERVAL are defined in module SYSTEMDAT. The time interval is defined in module PARAMETER.)

Table 5–5: Time-Related Kernel Values

Value	Use
KER$GB_TIME_SET	A flag to indicate that the system time has been set. This flag is set the first time the system time is set. When the flag is clear, the KER$GET_TIME procedure warns its callers that they are receiving the uncorrected base system time.

Table 5–5 (Cont.): Time-Related Kernel Values

Value	Use
KER$GQ_CLOCK_OFFSET	An accumulator for all changes to the system time. Stored in units of 100 nanoseconds, this value allows the KER$GET_UPTIME procedure to calculate system uptime based on the system time.
KER$GQ_IDLE_TIME	An array that accumulates, in 100-nanosecond intervals, the idle time for each processor.
KER$GQ_START_TIME	The time at which system time was last set.
KER$GQ_SYSTEM_TIME	The absolute system time in 100-nanosecond intervals.
KER$GQ_TIME_QUEUE	Listhead of the timer queue, which contains the timer wait control blocks representing timed process waits. See Section 5.3.3.
KER$GL_TIME_INTERVAL	The interval clock interrupt period in units of 100 nanoseconds. This value is based on the **Interval time** entry on the System Characteristics Menu. The value entered there in units of microseconds is multiplied by 10 to generate the value for KER$GL_TIME_INTERVAL. The default value is 100,000 nanosecond intervals, or 10 milliseconds. This value is stored in 100-nanosecond units so that the interval clock ISR can then update the system time by adding KER$GL_TIME_INTERVAL to KER$GQ_SYSTEM_TIME, which uses the same units.

5.3.3 Timer Queue and Timer Wait Control Blocks

The timer queue, central to the processing of timed waits, is a list of timer wait control blocks (WCBs), each containing a quadword time value representing the absolute system time at which a process wait expires. The list is ordered by these time values; the most imminent time comes first, the most distant comes last.

The listhead for this queue resides at KER$GQ_TIME_QUEUE. The listhead is actually the first eight bytes of a dummy timer WCB. (The structure of the WCB is described fully in Section 11.1.1.) Several fields in the WCB relate to its use in the timer queue. The fields WCB$A_WAIT_FLINK and WCB$A_WAIT_BLINK allow the WCB to be linked into the queue. The field WCB$Q_TIME contains the time value representing the system time at which the wait expires. The second bit in the WCB$B_WAIT field (WCB$V_WAIT_DELTA) signifies

how the timed wait was specified to the KER$WAIT procedure. If the bit is set, then the wait time was specified as a interval (relative or delta) time. If the bit is clear, an absolute time was specified. The setting of the WCB$V_WAIT_DELTA bit allows the KER$SET_TIME procedure to adjust only the interval wait times of processes if the system time is reset.

As shown in Figure 5–2, the dummy WCB located at KER$GQ_TIME_QUEUE acts as the first WCB in the timer queue, and its forward and backward links comprise the actual listhead for the queue. Its WCB$Q_TIME field is permanently set to –1. This value is used by the KER$WAIT procedure for comparison when it inserts timer WCBs into the queue. Because real timer WCBs contain absolute time values (that is, positive values), they are inserted into the queue following the dummy WCB. The negative value of WCB$Q_TIME in the dummy WCB also allows both the hardware and software timer ISRs to determine quickly that the timer queue is empty.

If the timer queue is not empty, the interval timer ISR, described in Section 5.3.4, checks to see whether the time value in the first real WCB in the queue is less than or equal to the value of the system time. If so, the ISR requests a software interrupt to awaken the software timer ISR, described in Section 5.3.5. When it runs, this ISR walks the timer queue and unblocks every process whose wait has expired.

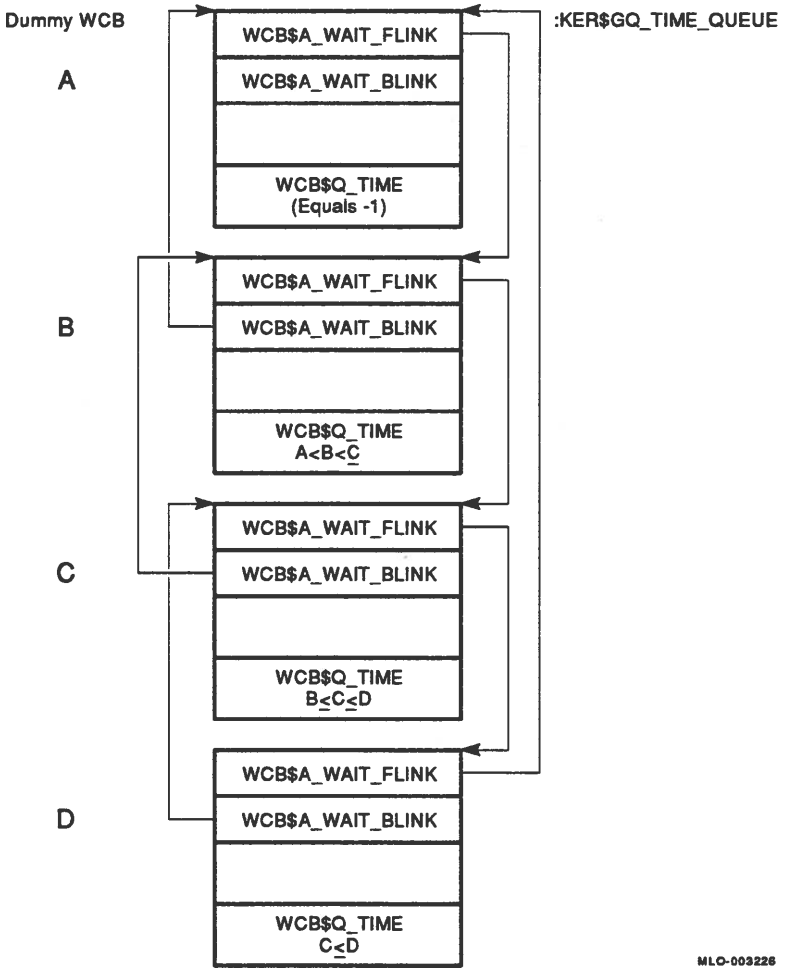
5.3.4 Interval Clock Interrupt Service Routine

The interval clock interrupt service routine, KER$HARDWARE_TIMER in module TIMERINT, services the hardware interrupt generated by the interval clock when the time defined in KER$GL_TIME_INTERVAL expires. On VAX–11/725, 730, and 750 targets this is an IPL 24 interrupt; on others, it is an IPL 22 interrupt.

The interval clock ISR has three major functions:

- Updating the system time
- Performing process and idle-time accounting
- Checking the timer queue for expired process waits

Figure 5–2: Timer Queue



On tightly coupled symmetric multiprocessing systems, this interrupt routine also maintains "watchdog timers," which allow one processor to monitor the status of another processor in the system. If a clock tick has expired and the watched processor has not updated a counter, a fatal system bugcheck is taken to bring down the entire system.

On all systems, the task of maintaining the system time and checking the timer queue belongs to the single or primary processor alone. KER$HARDWARE_TIMER executes as follows:

1. The interval clock CSR, PR$_ICCS, is reset to indicate that the interrupt has been serviced and to reenable the timer.
2. The value of KER$GQ_SYSTEM_TIME is updated by adding to it the value of KER$GL_TIME_INTERVAL.
3. The address of the first entry in the timer queue is obtained. If the queue is empty, this is the address of the dummy timer WCB.
4. The updated system time is compared to the time value in the first timer WCB.

   If the system time is less than or equal to the time in the WCB, then that first element has expired (this is never the case when the queue is empty), and a software interrupt is requested by writing IPL$K_TIMER (7) to PR$_SIRR. When IPL drops below IPL$K_TIMER, the software timer ISR runs to service the timer queue.
5. The interrupt-stack field in the PSL at the time of the interrupt is checked. If it is set, meaning that the system was executing outside of process context, the clock interval is charged against the processor's idle time by adding KER$GL_TIME_INTERVAL to KER$GQ_IDLE_TIME. (On tightly coupled symmetric multiprocessing systems, KER$GQ_IDLE_TIME is an array indexed by the processor number; therefore, KER$GL_TIME_INTERVAL is added to the appropriate element of KER$GQ_IDLE_TIME.)

   If the processor was in process context at the time of the interrupt, the clock tick is charged against the interrupted process by incrementing the value of PCB$L_CPU_TIME in the current PCB.
6. The interrupt is dismissed with the REI instruction.

5.3.5 Software Timer Interrupt Service Routine

The software time interrupt service routine, KER$SOFTWARE_TIMER in module TIMERINT, is invoked through the IPL$K_TIMER software interrupt. The software timer interrupt is requested by the interval clock hardware ISR when it finds that the entry in the timer queue has expired.

KER$SOFTWARE_TIMER services the timer queue as follows:

1. The LOCK macro is executed to ensure exclusive access to the timer queue and other system data.
2. The address of the first WCB in the timer queue is obtained. If the queue is empty, this is the address of the dummy timer WCB.
3. The system time, KER$GQ_SYSTEM_TIME, is compared to the value of WCB$Q_TIME in the timer WCB.

   If the system time is greater than or equal to the time in the WCB, the internal subroutine KER$UNWAIT is called to unblock the process that owns the expired WCB. KER$UNWAIT removes the WCB from the head of the timer queue and places the associated process into the ready state, regardless of the state of other wait conditions it may have specified. If the waiting process had specified a wait result variable, it is set to 0 to signify that the wait has timed out.
4. If the WCB has not expired, the interrupt is dismissed. Otherwise, control loops to step 2 to test the next queue entry. The next entry will always be at the head of the queue, because KER$UNWAIT removes the expired entries from the head of the queue.

The role of KER$WAIT in placing WCBs in the timer queue is described in Section 11.2, and the function of KER$UNWAIT is described in Section 11.3.3.3.

5.3.6 Time-Related Kernel Procedures

The kernel provides three procedures to allow callers to set the system time and to obtain the system time and the time that the system has been operating (uptime). The following sections describe these procedures, KERSET_TIME, KERGET_TIME, and KER$GET_UPTIME, all of which reside in module TIME.

The KER$WAIT_ANY and KER$WAIT_ALL procedures allow a process to wait until a specified date and time or for a specified interval. These procedures are described in Chapter 11.

5.3.6.1 KER$SET_TIME

The procedure KER$SET_TIME executes in kernel mode and allows its caller to replace the value of KER$GQ_SYSTEM_TIME with a specified 64-bit time value. The caller can create the binary time value before calling KER$SET_TIME by using the run-time library function ELN$TIME_VALUE to convert an ASCII time string. KER$SET_TIME is also responsible for adjusting the time values in the WCBs in the timer queue if they specify a wait for an interval time.

KER$SET_TIME expects two arguments: the address of an optional status value and the address of a 64-bit variable containing the new system time. The procedure executes as follows:

1. The current system time, KER$GQ_SYSTEM_TIME, is subtracted from the time specified by the caller, and the difference is saved.

   The difference between the specified time and the current time is used to update the wait time values in timer WCBs that were specified as relative waits.

2. The time difference is negated and added to the quadword KER$GQ_CLOCK_OFFSET. Before the correction to the system time is added to KER$GQ_CLOCK_OFFSET, the LOCK macro is executed to ensure the procedure's exclusive access to system time values and the timer queue. (The corrections to the system time are accumulated in KER$GQ_CLOCK_OFFSET so that the KER$GET_UPTIME procedure can calculate the system's elapsed time based on the current time; see Section 5.3.6.3.)
3. The caller's time value is copied to KER$GQ_SYSTEM_TIME, effectively resetting the system time to the new value.
4. The caller's time value is copied to KER$GQ_START_TIME, recording the time at which the system time was last set.
5. The timer queue is scanned for timer WCBs with the WCB$V_WAIT_DELTA bit set in WCB$B_WAIT. For each one, the time difference calculated earlier is added to the time value in WCB$Q_TIME.
6. The timer queue is reordered to correct any discrepancies created by the adjustments to the timer WCB time values.
7. IPL is restored, the spinlock is released, and KER$_SUCCESS status is returned to the caller.

The time values in timer WCBs are adjusted so that changes in system time do not affect the expiration times of interval waits. Adjustment is ultimately required because the values for timed waits — both absolute or relative — are stored as absolute system times. For relative waits, this absolute time is calculated in the KER$WAIT procedure by adding the specified relative time to the current system time. The result is entered into the WCB$Q_TIME field of the timer WCB, which is then inserted into the appropriate position in the timer queue. KER$SET_TIME, however, makes no adjustments to waits specified as absolute times. These waits are left to expire at their specified system times.

For example, if a process enters a five-minute wait one minute after the system is booted, but before the time is reset from the base time (the time would be 17–NOV–1858 00:01:00.00), its absolute expiration time would be recorded in WCB$Q_TIME as 17–NOV–1858 00:06:00.00, that is, five minutes after the time at which the wait was requested. If, one minute later, another process sets the system time to a more relevant value, such as 15–OCT–1989 08:20:00.00, the time value for the first process must be adjusted to preserve the remaining portion of its wait. Otherwise, its wait would expire immediately at the next clock tick, not in the expected four minutes.

KER$SET_TIME calculates the necessary adjustment by subtracting the value of KER$GQ_SYSTEM_TIME at the time of the call from the new system time specified, yielding, in this case, a difference of well over a century. Adding this value to the absolute time in WCB$Q_TIME would result in a new absolute expiration time of 08:24:00.00 15-OCT-1989, showing that four minutes remain in the five-minute wait.

This adjustment mechanism works the same way with less extreme adjustments to the system time. The adjustment preserves the relationship of an interval wait's expiration time to the new system time. If the system time were to be set back by five minutes, the subtraction from KER$GQ_SYSTEM_TIME would yield a difference of negative five minutes. Adding this negative offset to the values in the relative timer WCBs would adjust their timeout values back by the required five minutes.

5.3.6.2 KER$GET_TIME

The procedure KER$GET_TIME executes in the caller's mode and returns the 64-bit value KER$GQ_SYSTEM_TIME. The caller can then use the run-time library routine ELN$TIME_STRING to convert the binary time value to an ASCII string.

KER$GET_TIME expects two arguments: the address of an optional status value and the address of a 64-bit variable to receive the system time. The procedure executes by simply copying the value of KER$GQ_SYSTEM_TIME to the caller's quadword variable.

The status returned by the procedure depends on the setting of the low bit in the global value KER$GB_TIME_SET. If the bit is set, then the system time has been set and KER$_SUCCESS is returned. If The bit is clear, then system time has not been set. To warn the caller of this, the alternate success status KER$_TIME_NOT_SET is returned.

5.3.6.3 KER$GET_UPTIME

The procedure KER$GET_UPTIME executes in kernel mode and returns a 64-bit interval time value representing the time since the system was booted. The caller can then use the run-time library routine ELN$TIME_STRING to convert the binary time value to an ASCII string.

KER$GET_UPTIME expects two arguments: the address of an optional status value and the address of a 64-bit variable to receive the system uptime. Its operation depends on the value of KER$GQ_CLOCK_OFFSET, which is updated by each call to the KER$SET_TIME procedure to keep a running tally of adjustments to the system time.

To calculate the system uptime, KER$GET_UPTIME executes as follows:

1. The values of KER$GQ_SYSTEM_TIME and KER$GQ_CLOCK_OFFSET are added, yielding the actual uptime for the system.
2. The uptime value is negated and returned to the caller. Negating the uptime value transforms it to interval time format. This negative value can then be passed to the run-time library function ELN$TIME_STRING.

If the system time has not been set from the base time, represented in KER$GQ_SYSTEM_TIME as 0, KER$GQ_CLOCK_OFFSET will always be 0. Adding the offset to the system time, then, will simply yield the number of 100-nanosecond intervals that have passed since the system was initialized.

If the system time has been reset from the base time, KER$GQ_CLOCK_OFFSET contains the negative value of all the adjustments made to the system time. By adding this negative value to the current system time, KER$GET_UPTIME effectively rolls back any changes to the system time to yield the number of 100-nanosecond intervals that have transpired since the system was initialized.

Chapter 6

Condition Handling

Conditions are errors or anomalies detected by hardware and software and reported by the system software to a condition handler defined by the user. The VAXELN Kernel and its counterpart under the VMS operating system both employ the VAX condition-handling facility as set out in the VAX Procedure Calling and Condition Handling Standard. There are differences between the two implementations, but their functions parallel each other closely.

In particular, the VAX condition-handling facility defines the following aspects of condition handling:

- How a condition handler is declared and canceled
- How a condition handler is located and invoked
- How a condition handler may respond to a condition

The facility also provides that the same condition handlers be called in response to conditions detected by both hardware and software. This mechanism, called uniform condition dispatching, allows application software to centralize its handling of all errors in a single condition-handling procedure.

VAXELN extends this condition-dispatching mechanism to include software-generated conditions called asynchronous exceptions. An asynchronous exception allows one process to interrupt another asynchronously to the latter process's execution. The VAX hardware's support for asynchronous system traps (ASTs) and the VAXELN Kernel's condition-dispatching mechanism work together to deliver asynchronous exceptions.

The following sections describe the implementation of the VAX condition-handling facility within the VAXELN Kernel:

- Section 6.1 defines VAX conditions.
- Section 6.2 describes the data structures involved in dispatching and handling conditions.
- Section 6.3 describes how conditions detected by hardware are prepared for dispatching.
- Section 6.4 describes how conditions detected by software are prepared for dispatching.
- Section 6.5 describes how asynchronous exception conditions are generated and prepared for dispatching.
- Section 6.6 describes the uniform condition-dispatching mechanism; namely, how the kernel locates and calls a condition handler and how it deals with handled and unhandled conditions.
- Section 6.7 describes the options available to a condition handler, including unwinding the stack to change the flow of execution.

6.1 Conditions Detected by Hardware and Software

Conditions fall into two categories, based on whether the condition is detected by the hardware/microcode or by software:

- Exception conditions. An exception condition is the processor's response to an anomaly or error it encounters while executing an instruction. The hardware/microcode responds by changing the flow of execution, directing it to an exception service routine. Each exception routine is identified by its own longword vector, defined by the VAX architecture, in the SCB.

  An exception condition may be classified as a trap, a fault, or an abort:

  - A trap is an exception condition that occurs at the end of the instruction that causes the error. The PC pushed onto the stack by the hardware/microcode exception sequence is the address of the instruction following the offending instruction. Example: integer divide by zero.

- A fault is an exception condition that occurs during an instruction and leaves registers and memory in a consistent state. The PC pushed onto the stack during the exception sequence is the address of the instruction that caused the error, so that eliminating the fault and restarting the instruction is possible. Example: access control violation.
- An abort is an exception condition that occurs during an instruction and potentially leaves registers and memory in an inconsistent state, so that the instruction cannot be safely restarted, completed, or undone. Example: kernel stack not valid.

These hardware-detected errors occur synchronously, that is, at predictable points in the flow of process execution. The VAXELN Kernel recognizes another class of exception, called asynchronous exception conditions. Asynchronous exception conditions are generated outside the flow of process execution. These exceptions are in fact requested by software and are delivered asynchronously to a process through the cooperation of the kernel and the VAX hardware's support for ASTs, as described in Section 6.5.

- Software conditions. A software condition is an error or anomaly detected by the system or a user program and treated in a particular way. When software detects such an error, it transforms it into a software condition by calling the kernel procedure KER$RAISE_EXCEPTION. (Programs can also call the run-time library VMS-emulation routines LIB$SIGNAL and LIB$STOP, which then call KER$RAISE_EXCEPTION.)

  The description of the kernel's processing of software conditions begins with Section 6.4.

6.2 Data Structures for Condition Handling

The structures central to the VAX condition-handling facility are defined by the VAX Procedure Calling and Condition Handling Standard. This section describes the following structures:

- The call frame. This structure, built by the CALLG and CALLS instructions, records information about a called procedure that allows the kernel to trace the path of procedure calls down a process's stack.

- The condition-handler argument list. This structure, built by the kernel, is passed to a condition handler to enable it to locate the signal and mechanism arrays, which contain information about the condition.
- The signal array. This structure, built by the kernel, is passed to a condition handler and describes the condition — signal — that resulted in the calling of the handler.
- The mechanism array. This structure, built by the kernel, is passed to a condition handler and records information about the kernel's search for a condition handler.

This set of data structures applies to the VAX condition-handling facility as a whole. Another set of data structures exists to support the delivery of asynchronous exception conditions to a process. These structures are described separately in Section 6.5.1.

6.2.1 Call Frames

A VAX call frame saves information about the caller's state to be restored when the called procedure completes. The call frame also contains information that allows the kernel to trace the calling hierarchy down the stack.

Figure 6–1 shows the structure of the VAX call frame created on the stack by the CALLG and CALLS instructions. The figure also shows the location of the argument list transmitted on the stack by the CALLS instruction. Table 6–1 describes the fields in the frame. The symbols in the second column are defined in the $SFDEF macro in the VMS STARLET macro library.

The called procedure's local stack environment begins on top of its call frame. The value of the frame pointer (FP) points to the first longword at the top of the call frame. When an exception or software condition occurs, the kernel's condition-dispatching logic searches down the stack, through successive call frames, until it locates a frame that has established a condition handler by storing its address in that first longword in the call frame. The kernel is able to locate lower frames by using the saved FP value in one call frame to locate the top of the next lowest frame on the stack.

Figure 6–1: VAX Call Frame for CALLG and CALLS

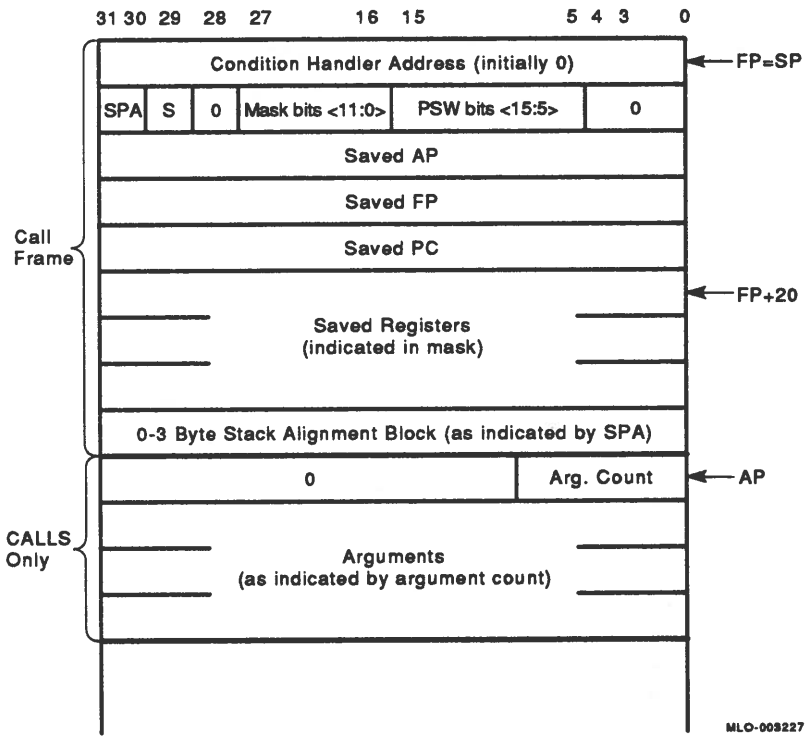


Table 6–1: Structure of a VAX Call Frame

Field	Symbols	Meaning
Condition handler address	SF$A_HANDLER	The address of the condition handler established for the frame. The CALL instructions set this field to 0 and the frame pointer to point to this longword. A condition handler is established when its address is written to this field.
SPA (stack pointer alignment) bits	SF$V_STACKOFFS SF$S_STACKOFFS	The low two bits of the stack pointer (SP) value at the time of the CALL. This value is subtracted from SP by the CALL instructions to align the stack on a longword boundary.
S bit	SF$V_CALLS SF$S_CALLS	The bit that indicates how the call frame was created. The bit is set if CALLS was executed; the bit is clear if CALLG was executed. If the S bit is set, the RET instruction clears the CALLS argument list from the stack.
Mask bits	SF$W_SAVE_MASK SF$V_SAVE_MASK SF$S_SAVE_MASK	The lowest twelve bits of the procedure entry mask, indicating which registers were saved on the stack on entry to the procedure.
PSW bits	SF$W_SAVE_PSW	The high 11 bits of the processor status word (PSW), indicating the settings of the integer overflow (IV), floating underflow (FU), and decimal overflow (DV) trap-enable bits at the time of the CALL.
Saved AP	SF$L_SAVE_AP	The value of the argument pointer at the time of the CALL.
Saved FP	SF$L_SAVE_FP	The value of the frame pointer at the time of the call. This value defines the calling procedure's stack environment and is used by the condition-dispatching logic to scan back through frames on the stack.
Saved PC	SF$L_SAVE_PC	The value of the program counter at the time of the CALL. This is the address of the instruction following the CALLS or CALLG instruction to which control will return when the called procedure returns.

Table 6–1 (Cont.): Structure of a VAX Call Frame

Field	Symbols	Meaning
Saved registers	SF$L_SAVE_REGS	The values of the registers specified in the procedure's entry mask. Procedures compliant with the VAX Procedure Calling and Condition Handling Standard save only registers R2 through R11. Registers R0 and R1 are for the return of function values.
Stack alignment block	None	Zero to three bytes added to the stack to align it on a longword boundary. The number of bytes here matches the value of the SPA bits.

6.2.2 Condition-Handler Argument List

When the kernel calls a condition handler, it passes two arguments in a standard VAX argument list: the address of the signal array and the address of the mechanism array. These longword arrays, described in Sections 6.2.3 and 6.2.4, respectively, give the handler information about the nature of the condition and the number of call frames that have already been searched for a handler.

Figure 6–2 shows the structure of the argument list, and Table 6–2 describes its fields. The symbols listed in the second column are defined in the $CHFDEF macro in STARLET.MLB.

Figure 6–2: Condition-Handler Argument List

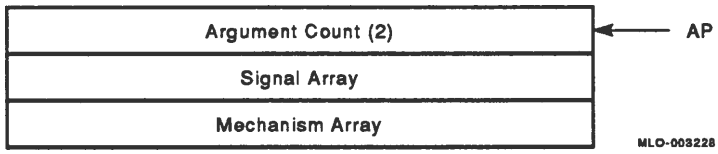


Table 6–2: Structure of the Condition-Handler Argument List

Field	Symbol	Meaning
Argument count	None	The number of arguments passed to the handler (always 2)
Signal array	CHF$L_SIGARGLST	The address of the signal array
Mechanism array	CHF$L_MCHARGLST	The address of the mechanism array

6.2.3 Signal Arrays

A condition handler examines the signal array to determine what condition — signal — caused it to be invoked. For exception conditions, part of the array is created by the hardware/microcode, and the rest is created within the service routine specific to the exception (see Section 6.3); for software conditions, the array is created by the KER$RAISE_EXCEPTION kernel procedure (see Section 6.4); and for asynchronous exception conditions, the array is created by the KER$AST_INTERRUPT IPL 2 software interrupt service routine (see Section 6.5).

Figure 6–3 shows the structure of the VAX signal array, and Table 6–3 describes the information it conveys to the condition handler. The symbols in the table's second column are defined in the $CHFDEF macro in STARLET.MLB.

Figure 6–3: Signal Array

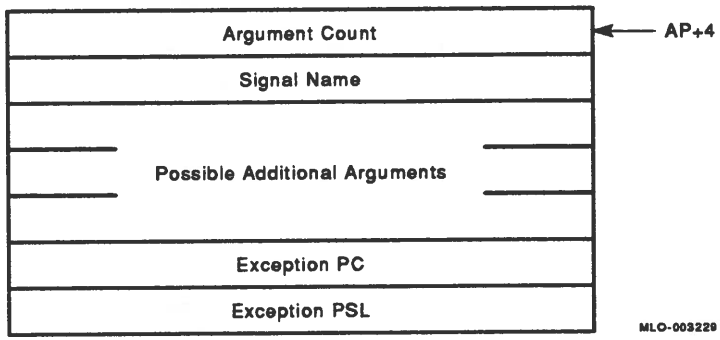


Table 6–3: Structure of the Signal Array

Field	Symbol	Meaning
Argument count	CHF$L_SIG_ARGS	The total number of condition arguments passed in the signal array. The minimum number is 3 — the signal name, the exception PC, and the exception PSL.
Signal name	CHF$L_SIG_NAME	A unique bit configuration that identifies the exception or software condition. The control, facility number, message number, and severity fields in the signal name are defined in the VAX Procedure Calling and Condition Handling Standard.
Additional arguments	None	Zero or more additional arguments that pass additional information about the exception or software condition. The number of additional arguments in the array is determined by subtracting 3 from the argument count.
Exception PC	None	The value of the program counter at the time the condition was signaled. For traps and software conditions, this value is the address of the instruction following the one that generated the condition. For faults, this value is the address of the instruction that generated the condition.
Exception PSL	None	The value of the processor status longword at the time condition was signaled. The condition-dispatching logic can use this value to determine the access mode of the process that generated the condition.

6.2.4 Mechanism Arrays

A condition handler can determine the circumstances of its calling by examining the mechanism array. Information in this array records how many call frames were searched for handlers before the current handler was called and identifies the frame (by FP value) that established the handler. These data are useful when the handler wants to unwind the stack to the frame that established it (see Section 6.7.2). For both exception and software conditions, the mechanism array is created

by the kernel's condition-dispatching logic (in module EXCEPTION). Section 6.6.1 describes this process.

Figure 6–4 shows the structure of the VAX mechanism array, and Table 6–4 describes the information it conveys to the condition handler. The symbols in the table's second column are defined in the $CHFDEF macro in STARLET.MLB.

Figure 6–4: Mechanism Array

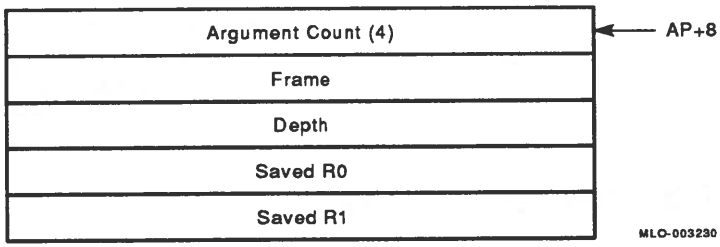


Table 6–4: Structure of the Mechanism Array

Field	Symbol	Meaning
Argument count	CHF$L_MCH_ARGS	The number of arguments in the mechanism array. This value is always 4.
Frame	CHF$L_MCH_ FRAME	The value of the frame pointer that defines the stack environment for the procedure that established the called condition handler. The handler can use this value to unwind the stack to the frame that established it.
Depth	CHF$L_MCH_ DEPTH	The number of call frames searched to locate the current condition handler. The handler can use this value to unwind the stack to the frame that established it.

Table 6–4 (Cont.): Structure of the Mechanism Array

Field	Symbol	Meaning
Saved R0	CHF$L_MCH_ SAVR0	The value of R0 at the time the condition was signaled. Since the VAX calling standard prohibits saving R0 on entry into a procedure, the condition-dispatching logic must save that value here. It is restored when execution continues after the condition is handled. A handler can return information to the procedure that raised the condition by altering the contents of this field.
Saved R1	CHF$L_MCH_ SAVR1	The value of R1 at the time the condition was signaled. As with R0, the calling standard prohibits the saving of R1.

6.3 Exception Conditions

When a synchronous exception is detected by the hardware/microcode, the processor and the kernel cooperate to build the signal array on the appropriate stack and transfer control to the uniform condition-dispatching logic to complete processing the condition. Section 6.3.1 describes the processor's role in creating the signal array, and Section 6.3.2 describes the role played by the kernel.

6.3.1 Initial Processor Actions

When the hardware/microcode detects an exception, the processor responds by taking the following actions:

1. Depending on the exception, access mode and stack may be changed. In general, the processor uses the settings of the low two bits of the exception's SCB vector to determine on which stack the exception is serviced. Table 6–5 summarizes the stack choices resulting from this architectural mechanism and the VAXELN SCB's definitions of exception vectors.
2. The exception PC is pushed onto the stack.
3. The exception PSL is pushed onto the stack.
4. Exception-specific parameters may be pushed onto the stack.

5. Control is transferred to the service routine whose address is stored in the SCB vector for the exception. The service routine then finishes constructing the signal array and transfers control to the routine that dispatches the exception, as described in Section 6.3.2. Table 6–6 shows the exceptions defined by the VAX architecture, their byte offsets in the SCB, their types, the number of their additional parameters, and the names of their service routines.

Table 6–5: Selection of Exception Stack

Exception	Kernel or User Mode?	On Interrupt Stack?	Exception Stack
Machine check	K or U	N/A	Interrupt
Kernel stack not valid	K	No	Interrupt
Subset emulation	K or U	N/A	Same
Change mode to kernel	K or U	No	Kernel
Change mode to kernel	K	Yes	Processor halt
All others	K or U	No	Kernel
All others	K	Yes	Interrupt

Table 6–6: VAX Exception Vectors Under VAXELN

Exception	Vector (Hex.)	Type	Extra Params.	Service Routine/Remarks
Machine check	4	Abort/Fault	0	Handling based on processor type, as described in Section 7.3. Process-level, nonfatal machine checks are reflected back to the service routine KER$MCHECK in module EXCEPTION.
Kernel stack not valid	8	Abort	0	KER$KERNEL_STACK in module EXCEPTION. See Section 6.3.2.3.
Reserved/privileged instruction	10	Fault	0	KER$DIGITAL_RESERVED in module EXCEPTION. This service routine also serves as the entry point for bugchecks, and this vector is also used as the entry point to the VAX floating-point emulation routines. See Section 6.3.2.4.

Table 6–6 (Cont.): VAX Exception Vectors Under VAXELN

Exception	Vector (Hex.)	Type	Extra Params.	Service Routine/Remarks
Customer reserved instruction	14	Fault	0	KER$CUSTOMER_RESERVED in module EXCEPTION.
Reserved operand	18	Abort/Fault	0	KER$RESERVED_OPERAND in module EXCEPTION.
Reserved addressing mode	1C	Fault	0	KER$RESERVED_MODE in module EXCEPTION.
Access violation	20	Fault	2	KER$ACCESS_VIOLATION in module EXCEPTION. This service routine also handles the dynamic expansion of the user stack. See Section 6.3.2.1.
Translation not valid	24	Fault	2	KER$ACCESS_VIOLATION in module EXCEPTION. Since the kernel does not page, this fault is reflected as an access violation.
Trace pending	28	Fault	0	KER$TRACE_TRAP in module EXCEPTION.
Breakpoint instruction	2C	Fault	0	KER$BREAKPOINT in module EXCEPTION.
Compatibility mode	30	Abort/Fault	1	KER$COMPATIBILITY in module EXCEPTION.
Arithmetic	34	Fault/Trap	1	KER$ARITHMETIC in module EXCEPTION. See Section 6.3.2.2.
Change mode to kernel	40	Trap	1	KER$KERNEL_SERVICE in module DISPATCH. See Section 8.2.
Change mode to executive	44	Trap	1	KER$CHANGE_EXECUTIVE in module EXCEPTION. This access mode is not supported under VAXELN.
Change mode to supervisor	48	Trap	1	KER$CHANGE_SUPERVISOR in module EXCEPTION. This access mode is not supported under VAXELN.
Change mode to user	4C	Trap	1	KER$CHANGE_USER in module EXCEPTION.

Table 6–6 (Cont.): VAX Exception Vectors Under VAXELN

Exception	Vector (Hex.)	Type	Extra Params.	Service Routine/Remarks
Emulation start	C8	Trap	10	VAX$EMULATE in module [EMULAT]VAXEMULATE. This vector is for string-instruction emulation.
Emulation continue	CC	Trap	0	KER$EMULATE_FPD in module [EMULAT]VAXEMULATE. The vector for continuing an interrupted emulated string instruction.

6.3.2 Initial Kernel Actions

Once the processor has transferred control to the appropriate exception service routine, that routine takes the following steps to finish building the signal array to allow processing to continue in the common dispatch code:

1. The exception (signal) name is pushed onto the stack, for example, SS$_ACCVIO for an access control violation.
2. The total argument count for the signal array is pushed onto the stack. For example, an access control violation has an argument count of 5: the exception PC, the exception PSL, two additional parameters, and the signal name.
3. Control is transferred to the global entry point KER$REFLECT in module EXCEPTION, the start of the uniform condition-dispatching mechanism. Section 6.6 describes the function of KER$REFLECT.

Table 6–7 shows the signal name pushed onto the stack for each exception serviced by a routine in module EXCEPTION. The table also shows the final number of arguments in the signal array. The following sections, as noted in the table, describe the exceptions that require additional processing before their service routines pass control to the dispatch code.

Table 6–7: Exceptions Serviced by Module EXCEPTION

Exception	Signal Name	Signal Array Size	Remarks
Access-control violation	SS$_ACCVIO	5	See Section 6.3.2.1.
Arithmetic exception	Varies	3	See Section 6.3.2.2.
Breakpoint instruction	SS$_BREAK	3	
Change mode to user	SS$_CMODUSER	4	The additional parameter is the change mode code.
Compatibility mode	SS$_COMPAT	4	The additional parameter is the compatibility exception code.
Customer reserved instruction	SS$_OPCCUS		
Kernel stack not valid	KER$_KERNEL_STACK	3	See Section 6.3.2.3.
Machine check	KER$_MACHINECHK	3	Machine-check exceptions reported to a process have no extra parameters in the signal array. The machine-check parameters have been examined, possibly written to the error log, and discarded by the machine-check handler.
Reserved/privileged instruction	SS$_OPCDEC	3	See Section 6.3.2.4.
Reserved addressing mode	SS$_RADRMOD	3	
Reserved operand	SS$_ROPRAND	3	
Trace pending	SS$_TBIT	3	

6.3.2.1 Access Control Violation Exceptions

The processor transfers control to the service routine KER$ACCESS_VIOLATION in the following circumstances:

- The memory management hardware/microcode detects an architecturally defined memory access control violation, such as attempting to access privileged or nonexistent virtual memory.

- A translation-not-valid fault occurs (V bit clear in a PTE). Under VMS, this exception is the entry into the paging subsystem. Since VAXELN does not page, this exception is reflected to the process as a normal access violation. VAXELN memory management always sets the V bit when constructing page table entries, so this fault is probably the result of a kernel-mode program corrupting a page table.

In response to an access violation fault, the processor pushes two extra exception parameters onto the stack: the inaccessible virtual address and a bit mask describing the reason for the fault. The exception service routine KER$ACCESS_VIOLATION first examines the PSL to determine whether the faulting process was in user mode. If so, the routine first assumes that the fault resulted from user-stack overflow and passes the virtual address parameter to the local subroutine KER$EXPAND_STACK to try to expand the stack. If the address argument is a P1 virtual address, KER$EXPAND_STACK calls KER$ALLOCATE_MEMORY to expand the user stack. If the expansion fails, or the address is not in P1 address space, the subroutine returns failure status.

If KER$EXPAND_STACK fails, the service routine pushes the signal name SS$_ACCVIO and the signal array argument count (5) onto the stack and transfers control to the dispatch code.

6.3.2.2 Arithmetic Exceptions

Seven possible arithmetic exceptions can occur on processors supported by VAXELN. For each exception, the processor pushes a unique exception parameter onto the stack. The service routine KER$ARITHMETIC, in module EXCEPTION, does not then simply push a generic exception name onto the stack. Instead, the exception parameter is used to create a unique signal name with the following formula:

$$signalname = (8 * code) + \text{SS\$\_ARTRES}$$

Table 6–8 shows the arithmetic exception names, their codes, and their symbolic representations.

Table 6–8: Signal Names for Arithmetic Exceptions

Exception	Code	Type	Symbol
Integer overflow	1	Trap	SS$_INTOVF
Integer divide by zero	2	Trap	SS$_INTDIV
Decimal overflow	6	Trap	SS$_DECOVF
Subscript range	7	Trap	SS$_SUBRNG
Floating overflow	8	Fault	SS$_FLTOVF_F
Floating divide by zero	9	Fault	SS$_FLTDIV_F
Floating underflow	10	Fault	SS$_FLTUND_F

6.3.2.3 Kernel-Stack-Not-Valid Exceptions

A kernel-stack-not-valid exception indicates that the kernel stack was not valid while the processor was pushing information onto the stack while initiating an exception or interrupt. This exception is serviced by the KER$KERNEL_STACK routine on the interrupt stack at IPL 31.

The service routine first determines whether the base of the kernel stack is actually accessible. If not, a fatal KRNLSTAKNV bugcheck is initiated. If so, the kernel stack pointer is reset to the original base of the kernel stack, ensuring that at least one page of stack space is available. KER$KERNEL_STACK then takes the following steps to report the exception:

1. The exception PC and PSL are copied from the interrupt stack to the reset kernel stack.
2. The KER$_KERNEL_STACK signal name is pushed onto the stack.
3. The signal array argument count (3) is pushed on the stack to complete the signal array.
4. The frame pointer is cleared, making the calling hierarchy untraceable and eliminating the possibility that the exception will be reported to a condition handler established by the user.
5. Control is transferred to the dispatch code to complete the processing of the exception.

6.3.2.4 Reserved Instruction Exceptions

A reserved/privileged instruction exception can indicate an attempt to execute an opcode not supported by the processor. This can occur, for example, when a floating-point instruction is attempted on a processor without microcode for that type of floating-point format. Software emulation of floating-point instructions is invoked through a service routine for this exception. (If floating-point emulation is enabled, the service routine — VAX$OPCDEC in module [EMULAT]FPEMULATE — tests to determine whether the reserved opcode indicates an emulated floating-point instruction. If not, it transfers control to the service routine KER$DIGITAL_RESERVED.)

KER$DIGITAL_RESERVED first checks whether the reserved opcode is FF_{16}, followed by FE_{16} or FD_{16}. This sequence is used by the kernel to initiate bugchecks. If the sequence matches, KER$DIGITAL_RESERVED transfers control to KER$BUG_CHECK (in module BUGCHECK) to process the bugcheck.

If a bugcheck is not indicated, the service routine pushes the signal name SS$_OPCDEC onto the stack, completes the signal array by pushing the argument count (3) onto the stack and transferring control to the dispatch code.

6.4 Software Conditions

The kernel procedure KER$RAISE_EXCEPTION (in module RAISE) gives software a way to raise a signal in response to an error. The condition sequence initiated by calling KER$RAISE_EXCEPTION parallels the sequence generated by the hardware.

The vectors for most kernel procedures call KER$RAISE_EXCEPTION to signal an error condition when the failed procedure's caller has not specified a status variable. This process is described in Section 8.4.2. KER$RAISE_EXCEPTION is also available to user programs to report any irregularities the program encounters during execution.

The primary function of KER$RAISE_EXCEPTION is to build a signal array on the caller's stack and pass control to the common dispatch code, which then delivers the signal to the caller's process in the caller's mode. The signal array constructed by KER$RAISE_EXCEPTION is identical to the array constructed for an exception condition. The signal name and additional arguments in the array are those passed as arguments to KER$RAISE_EXCEPTION.

KER$RAISE_EXCEPTION executes in the mode of its caller (see Section 8.3, Dispatch to Procedures That Execute in the Caller's Mode) and takes the following steps to create the signal array and dispatch the software condition:

1. The CALLS bit in the call frame for KER$RAISE_EXCEPTION is cleared. Clearing this bit means that the argument list (containing the signal name and any additional arguments) will not be removed from the stack when a RET instruction is executed.

   (The initial setting of the CALLS bit shows whether the procedure was called with a CALLS or CALLG instruction. The remainder of this description applies to the case of CALLS. Handling of the CALLG case differs slightly but the outcome is identical.)
2. The caller's saved PC and saved PSW are copied from the call frame for KER$RAISE_EXCEPTION and saved. These values become the exception PC and PSL values in the signal array.
3. The saved PC field in the call frame is replaced with an address within KER$RAISE_EXCEPTION.
4. A RET instruction is executed to clear the call frame from the stack. Execution continues within KER$RAISE_EXCEPTION but is now on the caller's frame.
5. The stack is manipulated to construct the signal array. This operation restores the exception PSL and PC from their saved locations, moves the signal name and additional arguments above them, and tops off the stack with the argument count for the array.
6. Control is transferred to the dispatch code, at KER$REFLECT, which will add the mechanism array to the stack and dispatch the software condition.

6.5 Asynchronous Exception Conditions

Asynchronous exceptions are a mechanism for signaling an asynchronous event to a process. As soon as possible after the asynchronous event occurs, the asynchronous signal is delivered to the process.

Under VAXELN, asynchronous exception conditions have a limited and specialized role, unlike their counterparts under the VMS operating system, asynchronous system traps (ASTs). Asynchronous exceptions have the following roles under VAXELN:

- To signal a process to force its exit

- To signal a process to gain its attention
- To notify a process that power-failure recovery is in progress
- To halt a process with the debugger

Asynchronous exceptions are delivered to a process through the uniform condition-dispatching mechanism described in Section 6.6. Each type of asynchronous exception has a kernel procedure dedicated to its delivery. Calling one of these procedures results in a call to the internal procedure KER$SIGNAL_AST, which posts the exception to the process through the cooperation of the hardware and software. The asynchronous exception is ultimately delivered to the target process through the IPL 2 interrupt service routine KER$AST_INTERRUPT.

Because the asynchronous exception is delivered in the same way as any other synchronous exception or software condition, a process must respond to the asynchronous event with special handling in a condition handler. For example, it can respond to a forced-exit signal, generated by KER$SIGNAL, by performing process-level cleanup, such as freeing jobwide resources (for example, removing messages from a port or unlocking a mutex).

Section 6.5.1 describes the software and hardware elements that cooperate to deliver asynchronous exceptions. Section 6.5.2 then describes the routines used to request asynchronous exceptions. Section 6.5.3 next describes the kernel routine KER$SIGNAL_AST, used to post an asynchronous exception to a process. Section 6.5.4 describes the AST delivery ISR, and, finally, Section 6.5.5 describes the kernel procedures that disable and reenable the delivery of asynchronous exceptions.

6.5.1 Data Structures and Hardware Features for Asynchronous Exceptions

The VAX hardware/microcode and several software data structures cooperate to deliver a requested asynchronous exception to a process. The hardware components are the REI instruction, the ASTLVL privileged register, and the IPL 2 software interrupt. The data structures that support asynchronous exceptions are the process control block and the process hardware context block. The following sections describe these components and data structures. Discussion of the IPL 2 interrupt, however, is deferred until Section 6.5.4.

6.5.1.1 REI Instruction

The REI (Return from Exception or Interrupt) instruction initiates the delivery of an asynchronous exception to a process by requesting an IPL 2 interrupt. The requested IPL 2 interrupt, however, is not in fact granted until processor IPL drops to user IPL.

The REI microcode requests the IPL 2 interrupt when the following conditions accompany the execution of the instruction:

1. Execution must be returning to process context. If the interrupt stack bit is set in the PSL (PSL<IS>) being restored, the REI microcode makes no further test and does not request the interrupt. The asynchronous exception must be delivered in process context and therefore cannot take place when execution is in system context.
2. The access mode of the process whose PSL is being restored must be as privileged or less privileged than the access mode for which an asynchronous exception is pending. In other words, the REI microcode checks the value of the ASTLVL privileged register. If its value is less than or equal to the current mode field in the PSL being restored, the interrupt is requested. This test prevents a user-mode process (access mode = 3) running temporarily in kernel mode (access mode = 0) from being interrupted to deliver an asynchronous exception. The delivery will occur only when the process returns to user mode through the subsequent execution of the REI instruction.

The role of the ASTLVL register in the REI tests is described in Section 6.5.1.2.

6.5.1.2 ASTLVL Register

The privileged processor register ASTLVL (PR$_ASTLVL) is used in conjunction with the REI instruction to control IPL 2 software interrupts. This register is part of the hardware context of a process and has a save area in the process hardware context block (PTX), described in Section 6.5.1.3. The LDPCTX (Load Process Context) instruction copies the ASTLVL value from the PTX save area to PR$_ASTLVL when a process is switched into execution.

Under VAXELN, two possible values in PR$_ASTLVL indicate to the REI microcode that an asynchronous exception is pending against the current process:

- The value 0 in PR$_ASTLVL indicates that an asynchronous exception is pending against a kernel-mode process.
- The value 3 in PR$_ASTLVL indicates that an asynchronous exception is pending against a user-mode process.

When no asynchronous exception is pending against a process, the value of PR$_ASTLVL is 4. No IPL 2 interrupt will be requested by the REI microcode, because no access mode can be greater than or equal to the ASTLVL of 4.

6.5.1.3 Hardware Context Block

When a process is switched into execution by the LDPCTX instruction, the value of the PR$_ASTLVL register is determined by the value of its save area in the process's PTX. This save area is in bits <26:24> of the PTX$L_P0_LIMIT field of the PTX.

When the kernel requests an asynchronous exception against a process that is not currently executing, it writes the base, or program (JCB$B_MODE), access mode of the target process (0 or 3) into the PTX$V_ASTLVL bit field in the PTX. When the process next becomes current, the restoration of PR$_ASTLVL from the PTX and the subsequent execution of REI cause the asynchronous exception to be delivered through the IPL 2 software interrupt.

When a process is created, the value of PTX$V_ASTLVL is set to 4, signifying that no asynchronous exception is pending against the process. Because the SVPCTX instruction does not save the value of PR$_ASTLVL in the PTX$V_ASTLVL, when the kernel changes PR$_ASTLVL it also changes PTX$V_ASTLVL. When an asynchronous exception has been delivered, the kernel returns the values of both PTX$V_ASTLVL and PR$_ASTLVL to 4.

6.5.1.4 Process Control Block

The kernel requests an asynchronous exception against a process by setting a specific bit in the process control block (PCB) of the process to receive the asynchronous exception.

The PCB$B_REASON field in the PCB contains the bits that represent the asynchronous exceptions pending against a process. When set, these bits have the following meanings:

- PCB$V_SIGNAL_DEBUG (bit 0) represents an asynchronous exception generated by a call to KER$RAISE_DEBUG_EXCEPTION.
- PCB$V_SIGNAL_POWER (bit 1) signifies an asynchronous exception generated by the power-failure recovery sequence.
- PCB$V_SIGNAL_QUIT (bit 2) signifies an asynchronous exception generated by a call to KER$SIGNAL.
- PCB$V_SIGNAL_PROCESS (bit 3) signifies an asynchronous exception generated by a call to KER$RAISE_PROCESS_EXCEPTION.
- PCB$V_SIGNAL_DISABLE (bit 4) signifies that a process has disabled the delivery of asynchronous exceptions by calling the kernel procedure KER$DISABLE_ASYNCH_EXCEPTION.

When the IPL 2 interrupt service routine executes, it tests the bits in PCB$B_REASON to determine the reason for the interrupt. The reason bits are tested in the following order:

1. PCB$V_SIGNAL_DISABLE
2. PCB$V_SIGNAL_POWER
3. PCB$V_SIGNAL_DEBUG
4. PCB$V_SIGNAL_QUIT
5. PCB$V_SIGNAL_PROCESS

If the disable bit is set, the scan completes, and the interrupt is dismissed. Otherwise, asynchronous exceptions are mutually exclusive: as soon as a set bit is discovered, it is cleared and the corresponding signal is delivered to the process. If, for example, both a debug and process-quit signal are pending against the process, the debug signal is delivered first. Only after that first asynchronous exception is delivered will the process-quit signal be delivered through a subsequent IPL 2 interrupt. (This process is described in Section 6.5.4.)

6.5.2 Uses of Asynchronous Exception Conditions

The kernel delivers the following asynchronous exceptions:

- Process signal. This asynchronous exception is delivered to a process specified in a call to KER$SIGNAL.
- Process attention signal. This asynchronous exception is delivered to a process specified in a call to KER$RAISE_PROCESS_EXCEPTION.
- Power failure. This asynchronous exception is delivered to processes during power failure recovery. The exception is delivered only when the process's job has **Powerfailure exception** set to *Yes* in its program description.
- Debugger HALT command. This asynchronous exception is delivered to a process specified in a debugger HALT command.

The first two applications of the asynchronous exception mechanism are available to users through the kernel procedures KER$SIGNAL and KER$RAISE_PROCESS_EXCEPTION. The remaining two are used internally by the kernel and debugger. All four are alike in that the events they represent must be delivered asynchronously to the normal execution of the target process, and all are delivered through the VAXELN uniform condition-dispatching mechanism described in Section 6.6.

The following sections describe these asynchronous exceptions.

6.5.2.1 Process Signal Exception

The asynchronous exception that signals a process is generated by a call to the KER$SIGNAL kernel procedure (in module SIGNAL), which specifies the object identifier for the process to receive the signal. The intention in signaling a process is to force its exit with the KER$_QUIT_SIGNAL status. If the process has established a condition handler that returns a continue status in response to the quit signal, the process will not exit.

KER$SIGNAL requests the quit signal against the target process by passing the appropriate bit number in PCB$B_REASON (PCB$V_SIGNAL_QUIT) to the internal subroutine KER$SIGNAL_AST, described in Section 6.5.3. KER$SIGNAL_AST then sets that bit in the target process's reason mask and adjusts ASTLVL values to request an IPL 2 interrupt the next time the target process executes.

6.5.2.2 Process Attention Signal Exception

The asynchronous exception that requests process attention is generated by a call to the KER$RAISE_PROCESS_EXCEPTION kernel procedure (in module RAISE), which specifies the object identifier for the process to receive the signal. The intention is to force the process to respond to the KER$_PROCESS_ATTENTION signal in a manner defined by the program. The process should have a condition handler that checks for this signal and takes a predefined action as a result. If the condition handler does not return a continue status, the process is forced to exit.

KER$RAISE_PROCESS_EXCEPTION requests the attention signal against the target process by passing the appropriate bit number in PCB$B_REASON (PCB$V_SIGNAL_PROCESS) to the internal subroutine KER$SIGNAL_AST.

6.5.2.3 Power-Failure Exception

The asynchronous exception that generates a power-failure signal is requested by the power failure-recovery routine KER$RESTART (in module POWERFAIL). During recovery, KER$RESTART scans the global job list for programs that have requested power-failure notification (the PRG$V_POWER_RECOVERY bit is set in PRG$B_OPTION_FLAGS).

When it locates such a program, KER$RESTART walks the list of PCBs for that job and, for each process, passes the appropriate bit number in PCB$B_REASON (PCB$V_SIGNAL_POWER) to the internal subroutine KER$SIGNAL_AST. When each process next executes, the KER$_POWER_FAIL signal will be delivered. The process should have a condition handler that checks for this signal and takes a power-recovery action as a result. If the condition handler does not return a continue status, it is forced to exit.

6.5.2.4 Debugger HALT Command

The asynchronous exception that results in the halting of a process under debugger control is generated by the debugger HALT command. When the debugger receives a HALT command, it calls the internal kernel procedure KER$RAISE_DEBUG_EXCEPTION (in module DEBUGUTIL), passing the specified job and process numbers as arguments.

KER$RAISE_DEBUG_EXCEPTION searches the global job list to locate the target job. It then searches that job's process list to locate the target process. KER$RAISE_DEBUG_EXCEPTION then requests the debug exception signal against the target process by passing the appropriate bit number in PCB$B_REASON (PCB$V_SIGNAL_DEBUG) to the internal subroutine KER$SIGNAL_AST.

When the kernel's condition-dispatching logic calls the debugger's first-chance handler, the request for the process halt is intercepted by the debugger. The target process is then placed in the debug command wait state to allow the user to interact with the process. Because the signal is intercepted by the debugger, the process does not require a condition handler to trap the signal.

6.5.3 Requesting an Asynchronous Exception

All requests for asynchronous exceptions are directed to a single internal subroutine, KER$SIGNAL_AST (in module ASTDELIVR). This routine accepts two register arguments: the bit number in the PCB$B_REASON mask that represents the requested signal and the address of the target process's PCB. KER$SIGNAL_AST is responsible for setting the correct bit in the PCB$B_REASON mask and inserting the appropriate values into PR$_ASTLVL and PTX$V_ASTLVL to request the IPL 2 software interrupt. The ASTLVL values are set to the base access mode of the target process (JCB$B_MODE).

KER$SIGNAL_AST executes as follows:

1. The requested bit is tested in PCB$B_REASON. If that bit is already set, the requested signal is already pending against the process, and the subroutine returns. Otherwise, the appropriate bit is set.
2. The PCB$V_SIGNAL_DISABLE bit is tested in PCB$B_REASON. If that bit is set, the target process has disabled the delivery of asynchronous exceptions. Therefore, the subroutine returns.
3. The execution state of the current process is tested.

   If the target process is running, the value of PR$_ASTLVL must be set. When the kernel procedure that called KER$SIGNAL_AST executes an REI to dismiss the CHMK exception, the microcode requests an IPL 2 interrupt to deliver the asynchronous exception.

If the target process is not running, the PTX$V_ASTLVL field in its hardware context block is modified as well. The point at which the asynchronous exception is delivered depends on the scheduling state of the target process:

a. Ready. When the process is next scheduled to run, the scheduler's final execution of an REI instruction to return to normal system operation immediately delivers the asynchronous exception.

b. Waiting. If the target process is waiting, KER$SIGNAL_AST calls the internal subroutine KER$UNWAIT to place it into the ready state. When the process is next scheduled to run, the scheduler's final execution of an REI instruction to return to normal system operation immediately delivers the asynchronous exception.

   After the asynchronous exception is delivered, however, the target process returns to its original waiting state (unless its wait conditins were satisfied in the meantime). This resumption of the wait is made possible by the interaction of the KER$UNWAIT routine and the kernel vectors for the KER$WAIT kernel procedures. KER$UNWAIT is described in Section 11.3.3.3.

c. Suspended. If the target process is suspended from execution, the asynchronous exception can be delivered only when the process is returned to the ready state. The delivery of the asynchronous exception then awaits the next execution of the process.

4. The subroutine returns to its caller.

6.5.4 Delivering an Asynchronous Exception: The IPL 2 Interrupt

The IPL 2 software interrupt service routine, KER$AST_INTERRUPT, executes in response to the IPL software interrupt. This interrupt is normally requested by microcode (the REI instruction), based on the contents of the ASTLVL register, rather than by the MTPR instruction in the kernel. The MTPR request is used in the single case of process preemption within a job. When the IPL 2 interrupt occurs, control is transferred to KER$AST_INTERRUPT (in module ASTDELIVR), the address in the SCB vector for the IPL 2 software interrupt.

The responsibility of KER$AST_INTERRUPT is to determine the cause of the interrupt. Two possible causes exist:

- The current process has an asynchronous exception pending against it. This is the case when KER$SIGNAL_AST has set PR$_ASTLVL or PTX$V_ASTLVL to request an interrupt against the current process.
- The current process is being preempted. This is the case when the current process has a lower priority than another process that has just become ready within the current job. The IPL 2 interrupt is requested by the KER$READY_PROCESS subroutine (in module SCHEDPRO), which inserts the PCB address of the preempting process into the JCB$A_NEXT_PCB field of the current JCB and requests an IPL 2 interrupt by writing IPL$K_AST_LEVEL to PR$_SIRR.

  KER$AST_INTERRUPT determines that preemption is required by checking the value of JCB$A_NEXT_PCB field and branching to the scheduler if the value is nonzero.

KER$AST_INTERRUPT is entered on the kernel stack in the context of the target process. If the target process is a user-mode process, execution is switched to the user stack before the asynchronous exception is delivered. Because KER$AST_INTERRUPT builds a signal array on the appropriate stack and executes in the context of the target process, the kernel can deliver the asynchronous exception through the uniform condition-dispatching mechanism as if it were a synchronous exception detected by hardware. When the asynchronous exception is delivered, a condition handler established by the target process should take a predefined action in response to the asynchronous exception.

KER$AST_INTERRUPT delivers an asynchronous exception as follows:

1. General registers R0 through R3 are saved on the stack as working registers.
2. The value of JCB$A_NEXT_PCB is tested in the current JCB.

   If that field contains a value, then the IPL 2 interrupt has been requested to start preempting the current process. Control is passed to the scheduler as follows to perform the preemption:

   a. The current PSL is pushed on the stack.

   b. A BSBW instruction is executed to transfer control to KER$RESCHEDULE in module SCHEDJOB.

When KER$RESCHEDULE executes the SVPCTX instruction, the return PC pushed onto the stack by the BSBW and the PSL pushed previously become the return PC and PSL saved in the current process's hardware context block. When this preempted process runs again, it continues execution at IPL 2, on the kernel stack, at the instruction after the BSBW in KER$AST_INTERRUPT.

If an asynchronous exception was requested against the preempted process while it was out of execution, the remainder of KER$AST_INTERRUPT delivers it. If the REI that is executed after the preempted process is restored requests an IPL 2 interrupt (this happens if the preempted process is in user mode), the request is not granted because the restored process is already running at IPL 2 in kernel mode. When IPL returns to 0 after the asynchronous exception is delivered, the additional IPL 2 interrupt is granted and dismissed if no further bits in PCB$B_REASON are set.

3. The LOCK macro is executed to block other software interrupts.
4. The values of PR$_ASTLVL and PTX$V_ASTLVL are set to 4 to prevent another IPL 2 interrupt from being requested for this process.
5. The PCB$V_SIGNAL_DISABLE bit in PCB$B_REASON is set to disable further asynchronous exceptions against this process.

   If PCB$V_SIGNAL_DISABLE is already set, asynchronous exceptions are disabled for the process. The interrupt is dismissed as follows:

   a. The PCB$V_SIGNAL_DISABLE bit in PCB$B_REASON is cleared to reenable asynchronous exceptions.
   b. The general registers are restored.
   c. The REI instruction is executed.
6. The base access mode of the process is obtained from JCB$B_MODE.
7. The current access mode of the process is compared to its base access mode. If the current mode is kernel and the base mode is user, the interrupt cannot be delivered until the mode returns to user. Such spurious IPL 2 interrupts are rare and are dismissed as follows:

   a. The PCB$V_SIGNAL_DISABLE bit in PCB$B_REASON is cleared to reenable asynchronous exceptions.

b. If any of the other bits in PCB$B_REASON is set, the values of PR$_ASTLVL and PTX$V_ASTLVL are reset to 3.

c. The general registers are restored.

d. The REI instruction is executed to dismiss the interrupt and restore IPL to its previous value.

8. The bits in PCB$B_REASON are tested; the first set bit that is found will be the asynchronous exception that is serviced. The bits are tested in the following order:

   a. PCB$V_SIGNAL_POWER

   b. PCB$V_SIGNAL_DEBUG

   c. PCB$V_SIGNAL_QUIT

   d. PCB$V_SIGNAL_PROCESS

   If no bits are set, the spurious interrupt is dismissed as follows:

   a. The PCB$V_SIGNAL_DISABLE bit in PCB$B_REASON is cleared to reenable asynchronous exceptions.

   b. The general registers are restored.

   c. The REI instruction is executed.

9. The set bit is cleared, and the appropriate signal name (for example, KER$_QUIT_SIGNAL) is pushed onto the stack.

10. Control is transferred to a subroutine to perform the following steps:

    a. IPL is lowered from IPL$K_SYNCHRONIZE to 0. The asynchronous exception will be delivered at user IPL; lowering it to that level here allows normal system activity to resume as soon as possible.

    b. If the asynchronous exception is to be delivered to a user-mode process, the data saved on the kernel stack is copied to the user stack. The internal subroutine KER$EXPAND_STACK (in module EXCEPTION) is called to expand the user stack if necessary. If the expansion fails, the asynchronous exception is turned into an access violation.

    c. A PSL appropriate to the access mode of the process (and reflecting IPL 0) is constructed on the kernel stack, and the return address from the subroutine call is pushed on top of it.

    d. The general registers are restored.

e. An REI instruction is executed to return the process to its base access mode. If that mode is user, execution is switched to the user stack.

Execution continues at the instruction following the subroutine call, at IPL 0 and in the appropriate access mode.

11. The argument count (3) of the items remaining on the stack is pushed onto the stack. The following items are now on the stack:
    - The argument count
    - The asynchronous exception signal name
    - The PC at the time of the interrupt
    - The PSL at the time of the interrupt
12. The local procedure DELIVER_AST is called with the CALLG instruction. The argument list in the called procedure points to the information on the stack.

    DELIVER_AST performs the following steps to deliver the asynchronous exception:

    a. A condition handler is established to detect the target process's unwinding of the stack. The handler is also called first in the kernel's search for a handler. Since this handler simply resignals, its presence has no impact on the delivery of the asynchronous exception.

       If the stack is unwound, the handler is called with the signal name SS$_UNWIND. When this occurs, the handler calls KER$ENABLE_ASYNCH_EXCEPTION to reenable asynchronous exceptions for the process and reset the value of ASTLVL if any further asynchronous exceptions are pending. The handler then resignals to allow the unwinding to resume.

    b. A standard VAX signal array is built on the stack as follows:

       i The current PSL is pushed onto the stack.

       ii The address of the RET instruction at the end of DELIVER_AST is pushed onto the stack. This PC and the PSL become current if the process continues from the asynchronous exception.

       iii The signal name and argument count are pushed from the CALLG argument list onto the stack.

    c. Control is transferred to KER$REFLECT in module EXCEPTION, the start of the uniform dispatching mechanism (see Section 6.6).

From this point, the asynchronous exception is treated like any synchronous exception or software condition.

d. If the process continues from the asynchronous exception, control returns to the RET instruction in DELIVER_AST when the exception is dismissed. The execution of the RET returns control to the main line of KER$AST_INTERRUPT, after the CALL to DELIVER_AST.

13. A call is made to KER$ENABLE_ASYNCH_EXCEPTION. As described in Section 6.5.5, this procedure resets PCB$V_SIGNAL_DISABLE and checks to see whether any more asynchronous exceptions are pending against the process. If so, the value of ASTLVL is set accordingly.
14. An REI is executed to dismiss the IPL 2 interrupt. Because IPL has already been lowered to 0, this REI has no effect on IPL.

    If further asynchronous exceptions are pending against the process, the REI requests a further IPL 2 interrupt to deliver the next one indicated by the reason mask.

If the process to which the asynchronous exception was delivered had been in the wait state before the IPL 2 interrupt was granted, after the interrupt is dismissed, its execution continues within the kernel vector for KER$WAIT_ANY or KER$WAIT_ALL. Within the vector, the value of R0 is tested, and the wait is reexecuted if the value is 0. Since the KER$UNWAIT procedure guarantees that the value of R0 will be zero when the process reenters execution, waiting processes interrupted by asynchronous exceptions resume their waiting states after the exception is dismissed.

6.5.5 Disabling and Enabling Asynchronous Exceptions

The kernel provides two procedures to disable and reenable the delivery of asynchronous exceptions. KER$DISABLE_ASYNCH_EXCEPTION (in module ASTCONTRL) disables the delivery of asynchronous exceptions by setting the PCB$V_SIGNAL_DISABLE bit in the PCB$B_REASON mask of the calling process. KER$SIGNAL_AST checks this bit and denies all requests for asynchronous exceptions against the process while the disable bit is set.

KER$ENABLE_ASYNCH_EXCEPTION reenables the delivery of asynchronous exceptions to the calling process by once again clearing the PCB$V_SIGNAL_DISABLE bit in the reason mask. The procedure also checks whether any asynchronous exceptions are now pending against the process by checking the other bits in PCB$B_REASON. If any bits are set, the procedure requests the delivery of an asynchronous exception by writing the process's base access mode to both PTX$V_ASTLVL and PR$_ASTLVL.

As it returns, the procedure executes an REI instruction, which requests an IPL 2 interrupt to deliver the pending asynchronous exception.

6.6 Uniform Condition Dispatching

Once the signal array has been built on the stack, there is no difference in the way the kernel handles synchronous exceptions, asynchronous exceptions, and software conditions. The kernel's operations, beginning at global label KER$REFLECT in module EXCEPTION, are as follows:

1. The mechanism array is constructed on the stack.
2. If the current mode is kernel and the kernel debugger is present, the condition state is passed to the kernel debugger. If the condition is handled there, the condition is dismissed.
3. If the condition was raised by a user-mode process but is being serviced on the kernel stack, the signal and mechanism arrays are copied to the user stack, and the mode is returned to user.
4. The condition-handler argument list is constructed on the stack.
5. If string-instruction emulation is enabled, a subroutine in the emulator is invoked to alter the exception PC, making it appear that the exceptions occurred at an emulated instruction instead of within the emulator.
6. If the debugger is present, it is called to perform a first-chance examination of the condition. The debugger checks to see whether the exception is one of the following:
   - Breakpoint exception (breakpoint reached, SS$_BREAK)
   - Process exit signal (for processes under control of the debugger, KER$_EXIT_SIGNAL)
   - Debugger attention signal (KER$_DEBUG_SIGNAL)

- Trace bit pending trap (used to implement breakpoints, SS$_TBIT)
- Reserved operand fault (used to implement STEP/OVER commands, SS$_ROPRAND)

If the first-chance handler intercepts one of these conditions, it sets a success bit and returns. The condition is then dismissed.

7. If IPL is elevated above process level (0) and the kernel debugger is enabled, it is called. This could be the case when a device interrupt service routine is being debugged.
8. The call frames on the stack are searched for an established condition handler. If one is found, it is called.
9. If the condition is handled, it is dismissed. Otherwise, the stack is searched for another handler.
10. If no handler deals with the condition, the debugger is called as a last-chance handler. If the condition can be handled within the debugger, the condition is dismissed. Otherwise, the process is deleted.
11. If the debugger is not present, the process is deleted.

The following sections describe several of these steps in greater detail.

6.6.1 Building the Mechanism Array and Argument List

To construct the mechanism array above the signal array, KER$REFLECT pushes the following values onto the stack:

- A zero longword. This longword separates the signal and mechanism arrays and provides compatibility with VMS condition-dispatching logic. Later, the high byte of this longword is used to save the original argument count for the signal array. This value must be saved, because condition handlers can alter the signal array argument count for their own purposes. The saved value allows the signal array to be cleared from the stack when the condition is dismissed. (Under VMS, the longword also stores the code that distinguishes a call to LIB$SIGNAL from a call to LIB$STOP.)
- The contents of registers R1 and R0.
- The value –1. This is the initial frame depth in the mechanism array and flags the initial pass in the search for a condition handler.
- The current value of the frame pointer. This is the FP of the frame that raised the signal.

- The value 4 — the number of arguments in the mechanism array.

On top of the mechanism array, KER$REFLECT constructs the condition-handler argument list by pushing the following values onto the stack:

- The address of the mechanism array
- The address of the signal array
- The value 2 — the number of arguments in the condition-handler argument list

In addition, the original argument count for the signal array is copied to the high byte of the VMS-compatibility longword, which separates the signal and mechanism arrays on the stack. This value is used when the condition is dismissed to clear the top of the signal array from the stack.

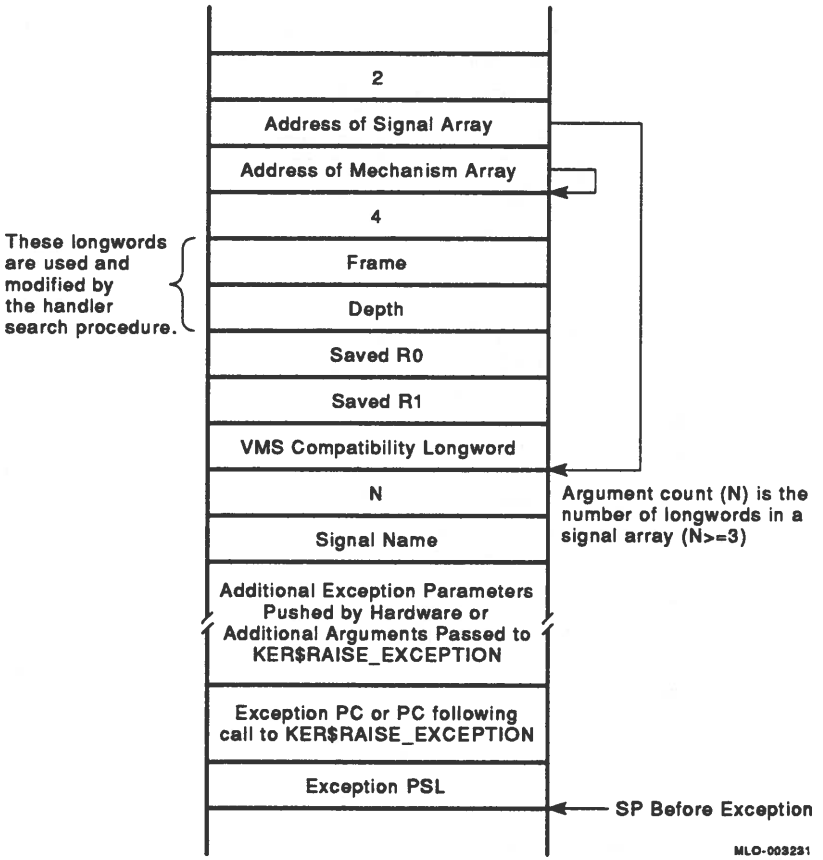
After these operations, the stack appears as shown in Figure 6–5. When a condition handler is called, the argument pointer (AP) points to the argument count in the condition-handler argument list. Then, using the CHF$L_SIGARGLST and CHF$L_MCHARGLST offsets from AP, the handler can obtain any value in either of the arrays.

6.6.2 Reflecting the Condition Back to the Originator's Mode

Exception conditions are reported to a process in the mode in which the exception occurred. If a user-mode process raises an exception that is serviced on the kernel stack, such as an access violation or arithmetic trap, KER$REFLECT copies the signal and mechanism arrays to the user-mode stack and restores the access mode to user by executing an REI instruction.

All exception conditions likely to be reported to a user-mode process are serviced on the kernel stack, so this process is always necessary. For software conditions raised by KER$RAISE_EXCEPTION, the process is never necessary, because that procedure executes in the mode of its caller.

Figure 6–5: Condition Stack



6.6.3 Dispatching the Condition

The major goal of the kernel's condition-dispatching logic is to locate a condition handler in a call frame and call it, passing the condition-handler argument list to the handler.

At this point in the dispatch sequence, the signal and mechanism arrays have been set up on the stack for the access mode in which the condition will be reported. As the search for the condition handler proceeds, the frame and depth fields in the mechanism array are updated to indicate how far the search has progressed. These fields in the mechanism array provide useful information to condition handlers that choose to unwind the stack to an earlier call frame (see Section 6.7.2).

The search for a condition handler begins at the global label KER$DISPATCH_EXCEPTION in module EXCEPTION, where the local procedure SEARCH is called. Within SEARCH, AP points to the top of the condition-handler argument list.

If SEARCH succeeds in locating a handler, the handler is called. If the handler cannot handle the condition (called resignaling the condition), SEARCH is called again. This process continues until a handler deals with the condition (called continuing from the condition) or SEARCH cannot locate another handler.

Figure 6–6 shows the code sequence used, first to call SEARCH and then to call a condition handler at the address returned by SEARCH. In the sequence, SP points to the condition-handler argument list. This argument list is passed to both the SEARCH procedure and the condition handler.

Figure 6–6: Locating and Calling a Condition Handler

```
KER$DISPATCH_EXCEPTION::
        CALLG   (SP),B^SEARCH           ; search for a handler
        BLBC    R0, LAST_CHANCE         ; no handler, try debugger
        CALLG   (SP),(R1)               ; call the handler
```

When a handler continues from a condition, the condition is dismissed by the following sequence:

1. The condition-handler argument list and mechanism array are removed from the stack. As the mechanism array is removed, the saved values of R0 and R1 are restored.
2. All but the last two longwords in the signal array are cleared from the stack. These remaining longwords are the exception PC and PSL.

3. An REI instruction is executed to pop the exception PC and the exception PSL into the appropriate processor registers. In the case of faults, execution resumes at the instruction that raised the signal. For traps (and KER$RAISE_EXCEPTION emulates a trap), execution resumes at the instruction following the one that caused the signal.

If no handler is found or all handlers resignal, the kernel attempts to call the debugger as a last-chance handler, as described in Section 6.6.4.1.

The following sections describe how a condition handler is established, how the SEARCH procedure locates the handler, and what special handling occurs when another signal is raised before the first one is dismissed.

6.6.3.1 Establishing a Condition Handler

Under the VAX architecture, a condition handler is established by writing the address of the handler procedure in the first longword at the top of the call frame. As the kernel searches the stack for a condition handler, it examines this longword in each call frame.

Because the frame pointer always points to this longword, the following VAX instruction establishes a condition handler:

```
MOVAB   handler,(FP)
```

Under the control of a higher-level language, such as VAXELN Pascal or VAX C, a language statement or library routine (ESTABLISH or **vaxc$establish**) performs the same function. In the case of VAXELN Pascal, the ESTABLISH statement actually places the address of the procedure PAS$HANDLER into the handler longword. When called, PAS$HANDLER then calls the procedure the user specified in the ESTABLISH statement. This arrangement places extra call frames on the stack but has no impact on the kernel's dispatch logic.

A condition handler is canceled by removing its address from the handler longword:

```
CLRL    (FP)
```

Issuing the Pascal REVERT statement and calling **vaxc$establish** with a null argument perform this function for their callers.

6.6.3.2 Searching the Call Stack

When called from KER$DISPATCH_EXCEPTION, the SEARCH procedure examines one call frame after another, beginning at the top of the stack, until it locates a frame that has established a handler or it runs out of frames to search. As each frame is examined, the depth and frame arguments in the mechanism array are updated. If SEARCH is called again (because a handler has resignaled), it resumes its search beginning at the call frame pointed to by the last frame argument it set. This ensures that no call frame is searched more than once.

SEARCH takes the following steps to locate a condition handler:

1. The frame pointer argument is obtained from the mechanism array.
2. The depth argument in the mechanism array is increased by 1.

   If this increment causes the depth to become 0, then the current frame belongs to the procedure that raised the signal, and execution skips to step 6. This is the case the first time that SEARCH is called, because the original frame depth is set to –1 by KER$REFLECT.
3. The current frame is tested to see whether it is the frame of a condition handler. If so, more than one condition has been signaled and special handling is required, as described in Section 6.6.3.3.
4. The saved frame pointer in the current call frame is obtained. If no saved FP is available, the bottom of the call stack has been reached, and SEARCH returns failure status.
5. The saved FP value is written to the frame argument in the mechanism array. This means that the next frame down the stack will be examined for a handler.
6. The handler longword in the updated frame argument is tested. If it contains a nonzero value, a success flag is set and the handler address is returned to KER$DISPATCH_EXCEPTION.

   If the handler longword is zero, SEARCH loops back to step 1 to continue the search.

As each handler is called, the updated depth and frame arguments in the mechanism array provide it with the following information:

- The depth argument represents the number of frames that have been searched for a handler before the current handler was called.

- The frame argument represents the value of the saved FP in the call frame that established the current handler.

6.6.3.3 Dealing with Multiple Active Signals

If a signal is raised in a condition handler or in a procedure called by a condition handler, a situation called multiple active signals is reached. To avoid an infinite loop of conditions, the SEARCH procedure called by KER$DISPATCH_EXCEPTION modifies its search algorithm so that those frames searched while servicing the first condition are skipped while servicing the second condition.

For this skipping to work correctly, call frames of condition handlers must be uniquely recognizable. The frames are made so by calling condition handlers from a standard site within module EXCEPTION. Figure 6–7 shows how this call site is identified by the global label KER$CALL_HANDLER_PC.

Figure 6–7: Common Call Site for Condition Handlers

```
KER$DISPATCH_EXCEPTION::
        CALLG   (SP),B^SEARCH           ; search for a handler
        BLBC    R0, LAST_CHANCE         ; no handler, try debugger
        CALLG   (SP),(R1)               ; call the handler

KER$CALL_HANDLER_PC::
        NOP
        BLBC    KER$DISPATCH_EXCEPTION  ; handler resignaled
```

The global label KER$CALL_HANDLER_PC represents the address of the instruction following the call to the condition handler. Within the handler's call frame, the saved PC always corresponds to the value of KER$CALL_HANDLER_PC. When SEARCH locates a call frame that has KER$CALL_HANDLER_PC as its saved PC, it branches to a code sequence that finds the call frame of the procedure that established the current condition handler.

This information is stored in the frame argument in the mechanism array built for this earlier signal, directly below the earlier handler's frame on the stack. SEARCH locates the array by calculating the size

of the handler's call frame, which yields the location of the condition-handler argument list. From this list the address of the mechanism array is obtained, and from there the frame argument is obtained.

To skip the call frames searched during the earlier signal, SEARCH continues examining call frames, beginning with the frame obtained from the handler's mechanism array. The depth argument is not increased to reflect the frames that are skipped, because those frames are not searched during the current signal.

Figures 6–8 and 6–9 show how the modified search procedure functions during multiple active signals. Procedure A has called procedure B, which has called procedure C, which raised signal S. Figure 6–8 shows the stack at this point. Procedures A, B, and C have established condition handlers AH, BH, and CH, respectively.

The numbers in both figures refer to the following steps, which describe the modified search procedure:

❶ Procedure A calls B, which calls C.

❷ Procedure C generates signal S.

In response to the signal, the condition-dispatching logic in module EXCEPTION creates the signal and mechanism arrays on top of the call frame for C and calls handler CH with a depth of 0.

CH resignals, causing its frame to be removed from the stack, and the next frame (B) is searched.

❸ Since B has established a handler, BH is called with a mechanism array depth argument of 1. Again, its frame appears on top of the mechanism and signal arrays for signal S. (See Figure 6–9.) The saved frame pointer in BH's call frame points to the frame for procedure C.

❹ Handler BH calls procedure X, which calls procedure Y (see Figure 6–9).

❺ Procedure Y generates signal T. In the search for a handler for signal T, the desired sequence of frames to be examined is frame Y, frame X, frame BH, and frame A. Frames B and C should be skipped because they were examined for signal S.

❻ The search proceeds in the normal way. Frames Y, X, and BH are examined, with handlers YH, XH, and BHH being called and resignaling in turn. After handler BHH resignals, SEARCH is called again. Examining the saved PC in BH's frame, SEARCH discovers that PC to be KER$CALL_HANDLER_PC. BH was called

as a handler; therefore, SEARCH must postpone its search until it has skipped to the frame beyond BH's establisher.

❼ The skipping is accomplished by locating the frame that established BH. The address of that frame resides in the mechanism array for signal S. To locate the mechanism array for signal S, the value of SP before the call to BH must be calculated, using the register save mask and stack alignment bits in the call frame. Given the value of SP, SEARCH obtains the frame argument from the mechanism array — this is the FP value for procedure B.

Because the frame pointed to by the frame argument has already been searched, the next frame examined by SEARCH is the frame pointed to by the saved FP in the call frame for procedure B — the frame for procedure A.

❽ Since procedure A has established a handler, AH is called. The following depth arguments are passed to the handlers as a result of the modified search for a handler for signal T: 0 for YH, 1 for XH, 2 for BHH, and 3 for AH.

❾ The frame for SEARCH or for any of handlers YH, XH, BHH, and AH is located on top of the signal and mechanism arrays for signal T.

Figure 6–8: Modified Search with Multiple Active Signals, Part 1

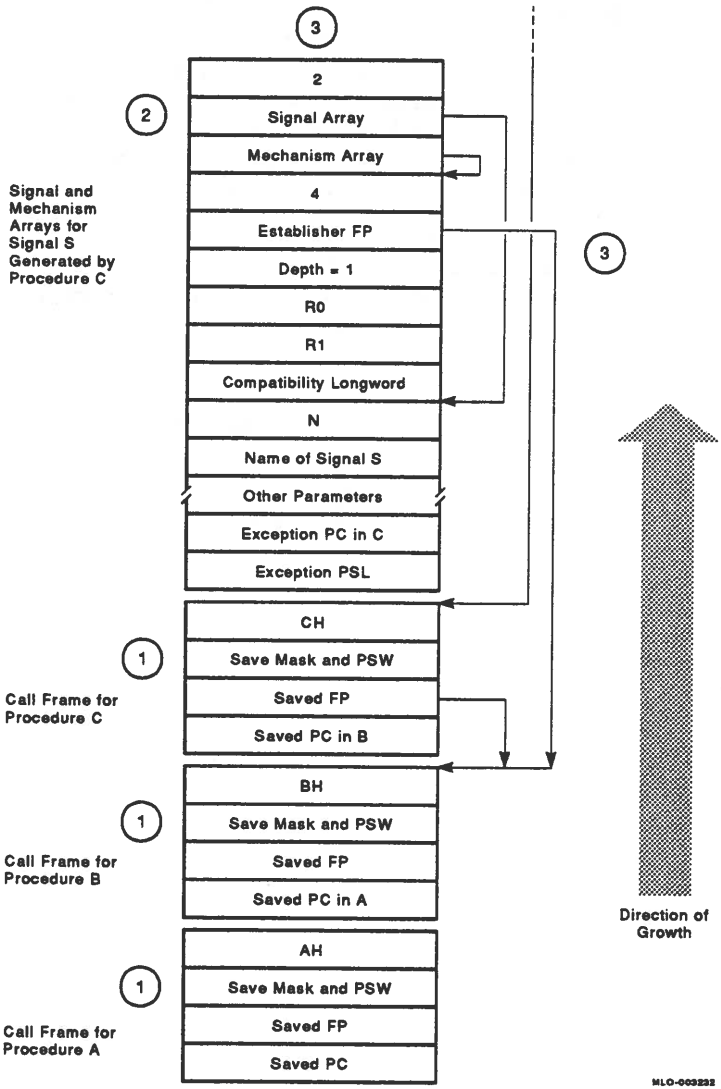
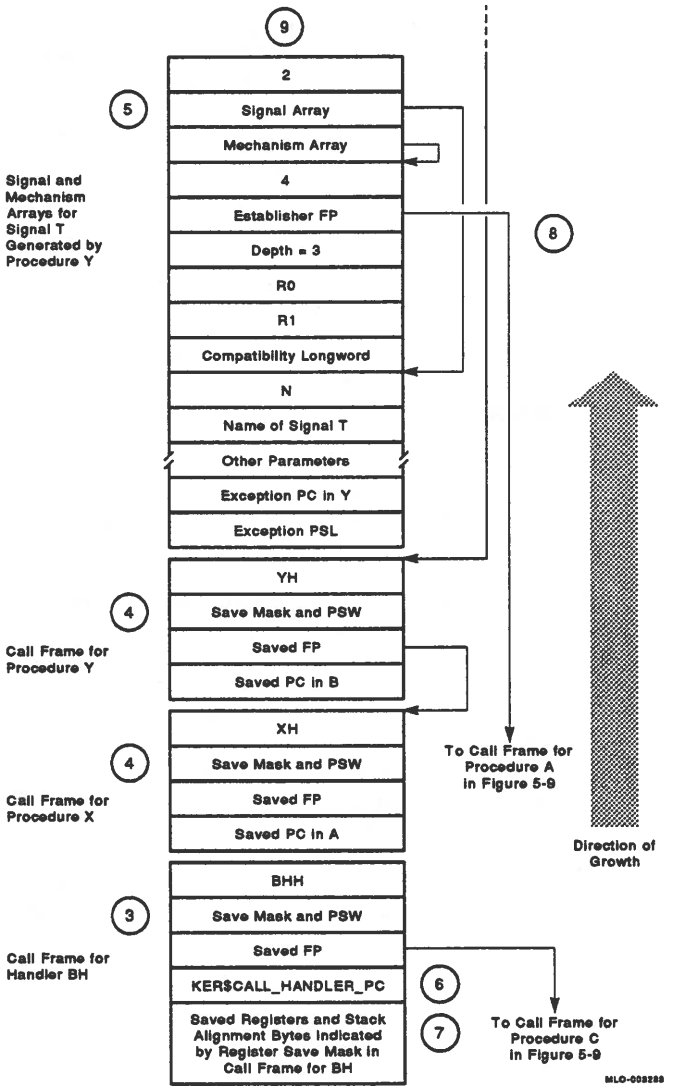


Figure 6–9: Modified Search with Multiple Active Signals, Part 2



6.6.4 Dealing with Unhandled Conditions

If the condition-dispatching logic fails to find a condition handler, or if all the condition handlers found in the stack resignal the exception, the kernel takes two further actions to deal with the condition:

- It attempts to call the debugger as a last-chance handler.
- It forces the offending process to exit.

The following sections describe these actions.

6.6.4.1 Calling the Last-Chance Handler

The kernel does not possess its own last-chance handler. Instead, it relies on the debugger to give the user a final chance to correct an error condition.

Once the SEARCH procedure fails to find a condition handler in the stack, execution branches to a test for the presence of the debugger. If the debugger is not included in the system, control branches to a code sequence that deletes the process (see Section 6.6.4.2).

If the debugger is included in the system (KER$GA_KERNEL_DEBUG_CODE contains a system address), the IPL of the current process is checked. If the IPL is at process level (0), the process-level debugger is called. If IPL is greater than 0, a test is made to determine whether the kernel debugger is present in the system. If so (KER$GA_KERNEL_DEBUG_DATA is nonzero), IPL is set to IPL$K_KERNEL_DEBUG, and the kernel debugger is called. If the kernel debugger is not in the system, the fatal bugcheck INVEXCEPTN (invalid exception) is taken to crash the system.

If the condition is handled by the user employing either debugger, the signal is finally dismissed. Otherwise, the offending process is forced to exit.

6.6.4.2 Forcing Process Exit

When a process cannot handle an exception or software condition, it must be deleted by the kernel. The kernel first obtains the signal name from the signal array for the condition. It then calls the KER$EXIT procedure on behalf of the process; the signal name is the exit status.

If the exiting process is a master process and was created by a program-level call to KER$CREATE_JOB that specified an exit port, the signal name is reported to the job's exit port. If this process is a subprocess and was created by a call to KER$CREATE_PROCESS that specified an exit variable, the signal name is returned to the caller as the exit status for the process.

6.7 Condition Handler Actions

When a condition handler is called by the condition dispatcher, several options are available to it:

- It can fix the condition and allow execution to continue at the interrupted point in the program.
- It can pass the condition along to another handler by resignaling.
- It can also allow execution to resume at any arbitrary place in the calling hierarchy by unwinding a number of call frames from the stack.

The following sections describe these options in more detail.

6.7.1 Continuing or Resignaling

A handler first determines the nature of the condition by examining the signal name field in the signal array. If the handler determines that it is incapable of resolving the current condition, it informs the kernel's condition-dispatching logic by passing an even return status (such as SS$_RESIGNAL) back to its caller, a process called resignaling. Given this return status, KER$DISPATCH_EXCEPTION continues its search down the stack for another condition handler.

If the condition handler is able to resolve the condition, it informs KER$DISPATCH_EXCEPTION of this by passing back an odd return status (such as SS$_CONTINUE).

When KER$DISPATCH_EXCEPTION finds an odd return value in R0, it dismisses the condition as described in Section 6.6.3.

6.7.2 Unwinding the Call Stack: KER$UNWIND

A condition handler can change the flow of execution when a condition occurs. This technique is called unwinding the stack and allows a handler to pass control back to a previous level in the calling hierarchy by discarding a specified number of call frames.

The VAXELN Pascal and VAX C languages offer this unwind capability in the form of the up-level GOTO and **longjump**, respectively. Both change program flow by calling the KER$UNWIND procedure (in module RAISE). KER$UNWIND, however, can be called by a handler written in any language supported under VAXELN. KER$UNWIND accepts as input either an absolute number of frames to be unwound or the actual FP value of the frame at which execution should continue. These alternative uses of KER$UNWIND are described in Section 6.7.2.1.

KER$UNWIND does not actually remove call frames from the stack. Rather, it changes the return PC in the frames above the target frame to point to a special routine within KER$UNWIND that will be executed as each procedure exits with a RET instruction.

As each frame to be unwound executes a RET instruction, registers saved in the call frame are restored and control is passed to the special kernel routine, which examines the current frame for a condition handler. If a handler is established for the frame, a signal and mechanism array are built on the stack, and the handler is called with the signal name SS$_UNWIND. When the handler returns to KER$UNWIND, a RET is issued to discard the current call frame. This sequence continues until the stack has been unwound to the target call frame. Calling handlers as a part of the unwind sequence allows handlers that previously resignaled a condition to regain control and perform procedure cleanup; it also ensures correct restoration of saved registers.

6.7.2.1 Interface to KER$UNWIND

KER$UNWIND was designed to be called by both the VAXELN C and Pascal run-time libraries and by user programs. To accommodate the needs of these different callers, KER$UNWIND provides a flexible calling sequence that allows its caller to specify either an absolute number of call frames to be unwound or a target frame to which the stack should be unwound. (The VAXELN run-time library also provides a procedure called SYS$UNWIND, which emulates its namesake under the VMS operating system.) This section describes the options available to the caller of KER$UNWIND and how the procedure modifies its execution depending on the type of unwind operation requested.

KER$UNWIND takes three arguments: a status argument, a new PC argument, and a depth, or new FP, argument. The optional new PC argument supplies the address at which execution should resume after the stack has been unwound.

The value supplied as the new FP argument significantly affects the operation of KER$UNWIND. These four values specify the following operations:

- A frame pointer value (assumed to be any value greater than 65,535): Unwind to a specific frame. The PAS$GOTO and **longjump** routines specify a target FP when they call KER$UNWIND; therefore, no signal need be active when the procedure is called. This represents an optimized path through KER$UNWIND. In this case, KER$UNWIND skips its usual search for an active condition handler and simply searches down the call stack for the specified frame.

  As it searches, it modifies all intervening frames so that they will be automatically unwound when KER$UNWIND exits. If a new PC has been specified, it is placed in the saved PC field of the call frame above the target frame on the stack.

- A depth greater than 0: Unwind that number of frames. In this case, KER$UNWIND first searches for an active condition handler. If no handler is found, the error status KER$_NOSIGNAL is returned. Therefore, this call can be made only from a handler or from a procedure called by a handler.

  Once it has found the handler, KER$UNWIND counts down the stack from the handler to find the call frame at the specified depth. It then modifies all the frames up to the target frame so that they will be automatically unwound when KER$UNWIND exits. If a new PC has been specified, it is placed in the saved PC field of the call frame above the target frame on the stack.

- A depth of 0: Unwind to the frame that called the establisher of the handler. Again, a handler must be active for this call. Once it has found the handler, KER$UNWIND finds the frame that established the handler. It then knows that the target frame — the caller of the establisher — is one frame further down the stack.

  Next, KER$UNWIND modifies all the frames up to the target frame so that they will be automatically unwound when KER$UNWIND exits. If a new PC has been specified, it is placed in the saved PC field of the call frame above the target frame on the stack.

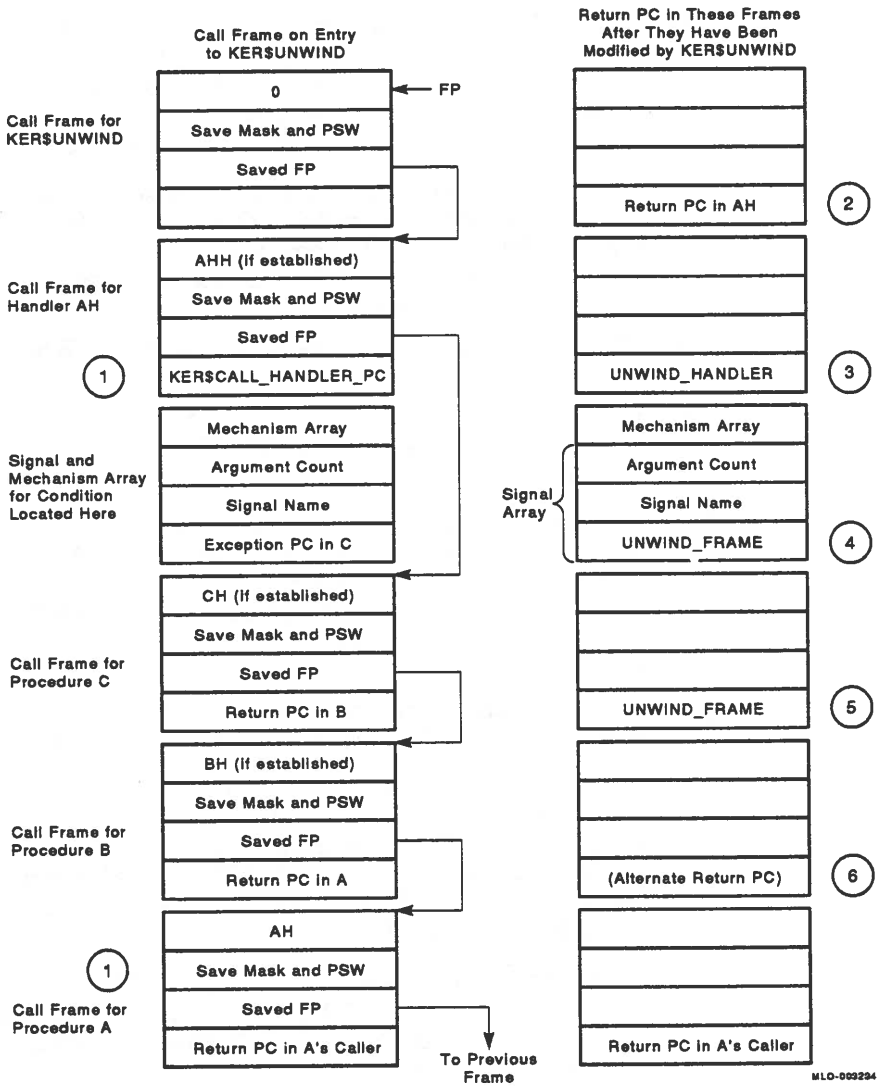
- A depth of –1: Unwind 0 frames. Again, a handler must be active for this call. Once it has found the handler, KER$UNWIND simply returns — unwinds zero frames. If a new PC has been specified, it is placed in the saved PC field of the call frame above the target frame on the stack. In general, unwinding 0 frames means to return execution to the frame that raised the signal. Supplying a new PC in this case allows the caller to resume execution at a new location within that frame. This is the main purpose for unwinding 0 frames.

6.7.2.2 A Sample Unwind

Figure 6–10 illustrates how KER$UNWIND manipulates the return PCs in the call frames on the stack to effect the return of control to the target frame. The example begins with the same sequence shown in Figure 6–8. Procedure A calls procedure B, which calls procedure C. Procedure C generates signal S. Handlers CH and BH, located by KER$DISPATCH_EXCEPTION, resignal.

Handler AH is then called. AH decides to unwind the stack back to the frame that established it, procedure A. To accomplish this, AH calls KER$UNWIND with a depth argument equal to the value contained in the depth argument of the mechanism array (the frame argument can also be used). In this example, the depth argument is 2. After the call to KER$UNWIND, which executes in the access mode of its caller, but before the frame modification occurs, the stack appears as shown on the left side of Figure 6–10.

Figure 6–10: Call Frame Modification by KER$UNWIND



The frame modification now proceeds as follows (the numbers refer to the numbers shown in the figure):

❶ The stack is searched until a condition handler is located; a handler's frame is marked by having the value of KER$CALL_HANDLER_PC as its saved PC. (This search occurs only when the procedure is called with a depth argument; if it is called with an FP value, no search for a handler frame is conducted.)

If KER$UNWIND is called with a depth argument, the first call frame modified is the frame of the first handler in the stack, in this case, the frame for AH. Therefore, if AH had called a procedure that then called KER$UNWIND with a depth argument of 2, that nested procedure's frame would not be altered by KER$UNWIND.

❷ KER$UNWIND's own frame is not modified. When KER$UNWIND exits, control returns directly to AH.

❸ The frame for AH is modified. The saved PC in its call frame is replaced with the address UNWIND_HANDLER, a routine internal to KER$UNWIND.

All return PCs of handler frames encountered on the way to the target frame are replaced with the address of UNWIND_HANDLER. This routine forces the handler to return a continue status and then transfers control to KER$CALL_HANDLER_PC in module EXCEPTION to clear the mechanism and signal arrays from the stack.

❹ The exception PC in the mechanism array is replaced with the address of UNWIND_FRAME, a routine internal to KER$UNWIND.

When signal S is dismissed, execution continues on the frame of procedure C at UNWIND_FRAME. UNWIND_FRAME checks to see whether C has established a handler. If so, that handler is called. When it returns, UNWIND_FRAME executes a RET instruction on behalf of procedure C, returning execution to the call frame for procedure B.

If a handler called from UNWIND_FRAME attempts to unwind the stack by calling KER$UNWIND with a depth argument, its call will fail with the status SS$_UNWINDING, indicating that an unwind is already in progress.

❺ The return PCs in additional frames on the stack are modified (how this is done depends on whether the frame belongs to a condition handler or to a normal procedure). This modification continues until the target frame has been reached in the stack.

In the example, the return PC in the frame for procedure C, because it is not a handler, is replaced with the address of UNWIND_FRAME.

❻ If the alternate PC argument was also passed to KER$UNWIND, the saved PC in the frame that will return to the target frame — that of procedure B — is replaced with the specified PC value.

Once the frames have been modified, the actual unwinding occurs in the following sequence:

1. KER$UNWIND returns control to handler AH.
2. When AH issues a RET instruction, control continues on the frame for procedure C at UNWIND_HANDLER.
3. UNWIND_HANDLER sets the low bit in R0 to force AH's return status to continue. Control then branches to KER$CALL_HANDLER_PC in module EXCEPTION, which clears the condition arrays off the stack, restores R0 and R1, and issues an REI instruction to dismiss the signal.
4. The REI pops the address of UNWIND_FRAME into the PC, where execution continues on procedure C's frame.
5. UNWIND_FRAME performs the following steps:
   a. If a handler is established for this frame, it is called with the signal name SS$_UNWIND.
   b. If either R0 or R1 is specified in the register save mask, UNWIND_FRAME replaces the value of the register in the register save area of the call frame with the current contents of the register. (This is an unusual case; the VAX procedure calling standard specifies that R0 and R1 are to be used to return status codes and function values.)
   c. Control is returned with a RET instruction to whatever address is specified in the saved PC of the current call frame.
6. The RET issued by UNWIND_FRAME discards the call frame for procedure C, passing control again to UNWIND_FRAME, this time on the frame for procedure B. UNWIND_FRAME again performs its three steps on behalf of procedure B.
7. The RET that discards the call frame for procedure B passes control back to the point in procedure A following the call to procedure B, (if no alternate return PC was specified) where execution will resume.

In effect, UNWIND_HANDLER and UNWIND_FRAME simulate returns from each nested procedure that is being unwound. These procedures never again receive control. However, the target procedure receives control as if all the nested procedures had returned normally.

6.7.2.3 Unwinding Multiple Active Signals

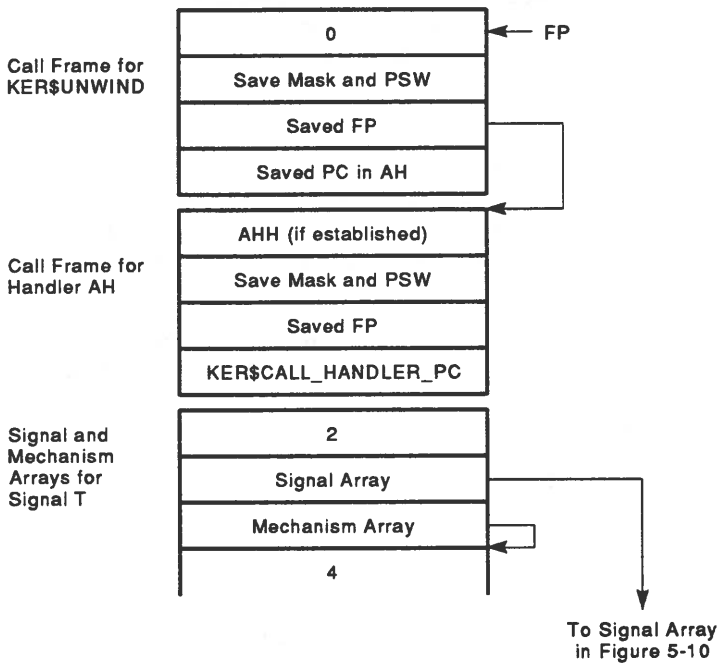
KER$UNWIND modifies its actions slightly when multiple signals are active and it has been called with a depth argument. Before modifying saved PCs, KER$UNWIND counts down the call frames in the stack to find the target frame for the unwind. It normally counts down the number of frames specified by the depth argument, beginning with the frame after the frame of the first condition handler. The frame it reaches when the count is exhausted is the target frame for its modification sequence.

When multiple active signals are present, KER$UNWIND parallels the action of the SEARCH procedure in module EXCEPTION (see Section 6.6.3.3); that is, it skips over the frames that were searched for a handler while the first signal was active. KER$UNWIND skips these frames by stopping its countdown until it reaches the frame that established the handler called in response to the first signal. (None of this applies when KER$UNWIND is called with a frame argument, because that frame's FP value is already known.)

The example of multiple active signals shown in Figures 6–8 and 6–9 can illustrate the modified unwinding. Recall that procedure A called procedure B, which called procedure C, which signaled S. Handler CH resignaled. Handler BH called procedure X, which called procedure Y, which signaled T. Handlers YH, XH, and BHH all resignaled. Finally, handler AH was called for signal T with a depth of 3.

If AH calls KER$UNWIND with a depth argument of 3 (to unwind to its establisher), the top of the stack is as shown in Figure 6–11. Assume that no alternate PC argument was specified.

Figure 6–11: Modified Unwind with Multiple Active Signals



The end result, then, of the operation of KER$UNWIND is as follows:

1. The stack is searched for a condition handler's frame; in this case, AH's frame is found.

   The saved PC in this frame is eventually replaced with the address of UNWIND_HANDLER, and the exception PC in the signal array for signal T is replaced with the address of UNWIND_FRAME.

2. The countdown of the stack begins. The first frame is that of procedure Y. The saved PC in this call frame is replaced with the address of UNWIND_FRAME. This is the first frame.

3. The next frame is that of procedure X. The saved PC in this frame is also replaced with the address of UNWIND_FRAME. This is the second frame.

4. The next frame on the stack belongs to BH, which was called as a condition handler (its saved PC is KER$CALL_HANDLER_PC). Its associated mechanism array is located by climbing over saved registers and stack alignment bytes. From this array, the frame of BH's establisher (B) is obtained.

   The PCs in all the frames that were not counted will eventually be modified. In this case, these are the frames for BH and C.

5. The count resumes with the frame that established BH, procedure B. Since B was not called as a handler, it is counted as frame 3, and the countdown is complete. KER$UNWIND now knows that the target frame for the unwind is the frame that called B: procedure A.

   The saved PC in B will not be modified. When it returns, execution will continue in procedure A after the call to B.

Chapter 7

Error and Event Reporting

The VAXELN Kernel and certain VAXELN subsystems can monitor their functioning and that of their associated hardware and software for the occurrence of errors. When errors and events occur, the VAXELN error-logging subsystem enables the software to report them in a local or remote error log file. Some errors, such as internal processor errors called machine checks, may require the shut-down of the system; others errors may require that a process be forced to exit. The kernel can shut down the system or force a process to exit using its bugcheck mechanism.

This chapter describes the following components of VAXELN's systemwide error reporting and handling systems:

- The error-logging subsystem, which enables the kernel and system jobs, such as device drivers, to report errors and other events in a local or remote error log file. See Section 7.1.
- The kernel's bugcheck handling mechanism. See Section 7.2.
- The kernel's machine-check handling mechanism and support for recovery from machine checks. See Section 7.3.

7.1 Error Logging Subsystem

The components of the error-logging subsystem are designed to minimize the kernel's role in logging errors, especially in systems that do not select support for error logging from the System Builder. Therefore, the kernel component of the error-logging subsystem comprises a small set of subroutines that allocate error message buffers and insert them into a queue of posted buffers. The larger component of the subsystem

exists as a job, called the ERRFORMAT job, that removes the buffers from the posted queue and writes them to the error-log file.

If error-logging support is included in a system, the ERRFORMAT job is included in the system image and begins to run at system initialization. A system without error logging enabled has no ERRFORMAT job and contains only the small core of kernel subroutines, which performs no significant function when a loggable event occurs. Only when error logging is included do these subroutines operate in full, and, even then, they execute a minimal number of instructions to post log entries for processing by the ERRFORMAT job.

The following sections describe the components and operation of the error-logging subsystem in greater detail, focusing on the following aspects of the system:

- The errors and events reported by the subsystem
- The components of the subsystem, including its major data structures, the ERRFORMAT job, and the error-log server that runs on a VMS system to support remote error logging
- The operation of the subsystem in logging an error or event

7.1.1 Errors and Events Reported by the Error-Logging Subsystem

The following errors and event are logged by the error-logging subsystem:

- Device errors. The peripheral mass storage devices that may be attached to VAXELN systems and supported by the error-logging subsystem consist of disk and tape drives using MSCP and TMSCP interface hardware and device drivers. Devices that do not use the VAXELN MSCP and TMSCP class drivers are not currently supported by the error-logging subsystem. The drivers report both controller/device hardware errors and media errors (such as bad block replacement).
- Machine checks. A machine check is an exception that results when the processor or an external adapter or controller detects a hardware error. Because machine checks are processor-specific, each VAX processor or processor class supported by the VAXELN has its own machine-check handler. The handler deals with the processor-specific portions and provides an interface to the kernel's uniform condition-dispatching logic and bugcheck mechanism.

Machine-check handlers may also log the following processor-specific hardware error:

— Bus errors

— Cache errors

— Soft (recoverable) and hard (unrecoverable) memory errors

Machine-check handling is described in Section 7.3.

- Bugchecks. When the VAXELN kernel detects an internal inconsistency, such as a corrupted data structure or unhandled exception, it declares a bugcheck. If the system can continue running, a nonfatal system bugcheck is declared, an error-log record is posted, and the system continues operating.

  In the case of a serious error whose effect on the system's integrity is uncertain, a fatal system bugcheck is declared to perform an orderly shut-down of the system. The contents of any posted error-log buffers, including those holding the bugcheck information, are written to the system's dump file before the system shuts down.

  Bugcheck handling is described in Section 7.2.

- Last-fail information. In the event of a fatal system bugcheck, the kernel dumps as much information as possible about the state of the system to the console terminal and, if the system dump facility is enabled, to a dump file on a local disk. When the system is next booted, the information in the dump file is recovered and written to the error-log file as error/event log entries.

- System service messages. Certain error conditions (such as being unable to initialize the crash dump subsystem or dump file) cause system service error-log entries to be posted. These entries consist of up to 255 bytes of ASCII text and are intended to assist in the evaluation and analysis of the final error-log report produced by the VMS Error Formatting Utility (ERF).

  The mechanism for generating these log entries is provided by the procedure ELN$LOG_EVENT, which is included with the ERRFORMAT module when error logging is selected. The ELN$LOG_EVENT procedure expects two arguments: the size of the message string and its address.

- System start-up. An error-log record entry is made for each successful system start-up or restart. System start-up can occur in one of the following circumstances:

  — Cold start, the initial bootstrap load of the system

— Warm start, the automatic restart of the system following a power failure

- Mass-storage volume activities. Whenever a disk or tape volume is mounted on a local mass storage device, the details of each mount/dismount transaction are recorded in an error-log record entry. A volume mount/dismount transaction contains information such as the generic device name, device unit number, volume label, and, on dismounts, unit error count. The VAXELN File Service is responsible for posting all volume activity record entries to the error-logging subsystem.
- New file creation. Whenever the error-logging subsystem creates a local log file, a record recording that event is written to the log. The creation of remote log files is not record directly by the local system; instead, that activity is written by the VMS host error-logging server (ELSE).
- Time stamps. The VAXELN error-logging subsystem posts a time-stamp record entry every ten minutes. Posting these time-stamped records provides a chronological audit trail of the normal operation of the system — information valuable when sporadic or intermittent system failures must be isolated.

7.1.2 Components of the Error-Logging Subsystem

The error-logging subsystem contains a number of components that support logging errors and events to both local and remote locations. The following sections describes these parts of the subsystem:

- Error-logging data structures
- The kernel subroutines that support error logging
- The ERRFORMAT job
- The system dump facility
- The error-logging server (ELSE)

7.1.2.1 Error-Logging Data Structures

The error-logging subsystem relies on a number of kernel data structures and global values. The following sections describe the error message buffer, which contains the data for the error-log entry, and the system global values and structures that support logging those buffers to the error log.

7.1.2.1.1 Error Message Buffers

Error message buffers (EMBs) are allocated each time a hardware or software event is to be logged. EMBs are allocated by the kernel subroutine KER$ALLOCEMB from a queue of available EMBs located at KER$GQ_EMB_AVAIL. These 512-byte buffers hold the information used to create an error-log entry.

Every EMB has a header, the EMB header, used by the kernel, and an EMB record header, used in error-log analysis. The EMB header contains the address linkages used to insert the EMB into the kernel's EMB queues for later processing. Table 7–1 shows the fields in an EMB header.

Table 7–1: EMB Header Fields

Field	Meaning
EMB$A_FLINK EMB$A_BLINK	Link to the next and previous EMB
EMB$W_SIZE	The size in bytes of this EMB's log entry, not including the EMB header itself

The EMB record header contains information required by the VMS Error Log Utility in the generation of an error log report. Table 7–2 shows the fields in an EMB entry record header.

Table 7–2: EMB Record Header Fields

Field	Meaning
ERL$L_SID	The system identification
ERL$W_HDRREV	The header revision level
ERL$L_SYS_TYPE	The contents of the system-type register
ERL$L_SMP_ID	The unique processor identifier
ERL$T_NODENAME	The SCS node name
ERL$W_FLAGS	Error log entry flags
ERL$W_ENTRY	The error log entry type
ERL$Q_TIME	The time the entry was posted
ERL$W_ERRSEQ	The error sequence number

The ERL$W_ENTRY field in the record header defines the type of the log entry for later analysis. This field can take one of the values shown in Table 7–3. The type value is inserted in the ERL$W_ENTRY field when the kernel initializes the record header before posting the EMB.

Table 7–3: Error-Log Entry Types and Their Values

Type Symbol	Value	Meaning
ERL$K_MACHINECHK	2	Machine check
ERL$K_SOFTERROR	6	Soft memory error
ERL$K_ASYNCWRITERR	7	Asynchronous write error
ERL$K_HARDERROR	8	Hard memory error
ERL$K_BIADPERR	18	VAXBI adapter error
ERL$K_BIBUSERR	19	VAXBI bus error
ERL$K_CACHEBUSERR	24	Bus/cache error
ERL$K_COLDSTART	32	Cold system start
ERL$K_NEWFILE	35	New log file creation
ERL$K_WARMSTART	36	Warm system start
ERL$K_CRASHRESTART	37	Crash-restart
ERL$K_TIMESTAMP	38	Time stamp
ERL$K_SYSSERVMSG	39	System service message
ERL$K_SYSBUGCHK	40	System bugcheck
ERL$K_VOLMOUNT	64	Volume mount
ERL$K_VOLDISMOUNT	65	Volume dismount
ERL$K_DEVATTENTION	98	Asynchronous device attention
ERL$K_SOFTPARAMS	99	Software parameter
ERL$K_LOGGEDMSG	100	Logged message
ERL$K_LOGMSCP	101	Logged MSCP message
ERL$K_PROCBUGCHK	112	Process bugcheck

The remainder of the EMB contains information specific to the error or event being logged. For example, the EMB for a volume mount or dismount contains information such as the device and unit number, the device name, the owner's UIC, and the volume label.

The pool of EMBs is created during system initialization, when the number of physical pages corresponding to the global parameter KER$GW_EMB_COUNT is allocated and mapped into system address space. This number is divided by two; the result becomes the global value KER$GW_MAX_POSTED, which is used as the threshold at which the ERRFORMAT job is awakened when the pool of EMBs has been half depleted.

As each EMB is mapped, it is inserted as an entry in the queue of free EMBs pointed to by the global structure KER$GQ_EMB_AVAIL. The EMBs are zeroed when they are removed from the queue for use.

Preallocated EMBs, called the crash-restart logs, are also created during system initialization. One page of physical memory is mapped and zeroed as a crash-restart log for each processor in the system. A crash-restart EMB is used to hold the information that explains a system shut-down.

7.1.2.1.2 System Data Items

A number of global data elements that support error logging reside in the system data block. Table 7–4 shows these values and their roles in error logging.

Table 7–4: System Data Items That Support Error Logging

Field	Meaning
KER$GA_CRASHLOG	An array of addresses of crash-restart EMBs used in logging fatal system bugchecks.
KER$GA_ERRFMT_JCB	The address of the job control block for the VAXELN ERRFORMAT job. The kernel uses this JCB to activate the ERRFORMAT job.
KER$GB_ERRLOG_ ENABLE	A Boolean value that indicates whether the user selected error logging in the System Builder. If the lower bit of this byte is set, error logging is enabled.
KER$GL_ERRFMT_ WAKEUP	The identifier value for the event object used to control the execution of the ERRFORMAT job.

Table 7–4 (Cont.): System Data Items That Support Error Logging

Field	Meaning
KER$GQ_DEVICE_QUEUE	The queue of device objects waiting for processing by the kernel. The kernel places the event object associated with the ERRFORMAT job in this queue to force its execution to flush the queue of posted EMBs. See Section 11.3.2 for more information about the device queue.
KER$GQ_EMB_AVAIL	The listhead for the queue of available EMBs.
KER$GQ_EMB_POSTED	The listhead for the queue of posted EMBs waiting to be flushed to the error-log file.
KER$GW_EMB_COUNT	The number of EMBs in the system, as indicated on the Error Log Characteristics Menu. The minimum number of EMBs is 2.
KER$GW_EMB_SIZE	The size in bytes of an EMB, currently 512.
KER$GW_ERRSEQ	The error sequence number. This value is increased each time an attempt is made to allocate an EMB and post an entry to the error-log file. Gaps in the sequence number, as reflected in the error log, indicate a failure to allocate an EMB for an error-log entry. This value is updated using the ADAWI interlocked instruction, which synchronizes access to the sequence number.
KER$GW_MAX_POSTED	The maximum number of EMBs that can be posted before the ERRFORMAT job will write them to the error log file. This value is half the value of KER$GW_EMB_COUNT.

7.1.2.2 Kernel Error-Logging Components

A small core of error-logging support resides in the kernel, regardless of whether the error-logging subsystem has been included in the system. This kernel-resident code is executed when attempts are made to log errors and events; if the error-logging subsystem is not present, control returns immediately to the original instruction stream.

The kernel core includes the following internal subroutines (in module ERRORLOG) to support error logging:

- The KER$ALLOCEMB subroutine allocates error message buffers. See Section 7.1.3.1.1.

- The KER$COLDSTART and KER$WARMSTART subroutines post start-up and power-failure recovery log entries, respectively.
- The KER$INIT_ERLHEADER subroutine, called by KER$RELEASEMB, initializes an EMB record header.
- The KER$POST_ERRORLOG procedure inserts an EMB into the queue of posted EMBS. This procedure calls the kernel subroutines KER$ALLOCEMB and KER$RELEASEMB to allocate and post the EMB.
- The KER$RELEASEMB subroutine releases error message buffers by inserting them into the queue of posted EMBs, so that the ERRFORMAT job can remove them and write them to the error-log file. The subroutine also calls KER$WAKEUP to activate the ERRFORMAT job if the threshold of posted EMBs has been exceeded. See Section 7.1.3.1.1.
- KER$WAKEUP activates the ERRFORMAT job by causing its event object to be signaled. See Section 7.1.3.2.

7.1.2.3 ERRFORMAT Job

The ERRFORMAT job, an optional component in the error-logging subsystem, is responsible for reading the queue of posted error-log entries and writing them to the error log file. It is also responsible for inserting time stamps and posting new-file-creation messages to local log files. When the system is restarted, if last-fail information has been written to the system dump file on a local disk, ERRFORMAT retrieves it and posts it to the error log file. See Section 7.1.3.3.

7.1.2.4 System Dump Facility

The system dump facility writes last-fail information to a local dump device for later recovery by the error-logging subsystem. It does not require any user action: if there is a system crash (fatal system bugcheck), the last-fail data will be recovered on the next system reboot if error logging is enabled. The error-log file can then be transferred to a VMS system for analysis with the ERF utility.

The system dump device must be a local disk. A device driver dispatcher and a hard-coded, minimally functional driver is incorporated into the final system image for which last-fail dumping is enabled. When last-fail information is to be written to disk, the dispatcher is called to select the proper dump driver to perform all I/O to the dump device.

7.1.2.5 Error-Logging Server

VAXELN provides for remote logging of errors over the Ethernet to a VAX system running under the VMS operating system and participating in the same local area network. The VMS system runs the error-logging server (ELSE), which accepts error-logging messages transmitted over the Ethernet from the VAXELN system. The server writes the error log to a file on the VMS system, pointed to by the logical name ELSE$ERRORLOG, which can be read and formatted by the VMS Error Log Utility.

ELSE may have up to 20 virtual circuits established at any given time. Each circuit represents a connection to a VAXELN system that has elected to use the remote logging feature of the error-logging subsystem. ELSE writes a separate error log file for each VAXELN system it serves and uses the System Communication Services (SCS) node name or number to create the error log file name. For example, if the VAXELN node PIGDOG uses the remote logging server, the error-log file created by ELSE will be named PIGDOG.SYS.

7.1.3 Error-Logging Operation

The following sections describe the actual operations of the error-logging subsystem, namely:

- The posting of an error or event directly from the kernel and from a job, such as a device driver
- The awakening of the ERRFORMAT job to write out EMBs
- The operation of the ERRFORMAT job

7.1.3.1 Posting an Error or Event

When an error or event occurs, a record of its occurrence must be posted to the error log by the kernel. Errors or events logged from job level are posted with the KER$POST_ERRORLOG kernel procedure. This procedure is the general interface to the error-logging subsystem and is provided for Digital-supplied device handlers. Within the kernel itself, this procedure interface can be bypassed, and error-log entries can be posted directly by manipulating the EMB queues. The following sections describe these two approaches to posting an error-log entry. The kernel-level operation — the basis for KER$POST_ERRORLOG — will be described first.

7.1.3.1.1 Posting Error-Log Entries from Kernel Level: KER$ALLOCEMB and KER$RELEASEMB

In certain instances in the operation of the kernel, error-log entries can be made without invoking the formal posting procedure required from job context. When system start-ups, system and process-level bugchecks, and processor-specific entries are logged, the kernel posts them directly by calling the internal subroutines KER$ALLOCEMB and KER$RELEASEMB. Both KER$ALLOCEMB and KER$RELEASEMB use interlocked queue instructions to synchronize access to the EMB queues.

When posting an error or event, the kernel first obtains a free EMB by calling the internal subroutine KER$ALLOCEMB. If the buffer is available, the kernel then initializes the EMB record header. The failure of KER$ALLOCEMB to allocate a buffer marks the end of the kernel's processing of the error or event, whether or not the system has error logging enabled.

The subroutine KER$ALLOCEMB expects the size of the requested EMB as an input value and returns the address of the allocated EMB's record storage area. The routine executes as follows:

1. An attempt is made to remove an EMB from the head of the queue of available EMBs (KER$GQ_EMB_AVAIL) using a VAX interlocked queue instruction. If the allocation fails, KER$GW_ERRSEQ is increased by 1, and a null EMB address is returned. The kernel will make no further attempt to post the error or event to the error log.
2. The size of the EMB is set in the EMB$W_SIZE field of the EMB header.
3. The EMB, including the EMB record header, is zeroed.
4. The address of the initialized EMB's record storage area is returned to the caller.

If the allocation of the EMB succeeds, the kernel fills in the entry-specific fields and then passes the error-log entry type and the address of the allocated EMB to the KER$RELEASEMB subroutine. KER$RELEASEMB initializes the remainder of the EMB record header and inserts the EMB into the queue of posted EMBs. This subroutine is also responsible for tracking the number of posted EMBs and calling KER$WAKEUP, described in Section 7.1.3.2, if the number exceeds the threshold value. KER$RELEASEMB executes as follows:

1. The error sequence number KER$GW_ERRSEQ is increased by 1.

2. A call is made to the internal subroutine KER$INIT_ERLHEADER_S to initialize the entry's record header. The routine expects two inputs: the error log entry type and the address of the EMB.
3. The EMB is inserted at the tail of the queue of posted entries, KER$GQ_EMB_POSTED, with an interlocked queue instruction.
4. The number of posted entries, KER$GW_CNT_POSTED, is updated, and a check is made to see whether the threshold has been exceeded. If not, the routine returns.
5. If the threshold has been exceeded, a call is made to the internal subroutine KER$WAKEUP to unblock the ERRFORMAT job, which actually writes the error log records to the log file.

7.1.3.1.2 Posting Errors and Events from Job Level: KER$POST_ERRORLOG

To post an error in the error log, jobs call the KER$POST_ERRORLOG kernel procedure with arguments indicating the type of log entry to be made, the address of the entry record text, and the size of that text. This procedure builds a shell around the kernel subroutines KER$ALLOCEMB and KER$RELEASEMB, which cannot be called directly from a job. The MSCP and TMSCP class drivers use this procedure to log errors and events for their associated devices.

KER$POST_ERRORLOG executes as follows:

1. The size of error-log entry text is compared to the size of the EMB remaining after the size of the record header and EMB header are taken into account. If the text does not fit into the remaining space, KER$_BAD_VALUE is returned to the caller.
2. Control branches to KER$ALLOCEMB to allocate an EMB.
3. If KER$ALLOCEMB returns a null address, the allocation has failed, and the procedure simply returns success status. The entry is not logged. The loss of the log entry will be reflected by a gap in the sequence number. If error-logging support is not enabled, the kernel makes no further attempt to log the error.
4. The error log text is copied into the EMB following the record header.
5. The entry type and the address of the now completed EMB are passed to the KER$RELEASEEMB subroutine to post the error log entry.
6. The procedure returns with success to its caller.

7.1.3.2 Awakening the ERRFORMAT Job with KER$WAKEUP

The mechanism that awakens ERRFORMAT is of particular importance in the error-logging operation, because the kernel must be able to awaken the job from kernel or user mode, from process or system context, at any IPL, and from any processor in a tightly coupled symmetric multiprocessing configuration. The only mechanism that guarantees that capability is the KER$SIGNAL_DEVICE procedure, which can be issued from any context and at elevated IPLs — conditions under which interrupt service routines, from which it is usually called, operate.

The kernel subroutine KER$WAKEUP, when called by KER$RELEASEMB, performs the equivalent of calling KER$SIGNAL_DEVICE for ERRFORMAT's event object. The KER$WAKEUP takes the following steps to get the job running:

1. A check is made to determine whether ERRFORMAT is present in the system by testing the value of KER$GA_ERRFMT_JCB. If the value is zero, then ERRFORMAT is not present and the routine returns. If the value is nonzero, it represents the address of ERRFORMAT's JCB.
2. The type field in the object pointed to by KER$GA_ERRFMT_JCB is checked to confirm that it is a JCB. If not, the routine returns.
3. Using the JCB, the event object identifier in KER$GL_ERRFMT_WAKEUP is translated to the address of the object. If the object is not an event object, the routine returns.
4. The event object is inserted into the device signal queue, whose listhead is located at KER$GQ_DEVICE_QUEUE. (The structure of an event and a device object share several fields that allow the event object to be treated as a device object.)
5. If the object is not the first entry in the queue, the routine returns, because at least one device object has already been signaled. If it is the first entry, then an IPL software interrupt is request to signal to interrupt and initiate the processing of the device wait queue. This technique guarantees that ERRFORMAT can be awakened at any IPL from either system or process context. Section 11.3.2 describes device signaling in greater detail.

7.1.3.3 Operation of the ERRFORMAT Job

The ERRFORMAT job is responsible for removing EMBs from the posted queue and writing them to the error-log file. The job is created because the System Builder includes it in the list of jobs that require initialization at system start-up. When ERRFORMAT initializes, it completes the following activities:

1. It obtains its job arguments. These arguments indicate the logging method (local disk or network) and, for local disk, the location of the disk and log file.
2. If the error-log destination is the network, ERRFORMAT creates a port to be used for sending error-log messages to ELSE.
3. It saves the address of its JCB as the global value KER$GA_ERRFMT_JCB. This address is used in waking up ERRFORMAT when the EMB threshold has been reached.
4. It creates an event object to use to block its execution until it is awakened to write out EMBs. The identifier for the event object becomes the global value KER$GL_ERRFMT_WAKEUP.
5. It initializes its time-stamp interval (10 minutes). When this interval expires, the job awakens to post a time-stamp entry in the error log.
6. It calls the kernel procedure KER$INTIALIZATION_DONE to indicate that it has completed its initialization code.
7. It determines the logging method, either local or remote. If logging is to a local disk, then ERRFORMAT attempts to recover any last-fail information that may have been dumped to the disk following a system crash. The contents of the dump file are treated as standard EMBs and are written to the current error-log file. If the system dump facility is currently included, the dump device and dump file are initialized.
8. It enters a closed loop to write EMBs to the error log file. If no EMBs require processing, the job calls KER$WAIT_ANY to wait until it is reactivated by the kernel.
9. It wakes up every ten mintues to post a time-stamp entry and poll the states of the system's processors.

The ERRFORMAT job is activated by the kernel, as described in Section 7.1.3.2, when any of the following conditions is true:

- The job is created by the start-up job.

- The number of posted EMBs exceeds the threshold value in KER$GW_MAX_POSTED.
- The time-stamp interval expires.

When reawakened, ERRFORMAT first determines the cause. If the time interval expired, a time-stamp log entry is posted with a call to the KER$POST_ERRORLOG procedure, and the running processor's status is polled with a call to KER$POLL_MACHINE. KER$POLL_MACHINE polls for processor-specific error conditions. Any errors detected are handled on a processor-specific basis.

If the event was signaled because the EMB threshold was reached, the event object KER$GL_ERRFMT_WAKEUP is first cleared for the next wait. ERRFORMAT then runs down the queue of posted log entries and writes a record for each to the log file or over the network. After an entry is dispatched, ERRFORMAT returns the processed EMB to the queue of available buffers. When all the buffers have been written out and returned to the available queue, ERRFORMAT again waits, and the loop continues for the life of the system.

If entries are to be logged to a local disk, the log file must be opened. If that file cannot be opened, the logged entries are lost. When the file is being created, the first record written is a new-file-creation entry. The subsequent log entries are appended to the end of the file. When the last record has been written to disk, the file is closed.

If entries are to be logged remotely, ERRFORMAT must first initiate the network link to the remote error-logging server (ELSE) by creating a DECnet virtual circuit and establishing a communication link with the ELSE server. ELSE must be running before the attempt to establish the circuit is made. If the circuit connection fails, then status is returned to indicate that the remote error-logging link cannot be established.

Once the circuit to ELSE has been established, log entries are processed by creating a VAXELN message, transferring the contents of the EMB to the message buffer, and sending the message with the KER$SEND procedure. If the message cannot be sent, the message is deleted, and the appropriate error status is returned, which will result in the log entry being lost.

7.2 Bugcheck Handling

A bugcheck is declared whenever the kernel or a job detects an internal inconsistency, such as a corrupted data structure or an unexpected exception. Bugchecks can be fatal or nonfatal and can be systemwide in scope or affect only a single process. Nonfatal bugchecks result in an error-log entry being posted and the system or process continuing execution. Fatal process-level bugchecks can force the process to exit, whereas fatal system bugchecks can result in an orderly shut-down of the system.

The kernel bugcheck mechanism is invoked through the execution of the BUG_CHECK marco. For example, if the kernel cannot create the start-up job during system initialization, it forces the system to shut down by issuing the following fatal system bugcheck:

```
BUG_CHECK       CRESTARTUP,FATAL
```

The bugchecks that the kernel uses for its own purposes are issued at an elevated IPL in kernel access mode.

The BUG_CHECK macro, which takes a bugcheck reason code and severity as its arguments, generates the VAX opcode FF, which results in a reserved or privileged instruction fault (SS$_OPCDEC, opcode reserved to DIGITAL), causing control to be transferred to the exception service routine KER$DIGITAL_RESERVED (in module EXCEPTION). This routine determines whether the operand specifier is either FE, for a word-length bugcheck reason code, or FD, for a longword-length reason code. KER$DIGITAL_RESERVED then transfers control to the kernel's bugcheck handler, the KER$BUG_CHECK subroutine in module BUGCHECK.

KER$BUG_CHECK performs several steps, depending on the access mode in which the bugcheck occurred, the IPL at the time of the bugcheck, and the severity level in bits <2:0> of bugcheck reason code. The combination of these factors determine whether the bugcheck is fatal.

Four kinds of bugchecks are possible, with the following results:

- Nonfatal process-level bugcheck. The access mode can be user or kernel, IPL is 0, and the severity code is nonfatal. The bugcheck is logged, and the process is allowed to continue execution at the instruction following the bugcheck invocation.

- Fatal process-level bugcheck. The access mode can be user or kernel, IPL is 0, and the severity code is fatal. The bugcheck is logged, and the KER$EXIT procedure is called to force the process to exit with KER$_BUGCHECK exit status.
- Nonfatal system bugcheck. The access mode is kernel, the IPL is greater than 0, and the severity code is nonfatal. The bugcheck is logged, and the system is allowed to continue execution at the instruction following the bugcheck invocation.
- Fatal system bugcheck. The access mode is kernel, the IPL is greater than 0, and the severity code is fatal. The system is shut down as follows:
  1. A 512-byte bugcheck data block containing the reason and hardware context for the crash is constructed.
  2. Information describing the bugcheck is written to the console.
  3. If the system dump facility is enabled, the bugcheck information, crash-restart data, and any posted error-log buffers are written to the local dump device.
  4. Execution enters a closed loop at IPL 31 to halt the normal operation of the system.

For fatal system bugchecks, a crash-restart error-log entry is constructed. It contains the bugcheck reason code and the contents of the general and internal processor registers and the processor-specific registers. The crash-restart log entry record is built in a preallocated EMB in system address space at the location pointed to by the global value KER$GA_CRASHLOG, since this is the only address space from which I/O can be performed at this point.

In a multiprocessing configuration, each processor participates in the fatal system bugcheck sequence by independently saving its individual processor context. However, after an individual processor saves its context and its identifier, it signals the other processors to shut down, leaving only one to complete the shut-down of the system. Since it is not known whether any remaining processors are capable of acknowledging the request to shut down, a ten-second wait is executed, after which it is assumed that the shut-down of all other processors is complete and the crash sequence can continue.

In the final moments before the system is halted, the bugcheck logic outputs last-fail information to the system console terminal. The last-fail information consists of the following items:

- The text of the message giving the reason for the bugcheck

- The name of the job that was running when the bugcheck occurred and the value of its job port (if any)
- The contents of the general processor registers
- The contents of the kernel or interrupt stack (if possible)
- The final shut-down message

Following the console dump, several checks are made to determine whether the dump facility is enabled, whether the dump device and its dump control block (DCB) are valid, and whether the dump device has been initialized. If all these tests are successful, the virtual address of the dump file I/O buffer is calculated, and a dump file header is initialized and written to the dump file. Next, the bugcheck and crash-restart information for the specific processor (or processors in a multiprocessing system) is written to the dump file. Finally, any posted EMB entries that have not been output to the error-log device are written to the dump file, and the dump device is disconnected.

At this point, a check is made to see whether the currently executing processor is the processor that crashed. If it is, a final system shut-down message is output to the console. In any event, the processor then enters an infinite loop at IPL 31 until the system is manually restarted.

7.3 Machine-Check Handling

A machine check is an exception that results when the processor detects an internal error in itself. VAXELN machine-check handling is supported by a series of subroutines in the kernel that are executed when a machine check occurs. These routines, called machine-check handlers, are entered through vectors in the system control block (SCB).

The initial processing of a machine check depends on the processor type. The goal of a machine-check handler is to keep as much of the system running as possible. To accomplish this goal, the machine-check handler must evaluate several pieces of information that determine how serious a specific machine check is: the nature of the machine check itself and the access mode and interrupt priority level (IPL) at which the machine check occurred.

The following sections provide an overview of machine-check handling under VAXELN and describe the mechanism used to protect the system from fatal machine checks during certain operations.

7.3.1 Machine-Check Handlers

The machine-check handler for each processor type or class is included when the kernel image for that type or class is created. In addition to machine checks initiated through the machine-check hardware vector, there may be other hardware error vectors that can be serviced by the machine-check handler. These errors include correctable read data errors (CRDs), memory errors (bus errors, uncorrectable ECC error, nonexistent memory), or bus adapter errors (for example, NBIA errors on VAX 8000-series processors). The individual modules (MCHECK*nnn*) supply detailed information on the specific operation of a given machine-check handler.

In general, machine-check handlers process machine checks and other hardware errors as follows:

1. A check is made to see whether a machine-check recovery block is in effect. If so, a machine-check recovery sequence is invoked (see Section 7.3.2 for a description of this recovery mechanism). If no recovery block is in effect, or if error logging is not inhibited by the recovery block function mask, one or more error-log entries are posted.
2. If the machine check occurred at user IPL (0) in either access mode, the error is reflected back to the process as a machine-check exception condition as follows:
   a. A PSL and PC at the time of the machine check are copied onto the kernel stack. These value will become the exception PSL and PC in the exception signal array.
   b. Control is transferred, in kernel mode, to the machine-check exception service routine KER$MCHECK in module EXCEPTION.
   c. KER$MCHECK completes the creation of the signal array by pushing the signal name, KER$_MACHINECHK, and the argument count, 3, onto the stack.
   d. Control is transferred to the system's uniform condition-handling mechanism at location KER$REFLECT in module EXCEPTION, which reports the machine-check exception to the offending process (see Section 6.6, Uniform Condition Dispatching).

3. If the machine check is unrecoverable and occurred in kernel mode at an elevated IPL, it is considered fatal. A further determination must be made: whether to take a fatal system bugcheck or to attempt to recover with a machine-check recovery block. If no recovery block is in effect, a fatal system bugcheck is taken, and the system begins an orderly shut-down.
4. Depending on the processor type, a record is kept of the interval between occurrences of certain machine checks. If these machine checks begin to occur too rapidly (that is, they exceed a minimum threshold, usually one every second), the system is considered to be "out of control," and a fatal system bugcheck is taken.

7.3.2 Machine-Check Recovery: KER$MACHINECHK_PROTECT

By establishing a machine-check recovery block, code running in kernel mode at an elevated IPL can avoid taking a fatal system bugcheck when a machine check occurs. A recovery block is a section of code protected by a special kernel mechanism. Some of the kernel's processor-specific initialization code uses this mechanism to intercept machine checks that are raised when an I/O bus is probed for adapters that may not be present.

To be protected by a recovery block, an assembly-language routine must pass two arguments to the KER$MACHINECHK_PROTECT subroutine (in module ERRORLOG):

- The address of the instruction following the protected block of code
- A mask that defines what functions are allowed in the event of a machine check (for example, inhibit error logging or nonexistent memory)

For example, the routine to configure I/O address space on VAX 8000-series processors (in module INIT8NN), protects its probing of the bus by calling KER$MACHINECHK_PROTECT, as shown in Figure 7–1. The kernel macro MCHKPRTCT_INIT generates the subroutine call to KER$MACHINECHK_PROTECT, passing it the function mask and the address of the instruction following the protected code in general registers. The protected code itself is a single MOVL instruction that attempts to read a CSR on an NBIA adapter that may or may not exist. If the adapter does not exist, a machine check is raised, and the recovery mechanism is activated. The kernel macro MCHKPRTCT_END marks the end of the protected code by providing a Return from

Subroutine (RSB) instruction and the actual label to mark the restart of the unprotected code.

The function mask specifies parameters for the machine-check recovery. Table 7–5 shows the kernel symbols (defined in module KERNELDEF) for the function mask and their effect when passed to KER$MACHINECHK_PROTECT.

Table 7–5: Machine-Check Recovery Function Masks

Function Symbol	Meaning
MCHK$M_LOG	Inhibit logging of this error.
MCHK$M_MCHK	Protect against machine checks.
MCHK$M_NEXM	Protect against nonexistent memory errors.
MCHK$M_ADAPTER	Protect against adapter error interrupts.

Figure 7–1: The Use of KER$MACHINECHK_PROTECT

```
MCHKPRTCT_INIT  B^20$,-          ; Protect against machine checks
                #<MCHK$M_NEXM!MCHK$M_MCHK!MCHK$M_LOG>
        pushal  B^20$
        pushl   #<MCHK$M_NEXM!MCHK$M_MCHK!MCHK$M_LOG>
        jsb     G^KER$MACHINECHK_PROTECT
MOVL    NBIA$L_CSR1(R10),R3      ; Protected code: read CSR 1
MCHKPRTCT_END   20$              ; End of protected code
        rsb
        20$:
BLBC    R0,30$                   ; If lbc, nothing at this NBIA,
                                 ; that is, a machine check occurred
```

KER$MACHINECHK_PROTECT provides protection from fatal machine checks as follows:

1. The address of the machine-check recovery block for the current processor is obtained from the processor's machine-check data block.

2. The function mask argument is copied to the recovery block. This mask will be used by the machine-check handler if the protected code causes a machine check.
3. Interrupts are disabled by raising the IPL to 31.
4. The protected code is called back as a subroutine. It uses the return PC of the protected code that was pushed onto the stack by the subroutine call to KER$MACHINECHK_PROTECT.
5. The protected code executes as a subroutine.
6. If the protected code does not generate a machine check, it executes the RSB instruction generated by the MCHKPRTCT_END macro, and control returns to KER$MACHINECHK_PROTECT, which cleans up the stack and returns status to the caller at its previous IPL. Execution then continues at the return address specified in the call to KER$MACHINECHK_PROTECT — the instruction following the protected code. The status value returned indicates whether a machine check occurred while the recovery block was in effect.

If the protected code generates a machine check, control is vectored by the processor through the SCB to the machine-check handler. The handler determines that a machine-check recovery block is in effect, clears the machine check, and transfers control to the subroutine KER$MACHINECHK_BUGCHK, which performs the following operations:

1. The function mask is examined to determine that recovery from a fatal machine check was enabled in the call to KER$MACHINECHK_PROTECT.
2. The stack is unwound to its state before the subroutine call to KER$MACHINECHK_PROTECT. The call frame for the previous call, the machine-check frame, and the stack arguments for the original subroutine call to KER$MACHINECHK_PROTECT are all cleared from the stack.
3. The access mode and IPL of the caller are restored.
4. The machine-check code is saved for inspection by the caller's code.
5. Control is returned to the code following the protected code block.

The code following the protected block may then continue and take whatever action is required after determining the cause of the machine check (if any action is required at all). The result is that the protected code block survives a potentially fatal machine check. The survival of the protected code guarantees the execution of some critical function for

the application or enables it to handle peculiarities of special hardware that would normally generate a machine check.

Chapter 8

Kernel Procedures and Procedure Dispatching

Most of the operations that the VAXELN Kernel performs at the request of VAXELN processes are implemented as procedures called kernel procedures. The majority of these procedures are the public, KER$ procedures described in the VAXELN documentation. Others are internal, private, procedures invoked on behalf of user jobs by the kernel or other system components, such as the debugger and the error-logging subsystem.

A call to a public or internal kernel procedure transfers control of execution to a small procedure in the kernel called a kernel vector. The code in the vector takes a minimal number of steps to transfer control to the actual code of the requested kernel procedure. When control returns to the vector after the execution of the procedure, the vector code returns procedure and status values and control to the caller.

All but a few of the kernel procedures execute in kernel access mode, allowing them to manipulate data structures protected from access by jobs running in user mode (VAXELN employs only these two of the four modes defined by the VAX architecture). Control is transferred from a kernel-mode procedure's kernel vector to the procedure code itself through the kernel's change mode to kernel (CHMK) dispatcher. The majority of these kernel-mode procedures also execute at elevated interrupt priority level (IPL) to synchronize access to job and system data structures.

A handful of kernel procedures execute in the mode of the caller, which may or may not be in kernel mode. The kernel vectors for these procedures dispatch control with subroutine or branch instructions, which do not change the access mode of the caller. Nor do these procedures alter processor IPL.

The kernel also provides a procedure, KER$ENTER_KERNEL_CONTEXT, to allow user-mode programs to execute kernel procedures that require their caller to be in kernel access mode.

This chapter describes the following topics related to the dispatch of kernel procedures:

- The format and use of kernel vectors are described in Section 8.1.
- The flow of control in kernel procedure dispatching is described in Sections 8.2 and 8.3. The first of these sections deals with procedures that execute in kernel access mode and the second with those that execute in the caller's mode.
- The return of control and procedure status values to the caller is described in Section 8.4.
- The function of the KER$ENTER_KERNEL_CONTEXT change-mode procedure is described in Section 8.5.

8.1 Kernel Vectors and Procedure Entry Points

As described briefly in Section 2.3, kernel procedures are entered through a series of vectors that comprise their public entry points. This block of kernel vectors appears at the beginning of the kernel image portion of a system image, following the system image header, if one exists. The vectors reside at the same locations in each release of the VAXELN software so that user programs will not have to be relinked with each new version of the kernel.

The kernel vectors are defined in module VECTORTAB. When the kernel image is built, VECTORTAB is assembled with both the SYSVECTOR module and the DISPATCH module. In the first instance, VECTORTAB builds the series of kernel vectors; in the second, it builds the dispatch table used by the CHMK exception service routine described in Section 8.2.

The kernel vectors represent the entry points for all publicly accessible kernel procedures. When a user program codes a call to a kernel procedure, the language processor usually generates a CALL instruction specifying the public entry point for the procedure — the documented KER$ procedure name. This is the entry point for the called procedure's kernel vector, not for the body of the procedure itself. When control is passed to the vector through the CALL instruction, the vector ultimately transfers control to the location of the actual procedure code within the body of the kernel.

A CALL instruction can be generated by a language processor, such as the VAXELN Pascal or VAX C compiler, or can appear in VAX assembly language code — the kernel calls its own procedures in this manner. For example, a VAXELN Pascal call to the RECEIVE procedure generates the following VAX instruction:

```
CALLS   #7,KER$RECEIVE
```

At run time, the execution of this instruction causes the processor to build a call frame on the caller's stack and transfer control to the global location KER$RECEIVE in the kernel. That location is the entry point of the kernel vector for the KER$RECEIVE procedure.

A kernel vector is in fact a miniature procedure and therefore begins with a procedure entry mask. Figure 8–1 shows the general structure of a kernel vector.

Figure 8–1: Structure of a Kernel Vector

```
KER$procedure::                         ; entry point for vector
        .WORD   entry-mask              ; mask of registers to be saved
        CHMK    #procedure-code         ; pass control to CHMK service
        MOVL    R1, @8(AP)              ; return up to 3 values to caller
        [MOVL   R2, @12(AP)]            ; [these instructions appear
        [MOVL   R3, @16(AP)]            ;  in vectors that require them]
        BRW     KER$RETURN_STATUS       ; branch to common exit path
                                        ; to return procedure status
```

In module SYSVECTOR, the procedure entry mask is generated using a global symbol in the form KER$*procedure*_M. This symbol is generated when the actual procedure code is assembled. For example, the symbol KER$RECEIVE_M appears as the word entry mask in the KER$RECEIVE kernel vector and represents the following list of registers to be saved when the procedure is called:

```
^M<R2,R3,R4,R5,R6,R7,R8,R9,R10,R11>
```

Along with the entry mask symbol, a symbol in the form KER$*procedure*_C is generated to represent the expected or minimum argument count for the procedure. For example, the KER$RECEIVE procedure expects seven arguments; therefore, the symbol KER$RECEIVE_C translates to the number 7. This value is used later in procedure dispatching to verify that the correct number of arguments has been passed in the procedure call. Some procedures, such as KER$WAIT_ANY, take a variable number of arguments; these procedures must then verify that the required minimum number of arguments has been specified.

The execution of the CHMK (Change Mode to Kernel) instruction causes the processor to generate an exception that will be handled by a special routine, in which the procedure-specific CHMK code is used to determine where control will be passed to begin execution of the actual procedure code. Upon return from that code, many vectors pass procedure values back to the caller using, MOVL instructions from general registers R1 through R3 to writeable locations pointed to by address variables in the caller's argument list. Once values have been returned, the vector passes control to a common exit path used to return the procedure status and control of execution to the caller.

Three types of kernel vectors reside in the vector block, as follows:

- Vectors that invoke a kernel procedure in kernel mode via a CHMK instruction. This group comprises the vast majority of kernel vectors and is exemplified by the vector shown in Figure 8–1. These vectors are entered with a CALL instruction.
- Vectors that invoke a kernel procedure in the caller's mode by transferring control via a BSBW (Branch to Subroutine with Word Displacment) or JSB (Jump to Subroutine) instruction. No CHMK instruction is used. These vectors are also entered with a CALL instruction.
- Vectors that invoke a kernel routine in the caller's mode by transferring control by a BRW (Branch with Word Displacement) or JMP (Jump) instruction. No CHMK instruction is used. These vectors are entered with a BSBW or JSB instruction.

The remainder of this chapter describes how control is passed to procedures that operate in kernel mode and procedures that operate in the caller's mode. The kernel vectors that transfer control to each kind of procedure are described in more detail, as are the techniques used to return control, status, and values to the caller from the kernel procedure.

8.2 Dispatch to Procedures That Execute in Kernel Mode

All but a few of the kernel procedures execute in kernel mode and are entered through change-mode vectors in the kernel's vector block. Change-mode vectors are entered by a CALL instruction and transfer control to an actual kernel procedure code through a CHMK instruction. All change-mode vectors use the vector model shown in Figure 8–1.

Executing a CHMK instruction generates an exception. The CHMK instruction microcode changes access mode to kernel and pushes the value of the processor status longword (PSL), the value of the program counter (PC) for the next instruction, and a specified CHMK code operand onto the kernel stack. For example, the execution of the instruction

```
CHMK    #5
```

pushes a PSL, the PC of the instruction following the CHMK instruction, and a 5 onto the kernel stack. Control is then passed to the exception service routine whose address is located in the CHMK entry in the system control block (SCB).

For VAXELN, the service routine for CHMK exceptions is KER$KERNEL_SERVICE in module DISPATCH. The module begins with a table — an array of bytes — containing the required (or minimum) argument counts for the kernel procedures; this count table is generated during assembly using the KER$*procedure*_C symbols. The table is indexed by the CHMK code number passed to the procedure on the kernel stack.

The CHMK dispatcher plays a central part in the overall flow of control from a CALL to a kernel procedure to the return of status and procedure values to the caller. Figure 8–2 illustrates this flow of control in dispatching a kernel procedure call. Execution occurs as follows:

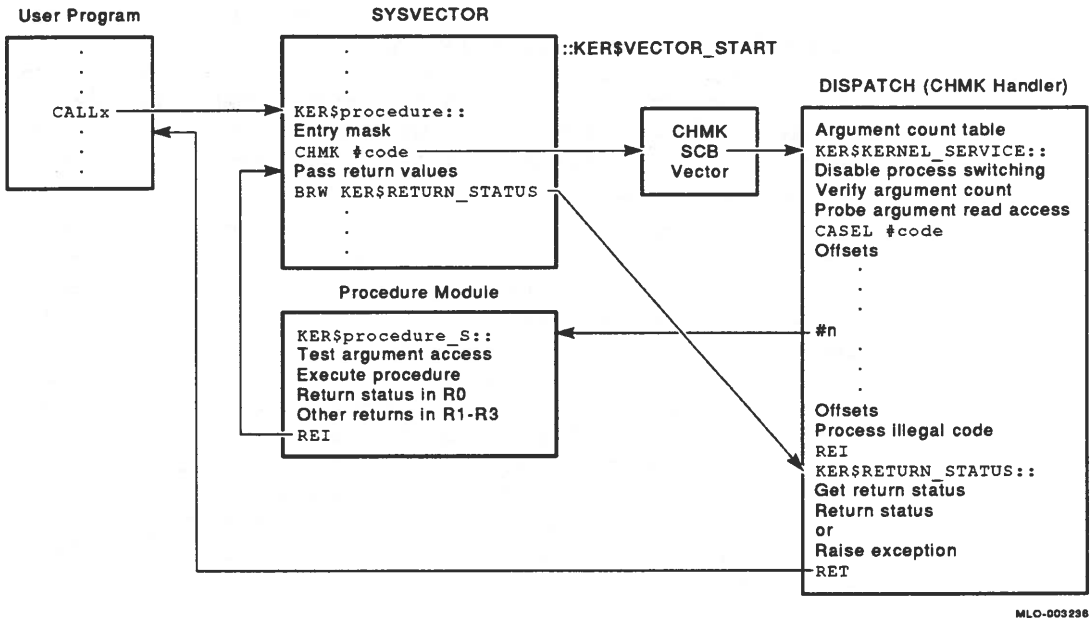
1. A CALL instruction transfers control to the kernel vector for the called procedure, located in module SYSVECTOR.

2. Upon entry into the appropriate vector, the general registers specified by the procedure's entry mask are saved, and the CHMK instruction is executed, placing the caller into kernel mode and pushing the CHMK code for the procedure onto the kernel stack.
3. The execution of the CHMK instruction causes the VAX hardware to transfer control to the CHMK dispatcher, KER$KERNEL_SERVICE, whose address appears in the CHMK exception vector in the SCB.
4. The CHMK dispatcher verifies the argument count and accessibility of the argument list and transfers control to the KER$*procedure*_S entry point via the CASEL (Case Longword) instruction.
5. The KER$*procedure*_S procedure body performs the requested service and moves the procedure status into R0 and any other procedure return values into R1 through R3, as necessary. It then executes an REI (Return from Exception or Interrupt) instruction to return to the caller's mode at the instruction following the CHMK in the kernel vector.
6. Any procedure return values are copied from R1 through R3 into the user's address space via pointers passed in the argument list, as described in Section 8.4.
7. Control branches to the KER$RETURN_STATUS procedure, described in Section 8.4. This procedure evaluates the procedure's return status and executes a RET (Return) instruction to return control to the instruction following the CALL instruction in the user's code.

Figure 8–3 shows relevant portions of the kernel's CHMK dispatcher, which executes as follows when it is entered through the SCB CHMK exception vector:

1. The kernel macro DISABLE_SWITCH inhibits context switching of the processes within the current job by elevating the processor interrupt priority level (IPL) to IPL$K_DISABLE_SWITCH (3). Disabling context switching prevents another process in the job from deleting or corrupting memory required for the execution of the called kernel procedure. Context switching is again permitted when the caller's IPL is restored through the execution of an REI instruction at the conclusion of the kernel procedure code.

Figure 8–2: Control Flow in Dispatching Kernel Procedures That Use Kernel Mode



2. The CHMK code is popped from the top of the kernel stack and compared to the maximum acceptable code. If the current code is greater than the maximum, control branches to the CASEL instruction in step 5. This illegal code will fall through the case statement to a location where the status value KER$_NO_SUCH_SERVICE is set, and an REI instruction is executed to return control to the kernel vector code in the caller's access mode.
3. The number of arguments passed to the procedure is obtained from the CALL argument list, and the argument count is compared to the appropriate KER$*procedure*_C value in the argument table, which is indexed by the CHMK code. If too few arguments are supplied, control is transferred to a location where the status value KER$_BAD_COUNT is set, and an REI instruction is executed to return control to the kernel vector code in the caller's access mode. The actual argument count can exceed the value in the argument-count table, because certain procedures, such as KER$WAIT_ANY, take a variable number of arguments.

4. The accessibility of the argument list to kernel-mode readers is tested by using the IFN_READ macro, which generates a PROBER (Probe Read Access) instruction. If the argument list is not readable, control is transferred to a location where the status value KER$_NO_ACCESS is set, and an REI instruction is executed to return control to the kernel vector code in the caller's access mode. This test confirms that access to the argument list is possible; however, it does not confirm that the memory to which pointer arguments point is accessible. This test is the responsibility of the kernel procedure itself.
5. A CASEL instruction dispatches control to the actual instructions for the requested service. Within the kernel, the internal entry points for the kernel procedures are represented by global locations in the form KER$*procedure*_S. For example, the procedure body for the KER$RECEIVE procedure begins at the global location KER$RECEIVE_S. The table associated with the CASEL instruction is a series of 16-bit words, indexed by the CHMK code, containing the byte offsets to each KER$*procedure*_S location. (Within the present structure of the kernel, none of the offsets to the procedures exceeds the 32,767-byte maximum offset supported by the CASE instruction, so no special processing is required for procedures located beyond this offset, as is true in the VMS executive.)

Two procedures, KER$WAIT_ALL and KER$WAIT_ANY, contain special tests in their respective kernel vectors. When control returns to these kernel vectors from the body of the WAIT procedures, the status value in R0 is tested. If the value is 0 — meaning that the calling process has an asynchronous exception pending against it or that the process's wait was interrupted by the delivery of an asynchronous exception — the vector code branches back and reexecutes the CHMK instruction to attempt to reestablish the wait for the calling process. When the procedure finally returns a nonzero value in R0, the branch to KER$RETURN_STATUS is made. See Chapter 11 for more on KER$WAIT_ANY and KER$WAIT_ALL procedures.

Figure 8–3: CHMK Dispatch — KER$KERNEL_SERVICES

```
ARGUMENT_TABLE:                          ; Array of bytes with argument
        .                                ; counts, indexed by CHMK code
        .
        .
        .BYTE #KER$procedure_C           ; Argument count for nth procedure
        .
        .
        .

KER$KERNEL_SERVICES::                    ; Begin CHMK dispatcher code
        DISABLE_SWITCH                   ; Prevent context switching within job
                mtpr    #IPL$K_DISABLE_SWITCH,#PR$_IPL

        MOVZBL  (AP),R1                  ; Get number of arguments passed
        MOVL    (SP)+,R0                 ; Pop vector number (CHMK code) to R0
        CMPL    R0,#LIMIT                ; Is the code within range
        BGTRU   CASE                     ; If not, don't test argument count
        CMPB    R1,W^ARGUMENT_TABLE[R0]  ; Enough arguments?
        BLSSU   BAD_COUNT                ; Return KER$_BAD_COUNT and REI
        ASHL    #2,R1,R1                 ; Compute number of bytes in argument list
        IFN_READ R1,4(AP),10$            ; If not readable, return KER$_NO_ACCESS and REI
                prober  #0,R1,4(AP)
                beql    NO_ACCESS

CASE:   CASEL   R0,#0,#LIMIT             ; Dispatch to correct kernel service
        .
        .
        .
        offset to KER$procedure_S        ; #n -- transfer control to procedure body
        .
        .
        .
        RETURN_STATUS NO_SUCH_SERVICE    ; Fall through to illegal code handler
                movzwl  #KER$_NO_SUCH_SERVICE,R0
                rei
```

8.3 Dispatch to Procedures That Execute in the Caller's Mode

A small number of kernel procedures — some of them public, some internal — execute in the mode of the process from which they are called. Execution of a kernel procedure in the mode of the caller may be desirable or necessary for one or more of the following reasons:

- The called procedure does not require kernel mode for its execution. Example: the procedure KER$GET_TIME.

- The called procedure must execute in the mode of the caller. Example: the procedure KER$RAISE_EXCEPTION.
- The caller must be in kernel mode even before calling the kernel procedure. Examples: the kernel procedure KER$SIGNAL_DEVICE and the internal procedure KER$INIT_ERLHEADER.
- The caller is executing in system context on the interrupt stack and therefore cannot use the CHMK instruction. Example: the kernel procedure KER$SIGNAL_DEVICE, which is called only from device interrupt service routines.

Some of the routines in this class are invoked with a CALL instruction, as are the procedures that change mode; others are invoked as subroutines with a BSBW or JSB instruction. Moreover, the structure of the kernel vector for a procedure that executes in the caller's mode depends on the method of invocation. The following sections describe these two forms of dispatch.

8.3.1 Routines Invoked with a CALL Instruction

The kernel vectors for procedures that execute in the caller's mode and that are invoked with CALL instructions transfer control to the actual procedure code using a subroutine call instruction, either BSBW or JSB. Figure 8–4 illustrates the general structure of such a kernel vector. The use of the branch or the jump instruction depends on the byte displacement to the subroutine entry point.

Figure 8–4: Structure of a Kernel Vector for Caller-Mode Procedures Invoked with a CALL Instruction

```
KER$procedure::                          ; entry point for vector
        .WORD   entry-mask               ; mask of registers to be saved
        BSBW/JSB KER$procedure_S         ; transfer control to _S entry point
        MOVL    Rn, @n(AP)               ; return values to caller...
        BRW     KER$RETURN_STATUS        ; branch to common exit path
                                         ; to return procedure status
```

By bypassing the CHMK dispatcher, the BSBW or JSB subroutine instruction transfers control to the KER$*procedure*_S procedure in the mode of the caller. The body of the procedure, which is entered as a subroutine, must test the argument count and the accessibility of the argument list; for procedures that change mode, these tests are performed by the CHMK dispatcher.

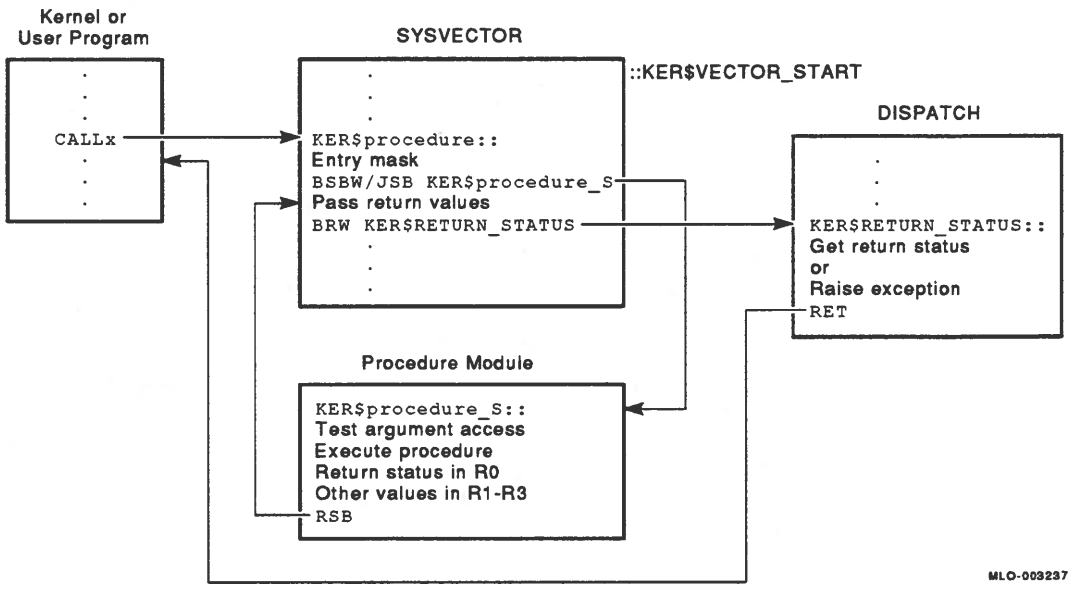
Figure 8–5 illustrates the flow of control in dispatching control to a kernel procedure call using this subroutine-call vector. Execution occurs as follows:

1. A CALL instruction transfers control to the kernel vector for the called procedure (defined in module SYSVECTOR).
2. Upon entry into the vector, the general registers specified by the procedure's entry mask are saved and the BSBW or JSB instruction is executed, pushing the return PC onto the current stack and transferring control to the KER$*procedure*_S entry point.
3. The KER$*procedure*_S procedure body performs the requested service and moves the procedure status into R0 and any other procedure values into R1 through R3, as necessary. It then executes an RSB (Return from Subroutine) instruction to return to the instruction following the BSBW or JSB in the kernel vector.
4. Any procedure return values are copied from R1 through R3 into the user's address space via pointers in the argument list, as described in Section 8.4.
5. Control branches to the KER$RETURN_STATUS procedure, described in Section 8.4. This procedure evaluates the procedure's return status and executes a RET instruction to return control to the instruction following the CALL in the user's code.

8.3.2 Routines Invoked with a Subroutine Instruction

The kernel vectors for procedures that execute in the caller's mode and that are invoked with a subroutine instruction transfer control to the actual procedure code using a VAX branch instruction, either BRW or JMP.

Figure 8–5: Control Flow in Dispatching Kernel Procedures That Use the Caller's Mode: CALL Invocation



As a rule, kernel routines invoked in this manner share the following characteristics:

- They are internal to the kernel and cannot be called directly from higher-level languages.
- Since they are not invoked with a CALL instruction, they return status and values directly to the caller through general registers.
- Since these kernel routines are invoked by a subroutine instruction from the caller's code, the kernel vector for the routine is coded as a subroutine rather than as a procedure; that is, it contains no procedure entry mask and no RET instruction.

Figure 8–6 illustrates the general structure of a kernel vector for a kernel routine invoked with a subroutine call.

Figure 8–6: Structure of a Kernel Vector for Caller-Mode Procedures Invoked with a Subroutine Instruction

```
KER$procedure::                         ; entry point for vector
        BRW/JMP KER$procedure_S         ; transfer control to _S entry point
                                        ; return via RSB in routine code
```

The use of the BRW or JMP instruction transfers control to the KER$*procedure*_S procedure in the mode of the caller. Because the routine is normally invoked internally by other kernel code, tests for the correctness of the invocation are usually less stringent than they are for routines that can be called publicly.

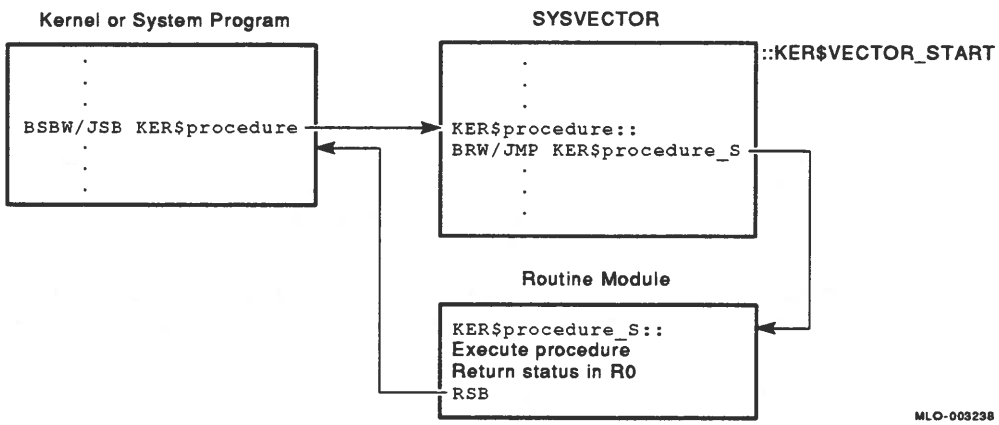
Figure 8–7 illustrates the flow of control in dispatching control to a kernel routine using this branching kernel vector. Execution occurs as follows:

1. A BSBW or JSB subroutine instruction transfers control to the kernel vector for the routine in SYSVECTOR.
2. Upon entry into the vector, the BRW or JMP instruction transfers control to the KER$*procedure*_S entry point.
3. The KER$*procedure*_S routine performs the requested service and moves the procedure status into R0 and any other return values into other general registers. It then executes an RSB instruction to return directly to the instruction following the BSBW or JSB in the code of the caller (and not to the kernel vector).

8.4 Return of Kernel Procedure Values and Status

Kernel procedures that are invoked by a CALL instruction return their values through MOVL instructions within the procedure's kernel vector. They return a status value to the caller with a common routine, KER$RETURN_STATUS. The following sections describe the return of procedure values and completion status.

Figure 8–7: Control Flow in Dispatching Kernel Routines That Use the Caller's Mode: Subroutine Invocation



8.4.1 Return of Procedure Values

Kernel procedures that are invoked by a CALL instruction cannot return procedure values through the general registers, because the registers used by the procedure are destroyed when a RET instruction restores the caller's register contents from the procedure call frame. For this reason, the kernel vectors for procedures that return values contain instructions to copy values from the procedure's registers into the caller's argument variables, which have been passed to the procedure by reference.

For example, the KER$RECEIVE procedure returns three values to its caller: the identification of the message object used to receive the message, the address of the message buffer, and the size in bytes of the received message. To accomplish this, the procedure code — KER$RECEIVE_S — determines these values, writes them to registers R1, R2, and R3, respectively, and executes an REI instruction to return to the kernel vector code following the procedure dispatch instruction. The following MOVL instructions in the KER$RECEIVE vector then copy procedure values from the registers, which will be destroyed when control is returned to the caller, to the caller's address space:

```
MOVL    R1,@8(AP)
MOVL    R2,@12(AP)
MOVL    R3,@16(AP)
```

When KER$RECEIVE was called, the caller supplied three writeable variables in its argument list to receive the values returned by the procedure. The addresses of those variables — pointers to the variables — appear in the argument list at the second, third, and fourth positions, located at offsets of 8, 12, and 16 bytes from the argument pointer (AP) passed to the kernel procedure by the CALL instruction. The MOVL instructions simply copy the procedure values in the registers to the locations of the caller's writeable variables, represented in the argument list as operands in the form @*n*(AP) (displacement deferred mode). When control returns to the caller, it will find the returned procedure values in those variables, and its general registers will be restored to their state before the kernel procedure call. (The return of a value from R0 to @4(AP) is reserved for procedure status values.)

8.4.2 Return of Status Values

The status of a kernel procedure is returned to its caller in a writeable variable optionally supplied as an argument to the procedure. If a status variable is supplied, the procedure status is simply written to that variable — the responsibility for ascertaining the success of the procedure call belongs to the caller. If no status variable is supplied, and if the procedure completes unsuccessfully, the kernel raises an exception against the calling process based on the completion status of the procedure. The evaluation of the procedure status is handled not by the kernel procedure itself but by the common exit path for all CALLed kernel procedures, the routine KER$RETURN_STATUS in module DISPATCH.

Control is transferred to KER$RETURN_STATUS from the last instruction in the kernel vector — BRW — executed after the procedure code has completed and all procedure values have been written back to the caller. The purpose of this common exit routine is to examine the status value written to general register R0 by each kernel procedure, then to take the appropriate action based on this value and on whether the status variable was present in the procedure call. The address of this status variable is passed to the called procedure as the first argument in its argument list and is referenced as the operand 4(AP). If the caller specified a status variable, the status argument contains the address of the variable; otherwise, the argument equals 0. Figure 8–8 shows the code for KER$RETURN_STATUS, which executes as follows:

1. The address of the caller's status variable is obtained from the status argument.
2. If the status argument is nonzero, then a status variable was specified by the caller. Therefore, the status value for the procedure is copied to the caller's status variable, and a RET instruction is executed to return to the caller, removing the call frame from the stack.
3. If no status variable is present (4(AP) equals 0), then the status argument is evaluated. If the lower bit in R0 is set, the procedure was successful, and a RET instruction is executed to return to the caller and remove the call frame from its stack.
4. If the lower bit in R0 is clear, then the procedure did not succeed. Therefore, the status value and a zero (representing a null status variable) are pushed onto the stack as arguments to the KER$RAISE_EXCEPTION kernel procedure, which is called to signal an exception against the caller in the caller's access mode. The final RET instruction will be executed only if the caller's process is able to handle the exception raised by the call to KER$RAISE_EXCEPTION.

Figure 8–8: Common Procedure Exit Code: KER$RETURN_STATUS

```
KER$RETURN_STATUS::
        MOVL    4(AP),AP            ; get the status return address
        BNEQ    10$                 ; if nonzero, a status variable exists
        BLBC    R0,20$              ; failure status -- raise an exception
        RET                         ; return to caller with success

10$:    MOVL    R0,(AP)             ; status variable present -- copy status there
        RET                         ; return to caller

20$:    PUSHL   R0                  ; push the status value
        PUSHL   0                   ; don't specify a status variable for this call
        CALLS   #2,W^KER$RAISE_EXCEPTION
                                    ; raise an exception against the caller
        RET
```

8.5 Change-Mode Service for User-Mode Jobs — KER$ENTER_KERNEL_CONTEXT

A number of kernel and run-time library procedures require that their callers execute in kernel mode to call the procedure. The use of kernel mode is required for procedures that interact closely with critical data structures within the kernel, protecting them from corruption by ordinary jobs within the system. On occasion, however, the need arises for a program to call one of these kernel-mode procedures while otherwise executing exclusively in user mode. The kernel procedure KER$ENTER_KERNEL_CONTEXT gives a program the means to temporarily elevate its access mode to kernel.

For example, a user-mode job may want to allocate a buffer in system virtual address space using the KER$ALLOCATE_SYSTEM_REGION procedure, which normally requires its caller to execute in kernel mode. By calling KER$ENTER_KERNEL_CONTEXT and specifying KER$ALLOCATE_SYSTEM_REGION and the address of the argument list for the procedure as its arguments, the user-mode caller can execute this single procedure in kernel mode to allocate the system region buffer and then access that buffer safely from user mode. In similar fashion, KER$ENTER_KERNEL_CONTEXT can call from a user-mode program a procedure that may have to call one or more additional procedures that do require kernel mode.

The KER$ENTER_KERNEL_CONTEXT procedure is dispatched through a typical kernel vector, one that uses the CHMK instruction to place its caller into kernel mode. This elevation of mode satisfies the access-mode requirement of the procedure to be called via KER$ENTER_KERNEL_CONTEXT. As shown in Figure 8–3, the first action the CHMK dispatcher takes is to elevate the caller's IPL to IPL$K_DISABLE_SWITCH to disable context switching within the current job. KER$ENTER_KERNEL_CONTEXT also allows its caller to execute the specified kernel-mode procedure at its original IPL by lowering the caller's IPL immediately upon entry. It then calls the specified procedure with a CALLG instruction, specifying the address of the argument list for the called procedure.

Figure 8–9 shows the code for KER$ENTER_KERNEL_CONTEXT, which executes as follows:

1. The caller's original IPL is extracted from the saved PSL that was pushed on the stack by the CHMK in the kernel vector and is written to R0.

2. This original IPL value is written to the PR$_IPL privileged register to return the caller's IPL to its value at the time of the call.
3. The specified procedure is called with a CALLG instruction. The CALLG operands are specified as deferred displacements from the pointer arguments for the argument list and the requested kernel-mode procedure.
4. When the called procedure completes execution, control returns to the REI instruction following the CALLG. When executed, the REI returns the caller to user mode and transfers control back to the kernel vector following the CHMK instruction.

Figure 8–9: KER$ENTER_KERNEL_CONTEXT Procedure

```
KER$ENTER_KERNEL_CONTEXT_S::
        EXTZV   #PSL$V_IPL,#PSL$S_IPL, -; fetch the caller's IPL
                4(SP),R0                ;  from the stack
        SETIPL  R0                      ; reset IPL for caller
                mtpr    R0,#PR$_IPL

        CALLG   @ARG_LIST(AP),@SUBROUTINE(AP)
                                        ; call the requested routine
        REI                             ; reenter caller's mode
```

Chapter 9

Memory Management and Dynamic Allocation

Support for VAXELN virtual memory is implemented partly in VAX hardware/microcode and partly in software. VAX microcode translates a virtual address to a physical address using the system and process page tables created and maintained by the VAXELN Kernel. (Virtual address translation is documented in the *VAX Architecture Reference Manual* and therefore is not described in this chapter.)

A VAXELN system image and its dynamic memory remain resident in the physical memory of the target computer throughout the life of the system — no paging to and from a disk is performed. This fact leads to a number of general characteristics of memory management under VAXELN:

- VAXELN systems can run in a diskless environment. Since a VAXELN system can boot and operate without a mass storage device, systems can function in environments that are too hostile for mass storage devices.
- Memory management is simplified. Compared to a paging executive such as that of VMS, memory management under VAXELN requires relatively few data structures and routines to support virtual memory.
- Memory management overhead is reduced. No effort is required of the kernel to keep track of valid (memory-resident) pages, nor does the real-time programmer have to ensure the validity of a page by locking it into physical memory. Under VAXELN, all mapped pages of virtual memory are valid.

- The amount of system virtual memory that a system can support is determined by the amount of physical memory available to the system.

The structure of VAXELN virtual memory and its mapping onto physical address space is managed by the VAXELN Kernel. The kernel maps portions of a VAXELN system into three virtual memory regions: the system region (S0 space), the program region (P0 space), and the control region (P1 space). S0 space is created during system initialization, described in Chapter 3. P0 space is created during job creation, and P1 space during process creation; both are described in Chapter 4.

This chapter describes the following aspects of VAXELN memory management and allocation that are supported by the VAXELN Kernel:

- The data structures that support memory management (Section 9.1)
- The allocation and deallocation of physical memory (Section 9.2)
- The allocation and deallocation of system and user virtual memory (Section 9.3)
- The allocation and deallocation of system pool blocks (Section 9.4)

9.1 Memory Management Data Structures

The VAXELN Kernel maintains a number of data structures to support mapping and allocating physical and virtual memory on the target system. Some of these structures, such as the system and process page tables, are defined by the VAX architecture and are required by the VAX microcode/hardware to perform address translation. Other structures are required by the kernel to support the memory requirements of jobs, processes, and hardware devices.

The following sections describe three classes of data structures that support VAXELN memory management:

- Allocation bitmaps and bitmap descriptors. The kernel uses bitmaps (a series of bits representing the elements of a resource) to monitor the allocation of physical and virtual pages of memory and the allocation page table. Within each bitmap, a 0 represents an allocated item and a 1 represents a free item. Each bitmap is paired with a bitmap descriptor that describes the location and size of the bitmap.

- Page tables and page table entries. The kernel creates and maintains page tables that map system, job, and process virtual memory to actual pages of physical memory. The mapping information for each page of virtual memory is stored within 32-bit page table entries (PTEs). One PTE exists for every page of virtual memory within the system.
- System, job, and process structures. Data that support system, job, and process virtual memory are stored in the kernel's global data block, the job control block (JCB), the process control block (PCB), and the process hardware context block (PTX). Data items that support system, job, and process memory appear in the global data block. Items associated with P0 memory are stored in the JCB, and items associated with P1 memory appear in the PCB and PTX.

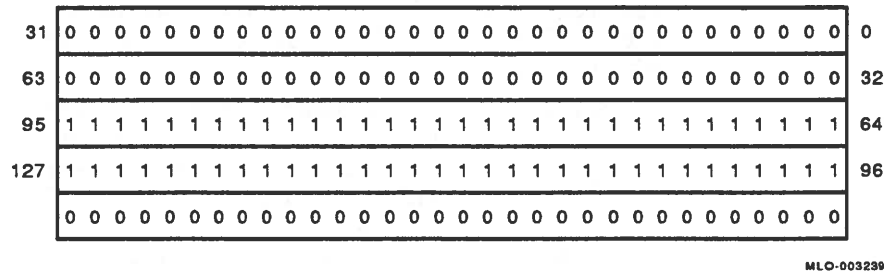
9.1.1 Allocation Bitmaps and Bitmap Descriptors

The kernel employs a simple form of data base to track the allocation of physical and virtual memory — the bitmap. An allocation bitmap is a series of bits in which each bit represents a page of memory or a process page table. A set (1) bit represents an available resource; a clear (0) bit represents an allocated resource. A cleared longword at the end of the bitmap represents its end.

The position of a bit within the bitmap (reading from low to high bits) represents the number of the allocated resource (the count begins at zero). For example, the tenth physical page of memory is represented by the tenth bit in the physical memory bitmap. That tenth bit would be reflected as page frame 9 (the actual bit number of the tenth bit) in the page table entry for a virtual page mapped to the tenth physical page on a target computer. For virtual pages, the bit number equates to the virtual page number (VPN) of the page within a region of virtual memory.

Figure 9–1 shows a bitmap that represents the allocation of 128 pages of memory. It consists of five longwords; the bits in the first two longwords are clear, showing that half the pages have been allocated. The bits in the next two longwords are set; the bits in the final longword are always clear to mark the end of the bitmap. The numbers on either side of the bitmap show the number of the page that the corresponding bit represents.

Figure 9–1: An Allocation Bitmap for 128 Pages of Memory



The kernel creates an allocation bitmap for each of the following resources:

- Physical memory (page frames). Each bit represents one page frame in the system's physical memory. This page frame bitmap, called the PFN bitmap, is created by the VMB bootstrap program (or the console program on the MicroVAX) when it sizes and tests the physical memory on the target computer. This bitmap is mapped into system space following the system control block.
- Communication region. Each bit represents one page of virtual memory within the system's communication region. The communication region bitmap is mapped into system address space adjacent to the communication region itself.
- P0 page table slots. Each bit represents one P0 page table slot (an area reserved for a P0 page table). The P0 page table bitmap is mapped into system space adjacent to the P0 page table slots themselves.
- P1 page tables slots. Each bit represents one P1 page table slot (an area reserved for a P1 page table). The P1 page table bitmap is mapped into system space adjacent to the P1 page table slots themselves.
- P0 virtual memory (one for each job created). Each bit represents one page of virtual memory in P0 address space; the bit number corresponds to the VPN of the page. A P0 bitmap is mapped into system space adjacent to its associated page table, within the page table slot.

- P1 virtual memory (one for each process created). Each bit represents one page of virtual memory in P1 address space; the bit number corresponds to the VPN of the page. A P1 bitmap is mapped into system space adjacent to its associated page table, within the page table slot.

Each bitmap in the system is associated with a descriptor to record its location and size. Figure 9–2 shows the structure of a bitmap descriptor (BMP), and Table 9–1 describes the uses of the BMP fields.

Figure 9–2: Structure of a Bitmap Descriptor

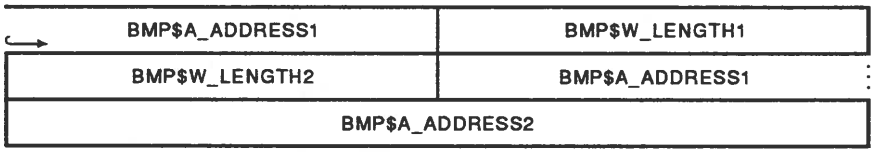


Table 9–1: Bitmap Descriptor Fields

Field	Meaning
BMP$W_ LENGTH1	The length in bytes of the bitmap from the first nonzero byte to the end of the bitmap (not including the end-of-bitmap longword)
BMP$A_ADDRESS1	The system virtual address of the first possible nonzero byte in the bitmap
BMP$W_ LENGTH2	The length in bytes of the entire bitmap (not including the end-of-bitmap longword)
BMP$A_ADDRESS2	The system virtual address of the entire bitmap

The length fields in the descriptor, which occupy one word each, are treated as unsigned values. This means that a bitmap can be up to 65,535 bytes in length and contain 524,280 (65535 * 8) bits — the

limit of the number of items the bitmap can be used to allocate. When applied to the page frame bitmap, for example, the length limit means that the kernel can address up to 524,280 pages of physical memory, or 256 megabytes.

The BMP$W_LENGTH1 and BMP$A_ADDRESS1 fields in the descriptor support the kernel's bitmap-manipulation subroutines by enabling it, under certain circumstances, to scan only the unused portion of the bitmap. Table 9–2 describes the local subroutines in module ALLOCATE that the kernel uses to manipulate the system's allocation bitmaps. These subroutines are called by memory and page table allocation routines.

Table 9–2: Bitmap Allocation Subroutines

Subroutine	Function
ALLOCATE_BITMAP	Allocates (that is, clears) a specified number of contiguous bits, updates the values of BMP$W_LENGTH1 and BMP$A_ADDRESS1, and returns the bit position of the first bit allocated.
EXAMINE_BITMAP	Attempts to allocate a specified number of contiguous bits beginning at a specified bit position. If all the specified bits are free (set), they are cleared and the starting bit position is returned; otherwise, the allocation fails.
FREE_BITMAP	Deallocates (that is, sets) a specified number of bits starting at a specified bit position and updates the values of BMP$W_LENGTH1 and BMP$A_ADDRESS1.

The sample allocation bitmap in Figure 9–1 would have the following values in its bitmap descriptor:

Field	Value	Meaning
BMP$W_LENGTH2	16	The entire bitmap is 16 bytes long.
BMP$W_LENGTH1	8	The length of the bitmap from the first nonzero byte to the end is 8 bytes.
BMP$A_ADDRESS2	*n*	Assume that the bitmap begins at address *n*.
BMP$A_ADDRESS1	*n* + 8	The first nonzero byte in the bitmap is 8 bytes from the base address of the bitmap (*n*).

9.1.2 Page Tables and Page Table Entries

A VAX page table is a contiguous array of longwords — page table entries (PTEs) — that records the characteristics of pages of virtual memory. The page table contains one PTE for each associated virtual page; each PTE specifies the physical page of memory (a page frame) to which the virtual page corresponds. The entries in the page table therefore map virtual memory to physical memory. The following sections describe page tables and PTEs under VAXELN.

9.1.2.1 VAXELN Page Tables

A page table is defined by its base address and its length in PTEs. The base and length of a page table are used during address translation to locate the page table in physical or virtual memory and validate the virtual address being translated.

The VAX architecture defines three types of page tables, S0, P0, and P1, to map the system, program, and control regions, respectively. The following sections describe these page tables and how the kernel creates them.

9.1.2.1.1 S0 Page Table

One system page table (SPT) maps the entire VAXELN system into system address space (see Figure 3–2). The SPT is defined by the SBR and SLR base and length internal processor registers. SBR contains the base physical address of the system page table, which must reside in contiguous physical memory. SLR records the size, in PTEs, of the system page table. Each system PTE (SPTE) maps one page of virtual memory to physical memory.

The size of the SPT — and therefore the size of S0 address space — is determined during system initialization when the kernel calculates the number of page frames the system's components require. Those components, including the SPT itself, are then mapped into system virtual memory, filling the SPT with PTEs.

The S0 page table enables jobs to share physical memory, such as areas and shareable image sections, within a system. Memory can be made shareable through a process called double mapping. To double map memory, the kernel copies the SPTEs that map the physical memory to be shared into the P0 page table of the job or jobs that will share the memory. The new P0 page table entries allow the job to use its P0 address space to refer to the same physical pages as the original SPTEs.

Once it has been initialized, the S0 page table cannot be expanded because it must reside in contiguous physical memory. Dynamic allocation of S0 memory comes not from expanding the S0 page table but from mapping physical memory with unused SPTEs. These unused entries exist in sections of the S0 page table reserved by the kernel for mapping dynamically allocated P0 and P1 page tables and the communication region.

The communication region comprises the only truly dynamic virtual memory within the S0 address space. After system initialization, most virtual pages in this range of virtual addresses remain unmapped. They are allocated dynamically on request from system and user code using the procedures described in Section 9.3.1. The number of pages that can be allocated from the communication region is limited by the value of KER$GW_IO_SIZE.

System pool blocks represent another method of obtaining S0 memory, this time in the form of fixed-length blocks that were mapped into system space during system initialization. The allocation of blocks from the system pool is described in Section 9.4.

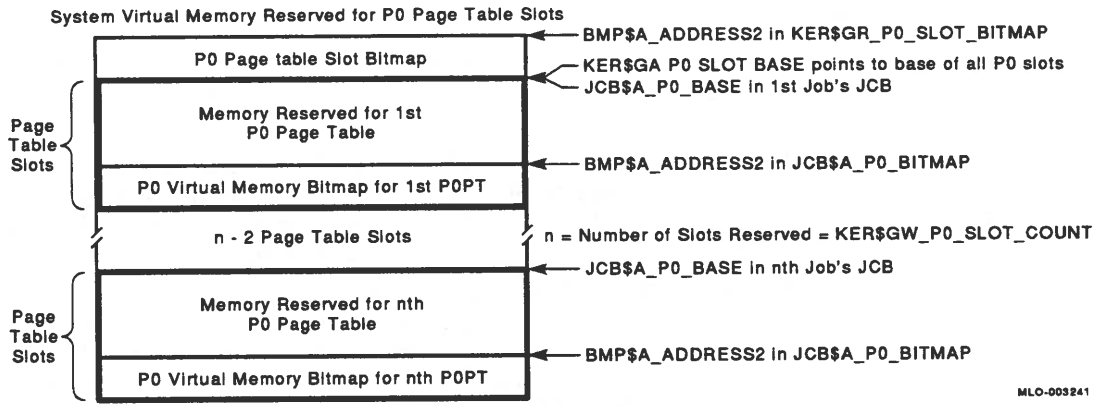
9.1.2.1.2 P0 Page Tables

The virtual memory for each job in the system is mapped to physical memory in a P0 page table (P0PT), which is shared by all processes in a job (see Figure 4–5). The P0 page table is defined by the P0BR and P0LR base and length processor registers and resides in contiguous system virtual memory. The first PTE in the P0 page table maps the first page of P0 virtual memory. P0BR contains the base system virtual address of the page table. P0LR records the size, in PTEs, of the P0 page table. This page table can expand dynamically at run time up

to the limit set by the **P0 virtual size** value specified on the System Characteristics Menu. (On the KA620 processor, which uses the rtVAX subset of the VAX architecture, the P0 page table resides in contiguous physical memory, and P0BR contains the physical address of the page table.)

During system initialization, the kernel reserves a block of system virtual memory large enough to contain the maximum number of P0 page tables. The reserved area is logically divided into segments called page table slots. Each slot is large enough to hold a single page table and its associated bitmap. This number of reserved slots is determined by the value of the **Number of jobs** entry on the System Characteristics menu. Figure 9–3 shows how the reserved area is divided into P0 page table slots. (The same arrangement holds for P1 page table slots, which follow the P0 slots in system virtual memory.)

Figure 9–3: Layout of P0 Page Table Slots



When a job is created, a P0 page table slot is allocated from the reserved area and initialized to contain no PTEs. When the kernel maps the job's program image into P0 space, virtual memory is allocated to contain the program, and a P0 page table entry (P0PTE) is created for each page of virtual memory.

When all the PTEs within the page table are allocated, the table is expanded by 128 more entries (that is, one page frame is allocated to hold the new PTEs), and the P0LR for each process in the job is updated to reflect the expansion of the table. This dynamic expansion of the P0 page table can continue until the table reaches its size limit, at which point the job's virtual memory is exhausted. (On the KA620 processor, where page tables must be in contiguous physical memory, the P0 page table is expanded to its maximum size at its creation because the page frames subsequently allocated for the table might not be contiguous.)

During address translation, the virtual page number (VPN) derived from bits <29:9> of the P0 virtual address is used as an index into the P0 page table. The VPN is multiplied by 4 and added to the value of P0BR to yield the system virtual address of the PTE for the virtual page being referenced. The information in the PTE is then used to locate the page frame mapped by the virtual page.

A P0 page table is created by the kernel during job creation to map the job's P0 virtual memory. In the KER$CREATE_JOB procedure, a P0 page table is created as follows:

1. The kernel calls the internal subroutine KER$ALLOCATE_P0_SLOT (in module ALLOCATE). This routine performs the following operations:
   a. The local subroutine ALLOCATE_BITMAP is called to clear a bit in the P0 page table bitmap, located at KER$GR_P0_SLOT_BITMAP. If no set bits remain in the bitmap, no more jobs can be created, and KER$CREATE_JOB will fail.
   b. The base system virtual address of the P0 page table is determined by multiplying the number of the bit cleared in the P0 slot bitmap by the size in pages of a P0 page table slot (KER$GW_P0_SLOT_LENGTH) and adding the result to the value of KER$GA_P0_SLOT_BASE.
   c. The internal subroutine KER$ALLOCATE_FRAME is called to allocate physical memory for the P0 memory allocation bitmap. The subroutine clears enough bits in the PFN bitmap to hold the bitmap. Those bit numbers are used to map the bitmap in the S0 page table.
   d. The entire P0 bitmap is cleared. Clearing the bitmap here allows P0 memory to expand in increments of 128 pages at a time. (On the KA620 processor, all page frames for the page table are allocated at once to ensure their physical contiguity.)

e. The subroutine returns the virtual address of the P0 page table and the size and address of the P0 bitmap.

2. The base address of the P0 page table is inserted into the PTX$A_P0BR field of the master process's hardware process context block and the JCB$A_P0_BASE field of the job control block.
3. The BMP$A_ADDRESS1 and BMP$A_ADDRESS2 fields in the job's P0 bitmap descriptor located at JCB$A_P0_BITMAP are initialized. The length fields in the descriptor are not set — their values remain 0. This zero-length bitmap will cause the allocation of the first 128 pages of virtual memory for the job when the kernel first attempts to allocate P0 memory to map the job's image sections. See Section 9.3.2.1.

Job creation is discussed in detail in Chapter 4. As a job is deleted, its P0 page table is returned to the pool of available page table slots when the kernel, in KER$DELETE, calls KER$FREE_P0_SLOT (in module ALLOCATE). This internal subroutine operates as follows:

1. Using the system virtual address of the P0 page table in JCB$A_P0_BASE, the following values are determined:
   - The SPTE in the system page table that maps the base of the P0 page table
   - The position of the P0 page table in the array of P0 page table slots beginning at KER$GA_P0_SLOT_BASE
2. The PFN of the first physical page of the page table is extracted from the SPTE.
3. The SPTE is cleared to unmap the page table page.
4. Using the PFN of the page, the page frame is returned to the system.
5. The previous three steps are repeated until all the SPTEs devoted to the page table are cleared.
6. The address translation buffers on all active processors are invalidated.
7. The position of the page table in the array of page table slots is passed to the local subroutine FREE_BITMAP to reset the bit allocated for the page table.

9.1.2.1.3 P1 Page Tables

The virtual memory for each process in the system is mapped to physical memory in a P1 page table (P1PT). Each process has its own P1 page table to map its user and kernel stacks (see Figure 4–6). The P1 page table is defined by the P1BR and P1LR base and length processor registers and resides in contiguous system virtual memory. The first PTE in the P1 page table maps the first page of P1 virtual memory, which grows from high to low addresses.

Like P1 space itself, the P1 page table grows toward smaller addresses. To simplify address translation, the base address of the page table, stored in P1BR, is the virtual address of the P1 page table entry (P1PTE) that would map virtual address 40000000_{16}, the base of P1 space. This allows a P1 virtual page number to be used as an index into the P1 page table. (In other words, the first P1 virtual page allocated is the last page at the high end of P1 address space, resulting in the maximum index value into the P1 page table. This index value will correspond to the "first" P1PTE in the page table, that is, the PTE at the high end of the page table.) Accordingly, P1LR contains the number of P1PTEs that do not exist. (The value of P1LR is calculated by subtracting the number of existing P1 pages from the maximum number of pages that can be mapped into P1 space, 2^{21}).

The P1 page table can expand dynamically at run time up to the limit set by the **P1 virtual size** value specified on the System Characteristics Menu. (On the KA620 processor, the P1 page table resides in contiguous physical memory, and P1BR contains the physical address of the PTE that would map virtual address 40000000_{16}.)

During system initialization, the kernel reserves a block of system virtual memory large enough to contain the maximum number of P1 page table slots. This number is determined by adding the value of the **Number of jobs** entry (that is, the number of master processes) on the System Characteristics menu to the value of the **Number of subprocesses** entry. When a process is created by KER$CREATE_PROCESS, a P1 page table slot is allocated from the reserved area and initialized to contain no PTEs. When the kernel then maps the process's local memory into P1 space, virtual memory is allocated, and a P1PTE is created to map each page.

When all the PTEs within the page table are allocated, the table is expanded by 128 more entries (that is, one page frame is allocated to hold the new PTEs). P1LR for the process is updated to reflect each expansion of the table. This dynamic expansion of the P1 page table can continue until the table reaches its size limit. (On the KA620

processor, the P1 page table is expanded to its maximum size at its creation because the allocation of subsequent page frames for the table might not be contiguous.)

During address translation, the virtual page number derived from bits <29:9> of the P1 virtual address is used as an index into the P1 page table. The VPN is multiplied by 4 and added to the value of P1BR to yield the address of the PTE for the virtual page being referenced. The information in the PTE is then used to locate the page frame mapped by the virtual page.

A P1 page table is created by the kernel during process creation to map the process's P1 virtual memory. In the KER$CREATE_JOB (for the job's master process) and KER$CREATE_PROCESS (for subprocesses) procedures, a P1 page table is created as follows:

1. The kernel calls the internal subroutine KER$ALLOCATE_P1_SLOT (in module ALLOCATE). This routine performs the following operations to create a P1 page table for the process:
   a. The local subroutine ALLOCATE_BITMAP is called to clear a bit in the P1 page table slot bitmap, located at KER$GR_P1_SLOT_BITMAP. If no set bits remain in the bitmap, no more processes can be created.
   b. The base system virtual address of the P1 page table is determined by multiplying the number of the bit cleared in the P1 slot bitmap by the size in pages of a P1 page table slot (KER$GW_P1_SLOT_LENGTH) and adding the result to the value of KER$GA_P1_SLOT_BASE.
   c. The internal subroutine KER$ALLOCATE_FRAME is called to allocate physical memory for the P1 memory allocation bitmap. The subroutine clears enough bits in the PFN bitmap to hold the bitmap. Those bit numbers are used to map the bitmap in the S0 page table.
   d. The entire P1 bitmap is cleared. Clearing the bitmap here allows P1 memory to expand in increments of 128 pages at a time. (On the KA620 processor, all page frames for the page table are allocated at once to guarantee that they be physically contiguous.)
   e. The subroutine returns the virtual address of the P1 page table and the size and address of the P1 bitmap.

2. The system virtual address of the P1 page table is inserted into PCB$A_P1_BASE field of the process control block. That address is then converted to reflect the base address of the P1PTE that would map virtual address 40000000_{16} and is inserted into the PTX$A_P1BR field of the process's hardware context block. This is the base address of the page table used in address translation.
3. The BMP$A_ADDRESS1 and BMP$A_ADDRESS2 fields in the process's P1 bitmap descriptor located at PCB$A_P1_BITMAP are initialized. The length fields in the descriptor are not set — their values remain 0. This zero-length bitmap will cause the allocation of the first 128 pages of virtual memory for the process when the kernel first attempts to allocate P1 for the process stacks. See Section 9.3.2.1.

Process creation is discussed in detail in Chapter 4, Job and Process Creation and Deletion. As a process is deleted, its P1 page table is returned to the pool of available page table slots when the kernel, in KER$DELETE, calls KER$FREE_P1_SLOT (in module ALLOCATE). This internal subroutine operates as follows:

1. Using the system virtual address of the P1 page table in PCB$A_P1_BASE, the following values are determined:
   - The SPTE in the system page table that maps the base of the P1 page table
   - The position of the of P1 page table in the array of P1 page table slots beginning at KER$GA_P1_SLOT_BASE
2. The PFN of the first physical page of the page table is extracted from the SPTE.
3. The SPTE is cleared to unmap the page table page.
4. Using the PFN of the page, the page frame is returned to the system.
5. The previous three steps are repeated until all the SPTEs devoted to the page table are cleared.
6. The address translation buffers on all active processors are invalidated.
7. The position of the page table in the array of page table slots is passed to the local subroutine FREE_BITMAP to reset the bit allocated for the page table.

9.1.2.2 VAXELN Page Table Entries

The page table entries in the VAXELN page tables are used by the VAX memory management subsystem to translate a virtual address to its physical counterpart. Allocating virtual memory is largely a matter of allocating page table entries. Those PTEs can then be used to map physical memory into system, job, or process address space. PTE allocation is described in Section 9.3.

Figure 9–4 shows the bit fields defined within the PTE longword. These fields describe the protection, ownership, type, and corresponding page frame of a page of virtual memory. Table 9–3 describes the meanings and uses of the PTE fields.

Figure 9–4: Structure of a VAXELN Page Table Entry

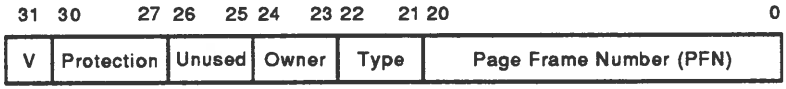


Table 9–3: VAXELN PTE Fields

Name	Extent	Meaning
Valid (V)	<31>	Under the VAX architecture, indicates whether the referenced virtual page is resident in physical memory. Because VAXELN does not page memory to disk, this bit is always set to ensure that the VAX microcode does not raise a translation-not-valid fault during address translation.
Protection	<30:27>	Indicates the access modes at which a process can reference the virtual page. During address translation, this field is evaluated and compared to the access mode of the referencer; if that process's access mode is insufficient for the attempted reference, an access-control violation fault occurs. Table 9–4 shows the VAXELN protection codes that can appear in this field.

Table 9–3 (Cont.): VAXELN PTE Fields

Name	Extent	Meaning
Owner	<24:23>	Indicates the owner of the virtual page, either the kernel (PTE$C_KOWN) or a user program (PTE$C_UOWN). P0 or P1 virtual pages allocated by the kernel or by kernel-mode jobs are marked with kernel ownership. User-mode jobs can only delete pages marked with PTE$C_UOWN ownership. Pages allocated in the communication region are always marked PTE$C_KOWN, even if they are allocated by a user-mode job calling KER$ALLOCATE_SYSTEM_REGION through the KER$ENTER_KERNEL_CONTEXT procedure.
Type	<22:21>	Indicates the virtual page's type. Table 9–5 shows the VAXELN type codes.
PFN	<20:0>	The upper 21 bits of the physical address of the base of the mapped page frame.

Table 9–4 shows the protection codes that can appear in VAXELN PTEs. Pages allocated in the context of a user-mode process are normally marked with PTE$C_UW protection. Pages allocated by the kernel or by a kernel-mode process are marked with PTE$C_URKW protection. The kernel or a kernel-mode process can read or write any page except one that is marked PTE$C_KR (kernel read-only) or PTE$C_NA (no access). Protections other than PTE$C_UW and PTE$C_URKW are usually set only by the internal kernel procedure KER$SET_PROTECTION.

Table 9–4: PTE Memory-Access Protection Codes

Protection	Symbol	Binary Value
No access allowed	PTE$C_NA	0000
Kernel write (and read)	PTE$C_KW	0010
Kernel read (no write)	PTE$C_KR	0011
User write (and read)	PTE$C_UW	0100
User read, kernel write	PTE$C_URKW	1110
User read (no write)	PTE$C_UR	1111

Table 9–5 shows the PTE type codes and their uses. These codes identify the nature of the mapped page. The type field is set when memory is allocated and is tested only when memory is being freed,

either by calls to KER$FREE_MEMORY and KER$FREE_SYSTEM_REGION or during job and object deletion.

Table 9–5: PTE Type Codes

Type	Symbol	Binary Value	Meaning
System	PTE$K_SYSTEM	00	The virtual page is mapped in the SPT, or the page is mapped to a specific physical address by a kernel-mode job (usually a device driver mapping I/O space). All virtual pages mapped in the S0 page table are of type PTE$K_SYSTEM.
Code	PTE$K_USER_CODE	01	The virtual page is user code and is double mapped to a page frame that can be shared by multiple jobs.
Message	PTE$K_MESSAGE	10	The virtual page is a user page mapped to a page frame containing a message or area data buffer; or the page is a user page mapped to a specific physical address in the communication region.
Data	PTE$K_RW_DATA	11	The virtual page is a user page mapped to a page frame containing a job's private read/write data. PTEs of this type can appear in both the P0 and P1 page tables. P1 pages, which do not map any portion of the system image, can contain only read/write pages for user-mode jobs.

When the virtual page is being deleted, the kernel checks the type field in the PTE. The action taken to delete the page depends on its type:

- Virtual pages of type PTE$K_USER_CODE are double mapped (that is, mapped in both the SPT and the P0PT) from physical pages containing program code that can be executed by other jobs in the system. When a page of this type is deleted, only the P0PTE itself is freed — the page frame containing the code cannot be freed because it may be mapped into P0 memory of other jobs.
- Virtual pages of type PTE$K_RW_DATA are mapped to page frames for the exclusive use of a job. When such a page is deleted, both the P0PTE and the page frame can be freed, because the page frame contains a private copy of the job's read/write data.
- Virtual pages of type PTE$K_SYSTEM are treated in the same way as user code pages; that is, only the PTE associated with the page is freed — the associated page frame is not freed for reuse.

- Virtual pages of type PTE$K_MESSAGE can be mapped into multiple jobs (first in a sending job, then in a receiving job). Therefore, they can be freed only by the KER$DELETE procedure when their associated message or area is deleted (which can occur during job deletion).

9.1.3 System, Job, and Process Structures

The kernel stores data required for memory management within the system data structures most closely associated with the region of memory supported by the data. For example, since P0 virtual memory is managed at the job level, the kernel stores data associated with a job's P0 memory in the job control block (JCB). Likewise, data associated with P1 memory is stored in the process control block (PCB) and process hardware context block (PTX).

The following sections describe locations and uses of the data structures the kernel maintains to support memory management.

9.1.3.1 System Memory Management Structures

The kernel stores data that is applicable across an entire VAXELN system within the global data block, defined in module SYSTEMDAT and described in Section 2.3.2.

Table 9–6 shows the data items in the kernel data block that support memory management at the system level.

Table 9–6: Memory Management Data Stored in the Kernel Data Block

Data Item	Meaning
KER$GA_SPT_PHYSICAL	The physical address of the system page table. This value is used to locate the SPT early in system initialization, before memory management is enabled. The value corresponds to the value of SBR.
KER$GA_SPT_BASE	The system virtual address of the system page table. This value is used to locate the SPT at run time.

Table 9–6 (Cont.): Memory Management Data Stored in the Kernel Data Block

Data Item	Meaning
KER$GL_SPT_LENGTH	The length in SPTEs of the system page table. This value corresponds to the number of physical pages in the VAXELN system and is used to set the value of SLR.
KER$GR_PAGE_BITMAP	The descriptor for the page frame allocation bitmap. This descriptor is used to locate the page frame bitmap during the allocation of physical memory.
KER$GR_REGION_BITMAP	The descriptor for the communication region allocation bitmap. This descriptor is used to locate the region bitmap during the allocation of memory from the communication region.
KER$GA_REGION_BASE	The system virtual address of the base of the communication region. This value is used during memory allocation to locate the SPTE that maps the first page of the communication region.
KER$GR_P0_SLOT_BITMAP	The descriptor for the P0 page table slot allocation bitmap. This descriptor is used to locate the P0 page table bitmap during the creation of a P0 page table.
KER$GW_P0_SLOT_LENGTH	The number of pages reserved for a P0 page table slot. This value includes the size of the P0 page table itself and the size of the P0 allocation bitmap that also resides in the slot.
KER$GW_P0_SLOT_SIZE	The size in pages of a P0 page table. This value is used to allocate page frames to hold a P0 page table.
KER$GA_P0_SLOT_BASE	The system virtual address of the base of the P0 page table slots. This value corresponds to the number of virtual pages a job can create.

Table 9–6 (Cont.): Memory Management Data Stored in the Kernel Data Block

Data Item	Meaning
KER$GR_P1_SLOT_BITMAP	The descriptor for the P1 page table slot allocation bitmap. This descriptor is used to locate the P1 page table bitmap during the creation of a P1 page table.
KER$GW_P1_SLOT_LENGTH	The number of pages reserved for a P1 page table slot. This value includes the size of the P1 page table itself and the size of the P1 allocation bitmap that also resides in the slot.
KER$GW_P1_SLOT_SIZE	The size in pages of a P1 page table. This value is used to allocate page frames to hold a P1 page table.
KER$GA_P1_SLOT_BASE	The system virtual address of the base of the P1 page table slots. This value corresponds to the number of virtual pages a job can create.

9.1.3.2 Job Memory Management Structures

The kernel stores data that is applicable to jobwide memory management in the JCB. The kernel uses this data when memory is allocated or freed for the job. Table 9–7 shows the data items in the JCB that support memory management at the job level.

Table 9–7: Job Memory Management Data Stored in the JCB

Data Item	Meaning
JCB$A_P0_BASE	The system virtual address of the P0 page table. This value is used to locate the P0 page table during page table expansion. The value corresponds to the value of P0BR.
JCB$L_P0_LIMIT	The number of PTEs in the P0 page table. This value is updated during the expansion of the P0 page table and corresponds to the value of the P0LR.

Table 9–7 (Cont.): Job Memory Management Data Stored In the JCB

Data Item	Meaning
JCB$A_P0_BITMAP	The descriptor for the P0 virtual page allocation bitmap. This descriptor is used to locate and update the P0 bitmap during the allocation of P0 virtual memory.
JCB$L_RW_DATA_PTE	A prototype page table entry for the creation of PTEs for the job's read/write data. The valid, protection (PTE$C_UW), owner (the program mode), and type (PTE$K_RW_DATA) fields are present in the prototype PTE. During P0 memory allocation, the allocated page frame number is inserted into the PFN field to create the actual PTE for the page.
JCB$L_MESSAGE_PTE	A prototype page table entry for the creation of PTEs for the job's message and area data buffers. The valid, protection (PTE$C_UW), owner (the program mode), and type (PTE$K_MESSAGE) fields are present in the prototype PTE. During the allocation of message and area buffers, the allocated page frame number is inserted into the PFN field to create the actual PTE for the page.
JCB$B_MODE	The base access mode of the program as set in the job's program descriptor. This value becomes the value of the ownership field in PTEs allocated for the job.
JCB$A_PROCESS_FLINK	The address of the first PCB in the list of the job's processes. During the expansion of the P0 page table, this address is used to locate the process hardware context block (PTX) of each process in the job and update the value of its P0 length register.

9.1.3.3 Process Memory Management Structures

The kernel stores data that is applicable to process memory management in a process's control block (PCB) and its hardware context block (PTX). The PCB records the software context of a process; the PTX stores the hardware context of a process. The kernel uses the information in the PCB when stack memory is allocated or freed for the

process. Whenever P0 or P1 space is expanded for a process, the kernel updates the appropriate values in both the PCB and PTX. When the process is scheduled to run, the information in its PTX structure is loaded into the processor by the LDPCTX instruction. The memory management data in the PTX then defines the virtual memory for the process.

Tables 9–8 and 9–9 show the data items in the PCB and PTX, respectively, that support memory management at the process level.

Table 9–8: Process Memory Management Data Stored in the PCB

Data Item	Meaning
PCB$A_P1_BASE	The system virtual address of the P1 page table. This is the address of the page table slot allocated during process creation. The page table grows toward this address from the high end. This value is not the same as PTX$A_P1BR, which reflects the base address of the nonexistent portion of the P1 page table. See Section 9.1.2.1.3.
PCB$A_P1_BITMAP	The descriptor for the P1 virtual page allocation bitmap. This descriptor is used to locate and update the P1 bitmap during the allocation of P1 virtual memory.
PCB$L_P1_LIMIT	The number of nonexistent PTEs in the P1 page table. This value is updated during the expansion of the P1 page table and corresponds to the value of the P1LR.

Table 9–9: Process Memory Management Data Stored in the PTX

Data Item	Meaning
PTX$A_P0BR	The contents of the P0 base register for this process. This value is identical for all processes in a job and is loaded into the P0BR when the process runs.
PTX$L_P0_LIMIT	The contents of the P0 length register for this process. This value is identical for all processes in a job and is loaded into the P0LR when the process runs.

Table 9–9 (Cont.): Process Memory Management Data Stored in the PTX

Data Item	Meaning
PTX$A_P1BR	The contents of the P1 base register for this process. This value is loaded into the P1BR when the process runs.
PTX$L_P1_LIMIT	The contents of the P1 length register for this process. This value is loaded into the P1LR when the process runs.

9.2 Allocating Physical Memory

The VAXELN Kernel supports up to 256 megabytes (524,280 pages) of physical memory. Each page frame of physical memory is represented by one of the bits in the PFN bitmap. Therefore, this 256-megabyte limit is set by the 16-bit size of the length field in the PFN bitmap descriptor. If the **Memory limit** system characteristic has been set to a value less than the number of bits in the PFN bitmap, the kernel, during system initialization, truncates the bitmap at the specified limit, making page frames beyond the limit unknown to the kernel.

Under VAXELN, the allocation of physical memory is a straightforward process of bitmap manipulation performed by the KER$ALLOCATE_FRAME subroutine (in module ALLOCATE). Physical memory is allocated by a number of kernel procedures and routines, such as KER$CREATE_JOB (to map read/write image sections) and KER$ALLOCATE_MEMORY (to hold dynamically allocated data), and never directly by the user. Before the kernel routine can allocate physical memory, it must first obtain a page table entry to receive the PFN of the physical page allocated. For example, to allocate P0 memory, the KER$ALLOCATE_MEMORY procedure reserves a range of virtual memory by allocating PTEs with a call to KER$ALLOCATE_P0_PTE (described in Section 9.3). These PTEs will hold the PFNs of the page frames allocated from physical memory.

Once PTEs are allocated, KER$ALLOCATE_FRAME is called to obtain physical memory. The number of page frames required is passed to the routine. The routine executes as follows:

1. The address of the PFN bitmap descriptor, located at KER$GR_PAGE_BITMAP, and the number of page frames to be allocated are passed to the local subroutine ALLOCATE_BITMAP. That routine

locates the required number of contiguous available bits in the PFN bitmap, clears them, updates the bitmap descriptor, and returns the starting bit number (PFN).

2. The starting bit number from ALLOCATE_BITMAP is returned to the calling routine.

Given the starting bit number — which equates to the page frame number — the routine allocating the physical memory inserts the allocated PFNs into the reserved PTEs.

Freeing physical memory simply reverses the allocation process. The routine freeing the physical memory locates the PTE that contains the PFN of the page frame — or the first in a series of page frames — to be freed. This PFN and the number of PFNs are then passed as arguments to the internal subroutine KER$FREE_FRAME, which executes as follows:

1. The address of the PFN bitmap descriptor, the starting PFN, and the number of PFNs to be freed are passed to the local subroutine FREE_BITMAP. That routine locates the specified bit or bits in the PFN bitmap, resets them, updates the bitmap descriptor, and returns.
2. The routine returns to the caller.

The calling routine then clears the associated PTEs, and the physical memory has been freed.

9.3 Allocating Virtual Memory

The allocation of system (S0) and user (P0 and P1) memory is supported by several low-level kernel routines, which in turn are called by several high-level kernel procedures to provide memory allocation services for user programs. The following sections describe those kernel routines — KER$ALLOCATE_REGION, KER$ALLOCATE_P0_PTE, and KER$ALLOCATE_P1_PTE — and their high-level interfaces — KER$ALLOCATE_SYSTEM_REGION and KER$ALLOCATE_MEMORY.

9.3.1 Allocating System Virtual Memory

When the kernel creates the system page table, it reserves the number of page frames necessary to hold the SPTEs to map all the memory required by the VAXELN system. A range of these SPTEs, at the end of the page table, is reserved to map a multipurpose range of virtual address space called the communication region (see Figure 3–2). (This range of addresses is referred to in the VAXELN user documentation as the "system region," a name not used here because it conflicts with the architectural name of S0 address space.) Because the S0 page table cannot be dynamically expanded, the communication region provides the only truly dynamic virtual memory within the S0 address space.

The communication region is used by a number of VAXELN facilities to provide dynamic memory within system address space. Among the uses of the communication region are the following:

- To map communication buffers for transmitting data between an interrupt service routine (ISR) and a device driver job. System memory is used for this purpose because an ISR executes in system context on the interrupt stack and cannot access memory within the context of the driver job. This memory is allocated by the KER$CREATE_DEVICE procedure, which also allocates page-long interrupt dispatch blocks from this region.
- To map job and process control blocks. System memory is used for these structure because their sizes exceed the 128-byte size of a system pool block. The page allocated to map a process control block alos contains the process's hardware context block. A job's object pointer table is also allocated from this region.
- To map dynamically loaded programs. System memory is used for this purpose because the image sections of the loaded program must be mapped from system addresses by KER$CREATE_JOB. This memory is allocated by the ELN$LOAD_PROGRAM utility with a call to KER$ALLOCATE_SYSTEM_REGION.
- To map the physical addresses of I/O adapter and device control/status registers (CSRs). System memory is used for this purpose because this address space must be accessible outside job context.
- To map the memory allocated for a virtual disk drive by the VMDRIVER. System memory is used for this purpose so that the virtual disk is not confined to P0 memory of the VMDRIVER, which limits the size of the disk to fewer than 16 megabytes.

- To map memory allocated by user programs. System memory is used for this purpose so that the memory allocated does not count against the virtual memory limit of the job or to make the memory potentially accessible outside job context.

The number of SPTEs reserved for the communication region is determined by the global parameter value KER$GW_IO_SIZE. The original value of this parameter, as transmitted by the System Builder to the kernel, is determined by adding the values of the **System region size** and **Dynamic program space** entries on the System Characteristics Menu. This value is then increased to accommodate the maximum number of jobs and processes the system can contain. During initialization, the kernel further augments this value by the number of pages required to map the system's bus space and device CSRs. This final value determines the number of SPTEs and the number of bits in the allocation bitmap devoted to the communication region.

Virtual memory from the communication region is allocated by the internal subroutine KER$ALLOCATE_REGION. The kernel calls this routine directly (for example, in KER$CREATE_DEVICE) and from the KER$ALLOCATE_SYSTEM_REGION kernel procedure, which provides the public interface for the allocation of system memory. The following sections describe these procedures and their complementary routines, KER$FREE_REGION and KER$FREE_SYSTEM_REGION.

9.3.1.1 KER$ALLOCATE_REGION and KER$FREE_REGION Subroutines

The subroutine KER$ALLOCATE_REGION (in module ALLOCATE) provides the kernel's internal method for allocating virtual memory from the communication region. The S0 memory allocated by this routine is both virtually and physically contiguous. Two internal entry points for the routine exist:

- KER$ALLOCATE_REGION sets memory protection to PTE$C_URKW. This protection prevents user-mode jobs from modifying the memory.
- KER$ALLOCATE_REGION_USER sets memory protection to PTE$C_UW. This protection allows user-mode jobs to modify the memory.

Either entry point expects two input values: the number of pages to be allocated and the physical address to which the allocated S0 addresses will be mapped. If 0 is supplied as the physical address, the first available contiguous set of page frames i sallocated (marked as used in the PFN bitmap). If a physical address is supplied, no page frames are allocated. The routine assumes that the caller has control over the use of those page frame and inserts the requested page frame numbers into the system page table. Both routines return a success/failure status value and the system virtual address of the memory allocated from the communication region.

KER$ALLOCATE_REGION executes as follows:

1. A prototype SPTE is constructed. If the entry point to the routine is through KER$ALLOCATE_REGION, the protection field in the PTE is set to user-read/kernel-write (PTE$C_URKW); otherwise, the protection is set to PTE$C_UW (user-write). In both cases, the owner field is set to kernel ownership (PTE$C_KOWN). After the creation of the prototype SPTE, execution continues in common code.
2. The address of the communication region bitmap descriptor, located at KER$GR_REGION_BITMAP, and the number of virtual pages to be allocated are passed to the local subroutine ALLOCATE_ BITMAP. That routine locates the required number of contiguous set bits in the region bitmap, clears them, updates the bitmap descriptor, and returns the starting bit number. This bit number represents the virtual page *within* the communication region of the base of the memory allocated. It determines which SPTE will map the allocated memory.
3. If no physical address was specified, the number of pages required is passed to the internal subroutine KER$ALLOCATE_FRAME to allocate that number of contiguous page frames. The subroutine returns the starting bit number (that is, the PFN) of the page frames allocated. This bit number is used to set the PFNs in the SPTEs that map the page frames.

   If a physical address was specified, no page frames are allocated. Instead, the starting PFN for the SPTEs is extracted from the specified physical address.
4. The VPN for the communication region is extracted from the value of KER$GA_REGION_BASE.
5. That VPN is added to the bit number set in the region bitmap by ALLOCATE_BITMAP.

6. This new value — the VPN of the allocated memory — is used as an index into KER$GA_SPT_BASE to yield the address of the SPTE that maps the base of the allocated memory.
7. The type field in the prototype SPTE is set. If the allocation is to a specified physical address, the type is set to PTE$K_MESSAGE. When this virtual page is freed, this type tells the kernel not to call KER$FREE_FRAME, because no bit was actually cleared in the PFN bitmap.

   If the allocation was not to a physical address, the type field in the prototype is set to PTE$K_SYSTEM, signifying that the page is mapped in the system page table.
8. Using the address of the first SPTE, the prototype SPTE and the PFN of the virtual page are inserted into the SPTE to map the virtual page to the page frame. If multiple pages were allocated, the PFN is increased by one, and the process is repeated until all the necessary SPTEs are filled.
9. The base system virtual address of the allocated memory is returned to the caller. This value is determined by adding the bit number returned from the region bitmap to the VPN in KER$GA_REGION_BASE.

Memory allocated by the KER$ALLOCATE_REGION subroutine can be returned to the system with the internal subroutine KER$FREE_REGION. This routine expects two input values: the starting virtual address of the region to be freed and the number of pages to be freed. KER$FREE_REGION executes as follows:

1. The base address of the communication region — KER$GA_REGION_BASE — is subtracted from the starting virtual address of the memory to be freed.
2. The VPN is extracted from the value just obtained. This represents the bit position of the page *within* the region bitmap.
3. The address of the region bitmap descriptor, the starting bit number, and the number of pages to be freed are passed to the local subroutine FREE_BITMAP. That routine locates the specified bit or bits in the region bitmap, resets them, updates the bitmap descriptor, and returns.
4. The VPN for the communication region is extracted from the value of KER$GA_REGION_BASE.
5. The base VPN of the communication region is added to the memory's bit number in the region bitmap.

6. This new value — the VPN of the allocated memory — is used as an index to KER$GA_SPT_BASE to yield the address of the SPTE that maps the base of the memory to be deallocated.
7. The protection code is extracted from the first SPTE and saved.
8. Using the address of the first SPTE, the required number of SPTEs is cleared to unmap the virtual memory.
9. The address translation buffers on all active processors are invalidated.
10. If the extracted protection code is PTEK_SYSTEM, KERFREE_FRAME is called to release the page frames occupied by the deallocated memory. If the code is PTE$K_MESSAGE — meaning that no physical memory was allocated when KER$ALLOCATE_REGION was called — no effort is made to deallocate the page frames.
11. The routine returns to its caller.

9.3.1.2 KER$ALLOCATE_SYSTEM_REGION and KER$FREE_SYSTEM_REGION Kernel Procedures

The KER$ALLOCATE_SYSTEM_REGION procedure (in module SMEMORY) provides the public, external interface for allocating system virtual memory. The S0 memory allocated by this procedure is both virtually and physically contiguous. The procedure can be called only by jobs running in kernel mode. The calling job, however, need not be a kernel mode job; that is, KER$ALLOCATE_SYSTEM_REGION can be called through the KER$ENTER_KERNEL_CONTEXT procedure to elevate a user-mode job's access mode to perform the allocation. System pages allocated to a kernel-mode job are set with PTE$C_URKW protection; those allocated to a user-mode job receive PTE$C_UW protection.

Arguments to KER$ALLOCATE_SYSTEM_REGION specify the number of bytes to be allocated and an optional physical address to which the system addresses will be mapped. The procedure returns a status value and the virtual address of the memory allocated from the system region. The procedure executes as follows:

1. The current access mode of the caller is tested. If it is not kernel mode, KER$_BAD_MODE status is returned.
2. The number of bytes to be allocated is converted to the number of S0 pages required (and rounded up to the next page boundary if necessary).

3. The base access mode of the caller (in JCB$B_MODE) is tested. If the mode is kernel, the physical address and number of pages are passed to the internal subroutine KER$ALLOCATE_REGION. If the mode is user, control passes to an alternate entry point, KER$ALLOCATE_REGION_USER. The routine returns a success/failure value and the virtual address of the base of the S0 memory allocated.
4. If the allocation from the communication region succeeded, KER$ALLOCATE_SYSTEM_REGION returns the virtual address of the allocated memory to its caller. If the allocation failed, KER$_NO_MEMORY status is returned.

If a job that called KER$ALLOCATE_SYSTEM_REGION fails to call KER$FREE_SYSTEM_REGION to return its system memory before exiting, that memory is permanently lost to the system. Once a job has exited, its pointer to the allocated system memory is lost, and the kernel has no way of identifying communication region memory no longer under the control of a job.

Arguments to the KER$FREE_SYSTEM_REGION specify the number of bytes to be deallocated and the base virtual address of the memory to be freed. The procedure optionally returns a status value. The virtual address passed to the procedure should be the one returned by the original call to KER$ALLOCATE_SYSTEM_REGION. KER$FREE_SYSTEM_REGION executes as follows:

1. The current access mode of the caller is tested. If it is not kernel mode, KER$_BAD_MODE status is returned.
2. The virtual address argument is tested. If it is not an S0 address, KER$_BAD_VALUE status is returned.
3. The number of bytes to be deallocated is converted to the number of S0 pages (and rounded up to the next page boundary if necessary).
4. The base address of the memory and the number of pages to be freed are passed to the KER$FREE_REGION subroutine to perform the actual deallocation.
5. Control is returned to the caller.

9.3.2 Allocating User Virtual Memory

User virtual memory — P0 and P1 memory — is allocated dynamically by the kernel to accommodate the memory demands of a job and its subprocesses. P0 space is used to map program image sections (including code and global data), the job context page, and dynamic memory such as heap data and message buffers (see Figure 4–5). P1 memory is allocated to hold a process's private memory: a single debugger context page and its stack or stacks (see Figure 4–6).

P0 memory is allocated by a number of kernel and run-time library services, such as the following:

- KER$CREATE_JOB to map a program image
- KER$CREATE_MESSAGE to map a message buffer
- KER$CREATE_AREA to map an area buffer
- The program loader to map dynamically loaded programs
- The heap routines (NEW, **calloc** to allocate memory for a job's dynamic heap)

Since P1 memory has a more specialized application, it is allocated by a smaller number of kernel and run-time library services:

- KER$CREATE_JOB to allocate a master process's stacks
- KER$CREATE_PROCESS to allocate a subprocess's stacks
- The internal procedure KER$EXPAND_STACK (in module EXCEPTION) to expand the user stack dynamically when a stack reference raises an access violation
- The stack utility, ELN$ALLOCATE_STACK, to expand the user or kernel stack under program control

To allocate user memory, the following steps are required:

1. The required number of P0 or P1 page table entries must be allocated from the appropriate page table.
2. The fields in the allocated PTEs must be set to contain the appropriate protection, owner, type, and PFN fields. If the memory is allocated at a specified physical address, that PFN is used; otherwise, KER$ALLOCATE_FRAME is called to obtain a PFN.
3. The job and process memory structures affected by the allocation are updated.

The first of these steps is performed by the internal kernel subroutines KER$ALLOCATE_P0_PTE and KER$ALLOCATE_P1_PTE, and are described in Section 9.3.2.1. The other steps must be performed by the caller of the PTE allocation routine, for example, KER$CREATE_JOB or KER$CREATE_MESSAGE. Under program control, these steps are performed on behalf of the job by KER$ALLOCATE_MEMORY, the public procedure for the dynamic allocation of P0 and P1 memory. KER$ALLOCATE_MEMORY is described in Section 9.3.2.2. This section also briefly contrasts KER$ALLOCATE_MEMORY with the heap-management routine PAS$NEW2, which itself calls KER$ALLOCATE_MEMORY to support language-specific allocation of dynamic memory.

9.3.2.1 Allocating and Deallocating User Page Table Entries

The allocation of P0 or P1 virtual memory begins with the allocation of page table entries from the appropriate page table. One PTE must be allocated to map each page of virtual memory to a page frame. The internal subroutines KER$ALLOCATE_P0_PTE and KER$ALLOCATE_P1_PTE allocate P0 and P1 PTEs, respectively. Deallocation of P0 or P1 virtual address space concludes with the clearing of the appropriate PTEs that mapped the deallocated memory. The internal subroutines KER$FREE_P0_PTE and KER$FREE_P1_PTE deallocate P0 and P1 PTEs, respectively.

These subroutines manage the allocation of P0 and P1 memory using the P0 and P1 allocation bitmaps and their associated descriptors, resident in the JCB for P0 memory and the PCB for P1 memory. Each bit in these bitmaps represents a page of virtual memory, and the position of the bit within the bitmap represents the VPN of the page within the address space. The VPN in turn represents the P0 or P1 PTE allocated. (These values are adjusted to reflect the fact that P1 memory and page tables grow toward lower addresses.)

On all processors except the KA620, user address space is expanded dynamically in increments of 128 pages up to the limit set for the address space in the System Builder. This 128-page expansion represents the addition of one page frame (holding 128 PTEs) to the P0 or P1 page table. As each page is added to the page table, that page is mapped in the system page table. On non-KA620 systems, these page tables are virtually contiguous; on the KA620 systems, they are also physically contiguous, because the P0 and P1 page tables are created in their entirety the first time the address space is expanded during job and process creation.

When the P0 or P1 bitmap is created, it has no bits set, and the length of the bitmap is recorded in the bitmap descriptor as 0. The first time either of the PTE-allocation routines is called, the discovery of the zero-length bitmap leads to the immediate expansion of the page table by 128 PTEs — the setting of the first 128 bits in the bitmap. The routine then attempts to allocate the desired memory from that first 128 bits in the bitmap, the active portion of the bitmap.

Subsequently, the page table and bitmap are expanded whenever an allocation request cannot be satisfied within the active portion of the bitmap. When the bitmap has been expanded to its full length, the limit of P0 or P1 virtual memory for the job or process has been reached, and subsequent attempts to expand the page table and bitmap will fail. (This description of the expansion of the page table and bitmap does not apply to the KA620 processor; these elements are expanded to their maximum allowed length at the first allocation of user virtual memory by the KER$CREATE_JOB procedure.)

The operation of KER$ALLOCATE_P0_PTE and KER$ALLOCATE_P1_PTE are nearly identical, differing only to accommodate the fact that the P1 page table expands toward lower addresses. This means that the number of the bit allocated in the P1 memory bitmap must be altered to represent correctly the corresponding PTE in the P1 page table. The PTE allocation routines expect four input values:

- The number of PTEs to be allocated
- The address of the current JCB (P0 allocation) or PCB (P1 allocation)
- An explicit-allocation flag indicating whether the PTEs should be allocated for a specified virtual address
- If the explicit-allocation flag is set, the starting virtual address for which the PTEs should be allocated

The routines operate in two phases. The first phase is the simple allocation of bits from the active portion of the P0 or P1 bitmap. This phase returns the starting bit number (VPN) allocated and a success/failure value. (The discussion of KER$ALLOCATE_MEMORY in Section 9.3.2.2 shows how these return values are used to map the address space.) Within this simple allocation path, the execution path depends on whether the allocation is for an explicit set of PTEs (a specified virtual address) or simply for the first available set of PTEs.

The expansion of the page and bitmap is the second phase of execution. This path through the routine is followed only when simple allocation within the active portion of the bitmap fails. If the expansion succeeds, the routine returns to the first phase for a second attempt at simple allocation.

The simple allocation phase of KER$ALLOCATE_P0_PTE and KER$ALLOCATE_P1_PTE executes as follows:

1. The explicit allocation flag is tested.
2. If explicit allocation is not required, the following execution path is taken:
   a. The address of the appropriate allocation bitmap descriptor, located at JCB$A_P0_BITMAP or PCB$A_P1_BITMAP, and the number of PTEs to be allocated are passed as arguments to the local subroutine ALLOCATE_BITMAP. That subroutine attempts to allocate the specified number contiguous bit at the first available location within the active portion of the bitmap.
   b. If the bitmap allocation succeeds, success status and the starting bit number (the VPN) are returned to the caller. If the allocation is for P1PTEs, the bit number is adjusted to indicate the correct P1PTE.
   c. If the bitmap allocation fails, control branches to the code path to expand the bitmap and page table. Control then returns to the start of the first phase, and allocation is attempted with the expanded bitmap.

   If explicit allocation is required, the following path is taken:
   a. The VPN is extracted from the virtual address argument. This value represents the explicit starting bit number to be allocated in the bitmap. If the allocation is for P1PTEs, the bit number is adjusted for P1 space.
   b. The byte position within the bitmap containing the target bit is calculated.
   c. The byte position is compared to the value of BMP$W_LENGTH2 in the bitmap descriptor.
   d. If the byte position is beyond the active portion of the bitmap (that is, the byte position is greater than or equal to BMP$W_LENGTH2), control branches to the code path to expand the bitmap and page table. Control then returns to the start of the first phase, and allocation is attempted with the expanded bitmap.

e. The address of the appropriate bitmap descriptor, located at JCB$A_P0_BITMAP or PCB$A_P1_BITMAP, the explicit starting bit number, and the number of PTEs to be allocated are passed as arguments to the local subroutine EXAMINE_BITMAP. That subroutine attempts to allocate the specified number of contiguous bits starting at the specified location within the active portion of the bitmap.

f. If the bitmap allocation succeeds, success status and the starting bit number (the VPN) are returned to the caller. If the allocation is for P1PTEs, the bit number is adjusted to indicate the correct P1PTE.

g. If the bitmap allocation fails, the requested PTEs cannot be allocated; therefore, failure status is returned to the caller.

The expansion of the P0 and P1 page tables, the second phase of the allocate routines, occurs as follows:

1. The maximum possible size in bytes for the allocation bitmap is calculated by multiplying KER$GW_P0_SLOT_SIZE or KER$GW_P1_SLOT_SIZE (the number of longwords in the P0 or P1 page table) by 16, the number of bitmap bytes necessary to support one page of PTEs.
2. If that value equals the value of BMP$W_LENGTH2 in the bitmap descriptor, further expansion of the page table is impossible — the virtual address space has reached its maximum size as defined by the user in the System Builder. Failure status is returned to the caller.
3. The internal subroutine KER$ALLOCATE_FRAME is called to allocate physical memory for the new page table page. On non-KA620 processors, one page frame is requested. On the KA620 processor, all the page frames that the page table may require are allocated at once. The values of KER$GW_P0_SLOT_SIZE and KER$GW_P1_SLOT_SIZE represent the number of page frames required.
4. The values of BMP$W_LENGTH1 and BMP$W_LENGTH2 in the bitmap descriptor are updated to reflect the expansion of the active portion of the bitmap.
5. The bits in the expanded portion of the bitmap are set to indicate that the corresponding PTEs are now available for allocation. On the KA620 processor, all the bits in the bitmap are set at once.

6. The new page (or pages for the KA620 processor) is mapped in the portion of the system page table reserved for mapping this page table. The PFN returned by KER$ALLOCATE_FRAME is used to map the page table page or pages.
7. The new page or pages of the page table are zeroed to create null PTEs for the still unmapped memory created by the expansion of the page table.
8. For the allocation of P0PTEs, the new P0 limit — resulting from the expansion of the page table — is inserted into the PTX$L_P0_LIMIT field in the hardware context block of each process in the job. For the allocation of P1PTEs, the new P1 limit is inserted into the PTX$L_P1_LIMIT field of the current process's hardware context block. The caller of KER$ALLOCATE_P0_PTE or KER$ALLOCATE_P1_PTE is responsible for updating the P0LR or P1LR register for the job or process for which it is allocating memory.
9. Control is transferred back to the first phase of execution so that PTEs can be allocated from the newly expanded page table.

Kernel procedures that deallocate user memory, such as KER$FREE_MEMORY, free the affected PTEs at the conclusion of a larger process that includes such operations as verifying the ownership of the PTEs and freeing their associated page frames.

Freeing the PTEs themselves is a straightforward process: the PTEs that mapped the deallocated memory are cleared and the corresponding bits in the allocation bitmap are reset. KER$FREE_P0_PTE and KER$FREE_P1_PTE expect three input values:

- The virtual address of the first PTE to be deallocated
- The number of PTEs to be deallocated
- The address of the current JCB (P0 deallocation) or PCB (P1 deallocation)

Given these values, the routines execute as follows:

1. The virtual address of the virtual page associated with the first PTE to be deallocated is calculated.
2. That virtual address is moved to the TBIS (translate buffer invalidate, single) privileged register to invalidate the translation buffer entry for that page.
3. Using the virtual address argument, the associate PTE is cleared.
4. The virtual address of the next PTE is calculated.

5. The virtual address of the next page whose PTE is being deallocated is calculated.
6. Control branches back four steps until all the PTEs have been cleared.
7. The address of the bitmap descriptor, the starting bit number, and the number of PTEs to be freed are passed to the local subroutine FREE_BITMAP. That routine locates the specified bit or bits in the bitmap, resets them, updates the bitmap descriptor, and returns.
8. Control returns to the caller.

9.3.2.2 Allocating User Memory Under Program Control: KER$ALLOCATE_MEMORY

A number of kernel procedures allocate user virtual memory on behalf of a job or process, and each must use KER$ALLOCATE_P0_PTE or KER$ALLOCATE_P1_PTE to obtain the PTEs to map the user memory being allocated. Procedures such as KER$CREATE_JOB allocate user memory while mapping a program image into job memory. Jobs can allocate P0 directly by calling the kernel procedure KER$ALLOCATE_MEMORY or by using the heap routines NEW in Pascal, and **calloc**, **malloc**, or **realloc** in C.

The public kernel procedures KER$ALLOCATE_MEMORY and KER$FREE_MEMORY (in module VMEMORY) provide the direct means for user programs to allocate and deallocate P0 and P1 virtual memory. The direct allocation of P1 memory by a user program is uncommon but is supported to allow run-time library procedures, such as ELN$ALLOCATE_STACK, to call KER$ALLOCATE_MEMORY to expand the process stack in P1 space.

The heap routine PAS$NEW2 calls KER$ALLOCATE_MEMORY to allocate P0 memory to hold heap data in response to Pascal and C calls to NEW and **calloc**. The heap routines provide a convenient means for temporarily allocating small segments of P0 memory; they maintain a list of all blocks of memory allocated from the heap. When called upon for a block of memory, PAS$NEW2 attempts to satisfy the request from memory on the list of free memory. If none of those blocks is large enough, KER$ALLOCATE_MEMORY is called to obtain a block of adequate size.

When PAS$DISPOSE2 is called to deallocate a block of heap memory, the routine simply returns the block to the heap's free list — it does not call KER$FREE_MEMORY to free the page frames and P0 memory. The memory remains allocated in the job's address space and can be reused by subsequent calls to PAS$NEW2. KER$ALLOCATE_MEMORY, therefore, provides the preferred method for allocating large blocks — a page or greater — of dynamic memory. When this memory is deallocated with KER$FREE_MEMORY, the resources consumed by the memory — page frames and PTEs — are returned for later use.

KER$ALLOCATE_MEMORY supports a number of options for the allocation of user virtual memory:

- Memory can be allocated at the first available location in P0 space.
- Memory can be allocated at a specific virtual address in P0 or P1 space. P1 memory can be allocated only at a specified address.
- The memory can be allocated at a specific physical address. This option enables jobs to map hardware devices that must reside at specific physical addresses into job memory. The allocation can be made only by jobs running in kernel mode, although they can reach that access mode by calling KER$ALLOCATE_MEMORY through KER$ENTER_KERNEL_CONTEXT.
- Combining the second and third options, the memory can be allocated at specific virtual and physical address.

The job mode of the caller is used to set the protection on the virtual pages allocated. Pages allocated to a kernel-mode job are set with PTE$C_URKW protection; those allocated to a user-mode job receive PTE$C_UW protection.

Arguments to KER$ALLOCATE_MEMORY specify the number of bytes to be allocated and optional physical and virtual addresses for the allocation. The procedure returns a status value and the virtual address of the memory allocated. The procedure executes as follows:

1. The number of bytes to be allocated is converted to the number of pages required (and rounded up to the next page boundary if necessary).
2. The virtual address argument is tested. If it is an S0 address, KER$_BAD_VALUE is returned.
3. If the virtual address argument is nonzero, an explicit allocation has been requested. A flag is set to indicate this to the PTE allocation routine.

4. If an explicit virtual allocation is required, the size argument is used to calculate the ending address of the area to be allocated. If the address falls into a different address space than the base address, KER$_BAD_VALUE is returned.
5. The physical address argument is tested. If an address is specified, the base access mode of the caller (in JCB$B_MODE) must be kernel. If not, KER$_BAD_MODE is returned.
6. The number of PTEs required is passed to the KER$ALLOCATE_P0_PTE or KER$ALLOCATE_P1_PTE subroutine. If the explicit allocation flag was set earlier, the virtual address argument is passed as well. For P1 memory, the allocation is always to an explicit virtual address.
7. If the subroutine failed to allocate the required number of contiguous PTEs, KER$_NO_VIRTUAL status is returned to the caller.
8. The length of the page table, which may have been updated during the allocation of PTEs, is copied from JCB$L_P0_LIMIT into the P0LR register or from PCB$L_P1_LIMIT into the P1LR register.
9. If explicit physical allocation is required, no page frames are allocated. Instead, the base PFN is extracted from the physical address argument. The result is the allocation of physically contiguous memory. Control transfers ahead three steps to create a PTE.
10. The internal subroutine KER$ALLOCATE_FRAME is called to allocate one page frame. Allocating the frames one at a time means that the allocated memory will not be physically contiguous, making efficient use of virtual memory. The memory will be virtually contiguous.
11. If the page frame allocation fails, the allocation cannot continue. All frames and PTEs allocated to this point are freed, and the translation buffer is invalidated. KER$_NO_MEMORY is returned to the caller.
12. If explicit physical allocation is not required, the PFN returned by KER$ALLOCATE_FRAME and the prototype PTE in JCB$L_RW_DATA_PTE are used to set the PTE for the allocated page. The allocated page is zeroed.

    If explicit physical allocation is required, the specified PFN is inserted into the allocated PTE and the PTE type is set to PTE$K_SYSTEM. If the job mode is user, protection is set to PTE$C_UW and ownership to PTE$C_UOWN. If the job mode is kernel, protection is set to PTE$C_URKW and ownership to PTE$C_

KOWN. The protection of P1 pages is set to PTE$C_URKW and the ownership to PTE$C_KOWN.

13. If more pages remain to be allocated, the process of obtaining a PFN and setting the PTE is repeated until all pages are allocated.
14. Using the bit number (VPN) returned by KER$ALLOCATE_P0_PTE or KER$ALLOCATE_P1_PTE, the starting virtual address of the allocated memory is calculated and returned to the caller.

P0 and P1 memory can be deallocated under program control with the public kernel procedure KER$FREE_MEMORY. The procedure frees user memory allocated with KER$ALLOCATE_MEMORY and through other means such as job and process creation. KER$FREE_MEMORY does not free memory allocated in P0 space by the KER$CREATE_MESSAGE and KER$CREATE_AREA procedures. Such memory must be deallocated explicitly with the KER$DELETE procedure. A user-mode job can free only pages marked with PTE$C_UOWN ownership, even if the job is running temporarily in kernel mode.

Arguments to KER$FREE_MEMORY specify the number of bytes to be freed and the virtual addresses at which the deallocation should begin. The procedure returns a status value and executes as follows:

1. The previous mode field is extracted from the current PSL. This value indicates the access mode that the caller held before calling KER$FREE_MEMORY. If the procedure has been called through KER$ENTER_KERNEL_CONTEXT, the mode will be kernel.
2. The number of bytes to be deallocated is converted to the number of pages required (and rounded up to the next page boundary if necessary).
3. The virtual address argument is tested. If it is an S0 address, KER$_BAD_VALUE is returned.
4. If the virtual address is not page-aligned, it is truncated to the previous page boundary.
5. The virtual address and the size argument are used to calculate the ending address of the area to be deallocated. If the address falls into a different address space than the base address, KER$_BAD_VALUE is returned.
6. The starting virtual page number is extracted from the virtual address.

7. The VPN is used as an index into the page table to determine the address of the PTE that maps the first page of the memory to be freed. For P0 memory, the value of JCB$A_P0_BASE is used to locate the page table; for P1 memory, PCB$A_P1_BASE, adjusted for P1 space, is used.
8. The VPN is compared to the length of the page table. If the VPN does not fall within the table, no page remains to be freed, and control returns to the caller.
9. The PTE is tested. If it is null, the next VPN is calculated, and control branches back two steps to process the next PTE.
10. The type field is extracted from the PTE. The steps taken to free the virtual page that the PTE maps depend on the type.
11. If the type is PTE$K_RW_DATA, then the page frame that the virtual page maps must be freed. Therefore, the following steps are taken:
    a. The ownership field in the PTE is tested. If the previous mode of the caller is less privileged than the ownership set in the PTE, the page is not freed. The next VPN is calculated, and control branches back to process the next PTE.
    b. The PFN is extracted from the PTE and is passed to the subroutine KER$ALLOCATE_FRAME to free the page frame.
    c. Execution then continues with the next step — the first step for PTEs with PTE$K_SYSTEM as their type — to clear the PTE itself.
12. If the type is PTE$K_SYSTEM or PTE$K_USER_CODE (valid only in P0 space), then the page frame the virtual page maps need not be freed. Therefore, the following steps are taken (these steps also apply to PTEs of type PTE$K_RW_DATA):
    a. The PTE is cleared to unmap the virtual page.
    b. The address of the cleared PTE is passed to the KER$FREE_P0_PTE or KER$FREE_P1_PTE to reset the appropriate bits in the P0 or P1 allocation bitmap.
    c. Execution then continues with the next step to calculate the next VPN.
13. If the type is PTE$K_MESSAGE (valid only in P0 space), no memory is freed. Pages of this type can be freed only with the KER$DELETE procedure. Therefore, the next VPN is calculated, and control branches back to process the next PTE.

14. If an unexpected PTE type is encountered, the kernel bugchecks. For example, if a PTE of type PTE$K_MESSAGE were to appear in the P1 page table, a bugcheck would result.

9.4 Allocating System Pool

The system pool, consisting of a user-specified number of fixed-length blocks of S0 memory, supports the creation of kernel objects and other structures that support real-time operations such as device handling and synchronization. The kernel uses pool blocks for the following purposes:

- Kernel objects. Area, device, event, message, name, process, and semaphore data structures reside in pool blocks. See Chapter 10.
- Object pointer tables. These pool blocks contain pointers to a job's kernel objects. See Chapter 10.
- Job parameter blocks. These pool blocks temporarily contain the arguments passed to a job from the System Builder or the KER$CREATE_JOB procedure.
- Interrupt dispatch block. This block contains a procedure that calls an interrupt service routine to handle a hardware interrupt.
- Wait control blocks. These blocks describe the conditions of a process's entrance to and exit from a wait state. One pool block can hold three wait control blocks.

The system pool contains a user-specified number of fixed-length pool blocks. Physical memory for fixed-length system pool blocks is allocated during system initialization and mapped into a range of system virtual addresses reserved by the kernel. The creation of the system pool is described in Section 9.4.1. System pool blocks cannot be directly allocated under program control. Rather, the kernel allocates and frees pool blocks on behalf of user programs using the KER$ALLOCATE_POOL and KER$FREE_POOL procedures described in Section 9.4.2.

9.4.1 Initializing System Pool

The system pool is created in S0 address space during the system initialization sequence with memory mapping enabled. The entire sequence is described in Chapter 3. The number of system pages devoted to the pool is determined by the global value KER$GW_POOL_SIZE. The size of a pool block is set by the assembly-time symbol OBJ$K_LENGTH. The current value is 128 bytes, meaning that each page of system pool contains four pool blocks.

The creation of the system pool, in module INITIAL, occurs as follows:

1. The number of pages to be mapped for the pool is obtained from KER$GW_POOL_SIZE.
2. The current system virtual address is copied to the global value KER$GA_POOL_BASE to mark the base address of the pool.
3. Control branches to the local subroutine GET_FRAME to obtain a page of physical memory and map it in the system page table.
4. The number of pool blocks within each page is calculated by dividing the size of a page (512 bytes) by OBJ$K_LENGTH (128 bytes) yielding four blocks for each page.
5. The virtual addresses of each pool block within the page frame are passed one at a time to the internal subroutine KER$FREE_POOL (see Section 9.4.2) to enter them onto the list of free pool blocks.
6. Steps 3 through 5 are repeated until all the pages in the pool are divided into pool blocks and inserted into the free list.

9.4.2 Allocating and Deallocating Pool Blocks

The allocation of pool blocks requires the cooperation of the allocating routine and the KER$ALLOCATE_POOL subroutine (in module ALLOCATE). The allocation routine only reserves a specified number of pool blocks for its caller; it is up to the caller to remove the reserved blocks from the list of free blocks. A kernel procedure, then, obtains a pool block in two stages:

1. It reserves the number of pool blocks it requires by passing that number to the internal subroutine KER$ALLOCATE_POOL (using the ALLOCATE POOL kernel macro).

2. It removes the reserved pool blocks as they are needed with the REMOVE kernel macro. REMOVE generates a REMQHI instruction to obtain the address of the pool block and remove it from the free list. The address is then used to manipulate the block as required.

This allocation sequence is illustrated in the discussion of creating kernel objects in Chapter 10.

KER$ALLOCATE_POOL manipulates the global value KER$GL_POOL_FREE. When the pool is created, this value contains the *negated* number of blocks in the pool. When a block is allocated, KER$GL_POOL_FREE is increased by one. (When the value reaches zero, the pool is empty. If the increment of KER$GL_POOL_FREE exceeds 0, a failure flag is returned to the caller. The caller will in turn return KER$_NO_POOL status to its caller.) KER$ALLOCATE_POOL then returns the address of the global quadword value KER$GQ_POOL_HEAD to the caller. The REMQHI instruction generated by the caller's subsequent invocation of the REMOVE macro uses this value to locate and remove each pool block from the queue of free blocks.

Pool blocks are returned to the pool by passing the address of the block to the internal subroutine KER$FREE_POOL, invoked from kernel procedures with the FREE POOL macro. That subroutine subtracts one from the value of KER$GL_POOL_FREE to reflect the addition of one block to the pool. It then executes the INSQTI instruction — with the address of the block and KER$GQ_POOL_HEAD as its operand specifier — to insert the block back into the queue of free blocks, and returns to its caller. The freeing of a pool block during object deletion is described in Chapter 10.

Chapter 10

Kernel Objects and Their Management

A VAXELN kernel object is a data structure in system address space that defines the state of a user-controlled system activity or resource. Kernel objects give VAXELN jobs the means to create subprocesses, synchronize execution, exchange messages, share memory, and control hardware devices.

The VAXELN Kernel creates most kernel objects for a job using the system dynamic pool. The process is initiated by an object-creation procedure call, such as KER$CREATE_PROCESS. The call returns an identifier value to the caller to represent the created object in all subsequent operations on the object. Depending on its type, a kernel object has an identifier that is unique within a job or unique within an entire VAXELN system.

Job-specific objects, whose identifiers are unique within a job, include the following objects:

- Area
- Device
- Event
- Message
- Name
- Process
- Semaphore

The only systemwide object is the port. Ports require unique identification throughout a system and therefore cannot be unique to any one job.

This chapter describes how kernel objects are created, manipulated, and managed. No detailed description of their structures and uses appears here — those discussions arise throughout this book and in the VAXELN user documentation. For the remainder of this chapter, the term *kernel objects* refers to the job-specific objects, and the term *port object* refers to the port object.

10.1 Creating, Managing, and Deleting Kernel Objects

At any given moment in its execution, a VAXELN job can contain up to 4096 kernel objects. Each object has a unique 32-bit identifier that the kernel uses to locate the address of the object in a two-tiered set of address tables. This two-level system of address tables helps the kernel protect and manage objects by hiding the actual system addresses of the objects from user jobs, preventing uncontrolled access. The system also allows for rapid validation of object operations and the orderly dynamic creation and deletion of objects.

When a job creates an object, the kernel takes the following steps on the caller's behalf:

1. It obtains a pool block from the system pool to contain the requested object. Process objects, however, occupy a 512-byte page of system memory from the communication region.
2. It formulates a unique identifier that represents an unused entry in a table of pointers to the job's kernel objects.
3. It fills in the object fields as appropriate for the requested object.
4. It inserts the address of the allocated object into the table entry represented by the new object identifier.
5. It returns the identifier to the calling job in a longword variable supplied by the caller.

When the job that created the object subsequently passes the object identifier variable in a kernel procedure call, the kernel uses the identifier value to look up the address of the associated kernel object in the job's table of pointers. Using this address, the kernel can then perform the requested operation on the object. When the job deletes the object, the kernel takes the following steps:

1. It uses the identifier value to locate the table entry containing the address of the object to be deleted.
2. It performs the necessary actions on the object to allow it to be removed from the job.
3. It returns the pool block occupied by the object to the system pool. For process objects, the page of system memory is returned to the communication region.
4. It removes the object address from the table entry and replaces it with a prototype object identifier.

The sections that follow describe the data structures involved in object processing and discuss creating and deleting kernel objects in more detail.

10.1.1 Structures and Data for Managing Kernel Objects

The kernel maintains a number of data structures and data items that support the creation, use, and deletion of kernel objects:

- The values for the jobwide data items JCB$A_OBJECT_TABLE and JCB$W_OBJECT_FREE. These values, stored in the job control block (JCB), are used to locate the object base table and to locate available object pointer table entries, respectively.
- The object base table, which contains the addresses of the object pointer tables. This table forms the first tier in the two-tiered arrangement of address tables and can hold the addresses of up to 128 object pointer tables.
- The object pointer tables, which contain the addresses of up to 32 kernel objects. These tables — up to 128 of them — form the second tier of address tables.
- The object identifier, a value that, when translated, locates an entry in an object pointer table that contains the address of an object.
- The kernel objects themselves, which contain the information that defines the type and state of an object.

The following sections describe these data items and structures and how the kernel uses them. They are described roughly in the order that they are created by the kernel when a VAXELN job is created.

10.1.1.1 Jobwide Data Items

The longword value JCB$A_OBJECT_TABLE contains the address of the system page that contains the object base table, which in turn holds pointers to existing object pointer tables. This value exists for the life of a job and is used as a base address in VAX indexed instructions to look up the address of a particular object pointer table.

The 16-bit value JCB$W_OBJECT_FREE contains, in encoded form, the location of the next available entry in the object pointer tables. This pointer-like value is used and updated during each kernel object creation and deletion.

When a object is being created, the pointer table entry that its address will occupy is determined by the value OF JCB$W_OBJECT_FREE. When the object's address is entered into the pointer table, the prototype identifier value formerly stored in that entry becomes the new value of JCB$W_OBJECT_FREE. Because each prototype identifier points indirectly to the next entry in the pointer table, the new value JCB$W_OBJECT_FREE points to the table entry after the one now occupied by the new object's address. The next object created will therefore occupy the next table entry, which again is pointed to by JCB$W_OBJECT_FREE.

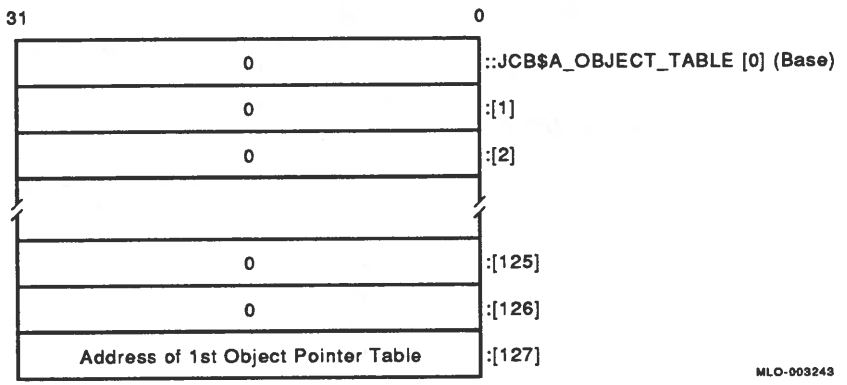
The functions of both jobwide values in object operations are described further throughout the remainder of this chapter.

10.1.1.2 Base Table

The base table contains the addresses of dynamically created object pointer tables. The base table occupies one 512-byte page in the system communication region, and, therefore, can contain up to 128 (512/4) longword pointers to object pointer tables. A single base table is created by the KER$CREATE_JOB procedure. The address of the base table is stored in the JCB$A_OBJECT_TABLE field in the job's JCB. The table exists until the job is deleted.

The KER$CREATE_JOB procedure creates and initializes the base table just before it creates the master process for the job; the master process is the first object created in any job. After job creation, the job's base table appears as shown in Figure 10–1. In the figure, the bracketed decimal numbers represent the zero-based index used to access the longword entries in the table.

Figure 10–1: Base Table



The kernel fills the table starting from its end. As the figure shows, all longwords but the last are initialized to zero. The last longword contains the address of the first dynamically allocated object pointer table, capable of holding the job's first 32 objects. When this original object pointer table is full, the kernel allocates another pointer table and places its address in the next-to-last longword in the base table, immediately preceding the address of the first pointer table.

The pointer-table addresses in the base table are accessed through VAX indexed instructions (in displacement-deferred indexed addressing mode), using the base field in the object identifier as the index into the base table to indicate which pointer table contains the object. (Object identifiers are described in Section 10.1.1.4). For example, if register R7 contains the address of the JCB, and R0 holds the base field from the object identifier, then after the following instruction, R1 will contain the address of the object pointer table that contains the object:

```
MOVL    @JCB$A_OBJECT_TABLE(R7)[R0],R1
```

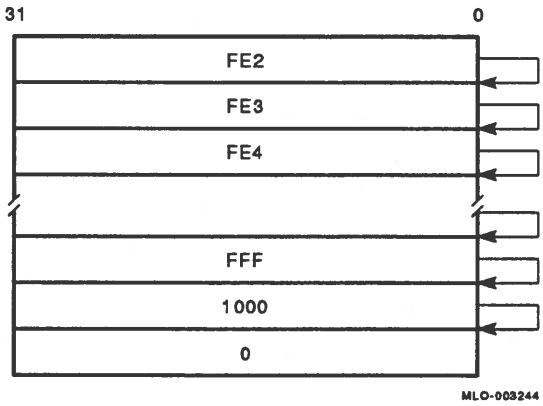
Once it has obtained the address of the object pointer table from the base table, the kernel can then look up the address of the actual kernel object in that pointer table.

10.1.1.3 Object Pointer Tables

An object pointer table contains the addresses of dynamically created kernel objects. A pointer table occupies one 128-byte pool block and therefore can contain the addresses of up to 32 (128/4) kernel objects. Because the base table can point to 128 pointer tables, and each pointer table can point to 32 objects, a job can therefore contain up to 4096 objects (128*32).

A job has at least one object pointer table created by KER$CREATE_JOB. When one table fills with object addresses, the kernel allocates another (with the KER$ALLOCATE_OBJECT procedure in module ALLOCATE) and places its address in the base table. Once a pointer table has been allocated, it exists until the job is deleted, even though the objects the table points to may no longer exist. When a newly allocated table is initialized, as shown in Figure 10–2, prototype object identifiers act as placeholders for the table entries.

Figure 10–2: First Object Pointer Table, after Initialization



These prototype identifiers, however, are more than placeholders. The values assigned to them play a crucial role in the creation of kernel objects. Each prototype identifier represents the base and index values that point to the next available object pointer table entry (the base value represents an entry in the base table, and the index value represents an entry in the object table pointed to). The arrows in Figure 10–2 emphasize how each prototype identifier, when decoded to its base and index components, points to the table entry that follows it. Section 10.1.1.4 describes in greater detail the role of the base and index values in the creation of object identifiers.

During object creation (described in Section 10.1.2), the value of the prototype identifier becomes the new value of JCB$W_OBJECT_FREE, so that it points to the next available pointer table entry. The previous value of JCB$W_OBJECT_FREE becomes the identifier for the newly created object.

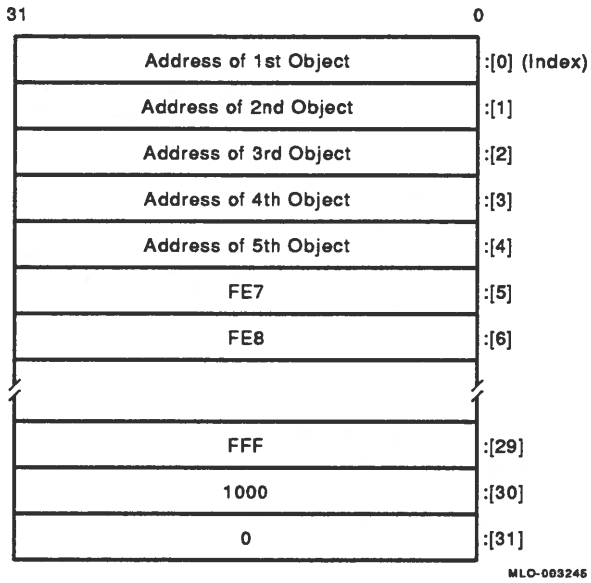
The final longword in the pointer table is initialized to 0 to mark the end of the table.

As kernel objects are created, the kernel inserts their addresses into the pointer table, starting at the top. Figure 10–3 shows the first object pointer table after five objects have been created. The bracketed numbers represent the zero-based index used to access the longword entries in the table. When an object's address is inserted into the table, it displaces the prototype identifier, which becomes the new value for JCB$W_OBJECT_FREE. When the final entry in the table is filled, the value of JCB$W_OBJECT_FREE becomes 0, which will indicate, at the next object creation, that the table is full.

At the next object creation, a new pointer table is created and initialized. If the new pointer table is the second one allocated, its first prototype identifier will have the value FC2 to reflect the position of the new pointer table's address in the base table. That is, the value FC2 decodes to represent a base table index of 126, meaning that the pointer table address appears in the next-to-last longword in the base table.

The object addresses in the pointer table are accessed through VAX indexed instructions (in register-deferred indexed addressing mode) using the index field in the object identifier as the index into the pointer table. For example, if register R0 contains the address of the object pointer table and R1 holds the index field from the object

Figure 10–3: First Object Pointer Table after the Creation of Five Objects



identifier, after the following instruction, R2 will contain the address of the identified object:

```
MOVL    R0[R1],R2
```

Once it has obtained the address of the object from the pointer table, the kernel manipulates the object as required.

10.1.1.4 Object Identifiers

When a job creates an object, it receives an identifier value in return to support further operations on the new object. All KER$CREATE procedures (such as KER$CREATE_EVENT) require their callers to supply a writeable longword variable to receive the identifier for the created object. This identifier should not be confused with the object itself, which resides in system space, or with the address of the object, which resides in an object pointer table. In fact, object identifiers usually have values that fall into the VAX S1 virtual address range;

therefore, using them in address expressions will result in run-time access violations.

Figure 10–4 shows the structure of an object identifier, and Table 10–1 describes the significance of the bit fields within the 32-bit identifier.

Figure 10–4: Structure of an Object Identifier

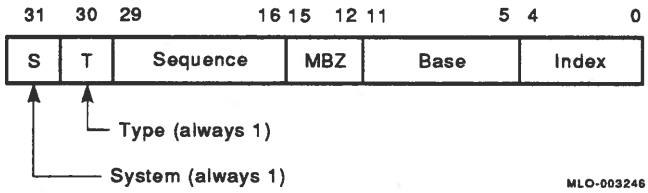


Table 10–1: Bit Fields Within the Object Identifier

Bit field	Size	Meaning
Index	5	The longword offset into the object pointer table to the address of the identified object. The maximum value this field can hold is 31, the highest valid index into the object pointer table.
Base	7	The longword offset into the base table to the address of the object pointer table. The maximum value this field can hold is 127, the highest valid index into the object base table.
Sequence	14	The generation number of the object in its object pointer table entry; used to detect mismatches between an identifier and an object.

Table 10–1 (Cont.): Bit Fields Within the Object Identifier

Bit field	Size	Meaning
Type	1	When set, indicates that the identified object is a job-specific object, that is, not a port object. (This bit and the system bit are required to distinguish an object identifier from a the address of a port identifier. If the port identifier has an S0 address, the bit corresponding to the system bit will be set, making it impossible to distinguish the address of a port identifier from an object identifier using the system bit alone.)
System	1	When set, forces the identifier to have a value that appears to be a system address; specifically, the setting of this bit and the type bit causes the identifier to appear as an S1 address ($C0000000_{16}$ and higher).

The fields in the identifier are represented within kernel code by symbolic values. Table 10–2 shows the names, values, and significance of the symbolic values the kernel uses when manipulating object identifiers.

Table 10–2: Assembly-Time Symbols Representing Object Identifier Bit Fields

Symbol	Value	Meaning
JID$S_BASE	7	The size of the base bit field
JID$V_BASE	5	The starting bit number of the base bit field
JID$S_INDEX	5	The size of the index bit field
JID$V_INDEX	0	The starting bit number of the index bit field
JID$S_SEQUENCE	14	Thes size of the sequence bit field
JID$V_SEQUENCE	15	The starting bit number of the sequence bit field
JID$V_SYSTEM	31	The starting bit number of the system bit field

Table 10–2 (Cont.): Assembly-Time Symbols Representing Object Identifier Bit Fields

Symbol	Value	Meaning
JID$V_TYPE	30	The starting bit number of the type bit field

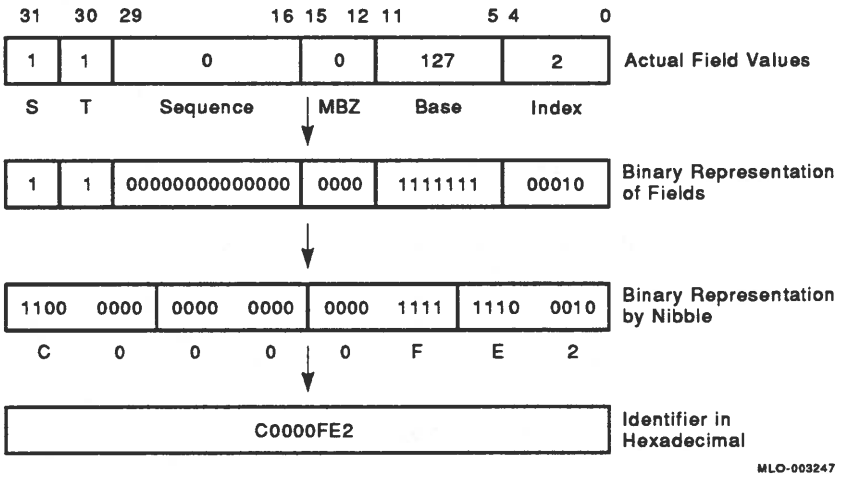
To form the identifier for the object it is creating, the kernel takes the following steps:

1. It inserts the longword index within the base table of the address of the current pointer table into the base field of the identifier. In other words, if the address of the current object pointer table appears in, say, the 128th entry in the base table, the base value becomes 127 (with a zero-based index).
2. It inserts the longword index within the object pointer table of the next available table entry into the index field of the identifier. In other words, if the address of the created object will appear in, say, the third longword in the pointer table, the index value becomes 2 (with a zero-based index).
3. It inserts the incremented sequence number for the object into the sequence field of the identifier.

Based on a base value of 127, an index value of 2, a sequence value of 0, and a value of 1 for both the system and type bits, an object identifier would be formed as shown in Figure 10–5, with a resulting value of C0000FE2. Later, when the resulting identifier is used in a kernel call, the kernel translates the value by extracting the index and base fields to locate the actual address of the identified object. This process is described in Section 10.1.3.

Through subsequent uses of the same object pointer table entry, the value of the sequence field will increase. This increase is reflected in the object identifier. For example, the identifier for the second object to use a table entry will contain a sequence number of 1. Although not reflected in Figure 10–5, the sequence number is represented by the most-significant four hexadecimal digits of the identifier. As the sequence number reaches its maximum value (16,383), the C that is normally the most-significant digit in the identifier is replaced, successively, by a D, an E, and an F.

Figure 10–5: Formation of an Object Identifier



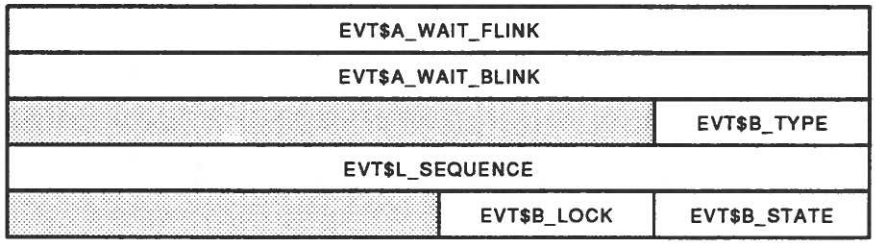
10.1.1.5 Kernel Object Structures

A kernel object occupies one 128-byte system pool block, which is allocated from the system pool when the object is created. An object therefore resides in system address space, where it is protected from uncontrolled access by user programs.

The structure of an object depends largely on its type. As an example, Figure 10–6 shows the structure of an event object.

All the fields in the event object (EVT), except the EVT$B_TYPE and EVT$L_SEQUENCE fields, are specialized to meet the requirements of event operations. The ninth byte in all kernel objects contains a unique value representing the object's type. Table 10–3 shows the constant values that can appear in an object's type field.

Figure 10–6: Structure of an Event Object



The type field is used by the kernel to ensure that an operation is appropriate for a particular object. For example, the KER$DELETE procedure is used to delete all object types; within the procedure, the kernel uses the type field to determine what kind of object will be deleted. It then takes the appropriate action to delete that type of object. The seuqnce field, which always occupies the fourth longword in an object, is used in validating object identifiers.

Table 10–3: Kernel Constants That Identify Object Types

Constant/Type	Value
OBJ$K_DEVICE	1
OBJ$K_EVENT	2
OBJ$K_MESSAGE	4
OBJ$K_NAME	5
OBJ$K_PORT	6
OBJ$K_SEMAPHORE	7
OBJ$K_PROCESS	8
OBJ$K_AREA	13

10.1.2 Creating Kernel Objects

Kernel objects are created when a job calls one of the KER$CREATE procedures, such as KER$CREATE_PROCESS or KER$CREATE_EVENT. Each of these procedures is responsible for the aspects of object creation that are specific to its type of object, but all of them call the internal subroutine KER$ALLOCATE_OBJECT (in module ALLOCATE) for these tasks: generating an object identifier and returning the address of the entry in the object pointer table that will hold the address of the new object.

The kernel procedure KER$CREATE_EVENT (in module CREATEEVT) offers a brief example of object creation. The relevant portions of its code are shown in Figure 10–7. The procedure executes as follows:

1. The kernel macro ALLOCATE POOL generates a branch to a subroutine to allocate a pool block for the object.
2. The kernel macro REMOVE obtains the address of the pool block.
3. The kernel macro ALLOCATE OBJECT generates a branch to the routine KER$ALLOCATE_OBJECT. This routine generates an object identifier for the new event object and returns the address of the object pointer table entry where the object address will be stored.
4. The fields of the event object are initialized.
5. The address of the initialized event object (that is, the address of its pool block) is inserted into the object pointer table using the table entry address returned by KER$ALLOCATE_OBJECT. The sequence number for the allocated identifier is inserted into the EVT$L_SEQUENCE field of the event object.

The internal subroutine KER$ALLOCATE_OBJECT (invoked by the ALLOCATE kernel macro) plays a key role in object creation. The routine has these responsibilities:

- Generate and return an object identifier for the new object.
- Generate and return a sequence number for the new object.
- Return the address of the entry in the current object pointer table that will receive the address of the new object.

Figure 10–7: Creation of an Event Object

```
KER$CREATE_EVENT_S::
        .
        .
        .
        MOVL    #1,R0                   ; set number of pool blocks to allocate
        ALLOCATE POOL                   ; allocate event object pool block
                bsbw    KER$ALLOCATE_POOL
        BLBC    R0,20$                  ; if lower bit clear, failure
        REMOVE R6                       ; remove event object block from pool
                remqhi  (R1),R6
        ALLOCATE OBJECT                 ; allocate object table entry
                bsbw    KER$ALLOCATE_OBJECT
        BLBC    R0,30$                  ; failure if all tables full
                                        ; R1 contains ID, R2 object address
        .
        . initialize the Event object
        .
        MOVL    R6,(R2)                 ; store address of event object in pointer table
        MOVL    R3,EVT$L_SEQUENCE(R6)   ; store sequence number in event object
```

- If the current object pointer table is full, allocate and initialize a new table and place its address in the base table.

KER$ALLOCATE_OBJECT executes as follows to allocate an object for the calling KER$CREATE routine:

1. The value of JCB$W_OBJECT_FREE is obtained. This value encodes the location of the next available pointer table entry and therefore determines the object identifier for the new object. For example, before the very first object in a job (the master process) is created, JCB$W_OBJECT_FREE has the value $FE1_{16}$, representing the 127th base table entry (the first pointer table created) and the first entry in that pointer table.
2. If JCB$W_OBJECT_FREE is 0, this means that the current pointer table is full and a new table must be allocated (this process is examined later).

   If JCB$W_OBJECT_FREE is not 0, then the current table will hold the object address of the new object.
3. The value of JCB$W_OBJECT_FREE is reduced by 1 to generate the actual index value for new object's identifier. In other words, the nonzero value of JCB$W_OBJECT_FREE is always one higher than that of the object identifier it will generate. (This offset of 1 is required to adjust for the 0 used as the marker for the end of the pointer table.)

4. The base field is extracted from the adjusted value of JCB$W_OBJECT_FREE. This value represents the base value in the object identifier; that is, the value is the offset into the base table of the longword of the appropriate pointer table. For example, a base value of 127 indicates that the address of the pointer table appears in the last longword in the base table.
5. The index field is extracted from the adjusted value of JCB$W_OBJECT_FREE. This value represents the index value in the object identifier; that is, the value is the offset into the pointer table of the longword that will hold the address of the new object. For example, an index value of 2 indicates that the address of the object will appear in the third longword in the pointer table.
6. The address of the object pointer table is obtained from the appropriate entry in the base table by indexing the table by the base value obtained from JCB$W_OBJECT_FREE.
7. The address of the pointer table entry to hold the new object's address is calculated by using the index value as an index into the pointer table. For example, if the index value is 2, the address of the third entry in the pointer table is obtained. The KER$CREATE procedure that called KER$ALLOCATE_OBJECT will use this address to copy the address of the object it creates into the pointer table.
8. The prototype identifier value in the selected pointer table entry is copied to JCB$W_OBJECT_FREE. This means that when the next object is allocated, its address will be placed in the pointer table entry following that of the object just created. If the prototype entry copied to JCB$W_OBJECT_FREE is 0, then the pointer table is now full, and a new one will be allocated when the next object is created.
9. The sequence field value is extracted from the prototype identifier in the table entry and inserted into the same field in the object identifier. This sequence value is also returned to the caller so that the sequence can be set in the created object.
10. The system and type bits are set in the constructed object identifier. The KER$CREATE procedure that called KER$ALLOCATE_OBJECT returns the value to its caller as the identifier of the object it created. In the subsequent object operations, the identifier will allow the kernel to locate the pointer table entry that holds the address of the object.
11. The pointer to the allocated pointer table entry is returned to the KER$CREATE procedure.

When KER$ALLOCATE_OBJECT discovers the value of JCB$W_OBJECT_FREE to be 0, it must attempt to allocate a new object pointer table. It does so by branching to a series of instructions that execute as follows:

1. The base table is searched from bottom to top for an unused entry. If none is found, the procedure returns a failure status to the KER$CREATE procedure to indicate that no new object can be created by the job until another is deleted, freeing a pointer table entry.
2. If a table entry is available, a pool block is allocated to hold the table, and the address of the block is placed in the free base table entry. The offset of this entry in the base table will become the base field in the identifiers for all objects associated with the new pointer table.
3. That base field value is shifted into base bit field, and one is added to initialize the index field to 1. The resulting value ($FC1_{16}$ for the second pointer table) is then copied into JCB$W_OBJECT_FREE to establish a pointer to the first entry in the pointer table. The following instruction sequence is used:

```
ASHL    #JID$V_BASE,R2,R2          ; shift base offset into base field
INCL    R2                         ; adjust index for pointer table
MOVW    R2,JCB$W_OBJECT_FREE(R7)   ; set OBJECT_FREE for first entry
```

4. The table is then initialized by writing prototype identifier values to all but the last longword in the table; that entry becomes 0 to mark the end of the table. The prototype values are generated by looping to add 1 to the index field (which equals JCB$W_OBJECT_FREE) and writing the result to the first through 32nd entries in the table. For example, the first prototype value in the second pointer table will be $FC2_{16}$; the next to last will be $FE0_{16}$. The sequence field in the prototype identifiers is 0.
5. Control branches back to the start of KER$ALLOCATE_OBJECT, which then returns an object identifier, a sequence value, and the address of the first entry in the newly allocated pointer table to the calling KER$CREATE procedure. The first identifier returned for the first object associated with the second pointer table will be C0000FC0.

Figure 10–8 shows a job's object-management values and data structures as they would appear after the creation of 34 kernel objects. The addresses of the first 32 objects fill the first pointer table, whose address appears in the last longword in the base table. Therefore, objects associated with the first pointer table all have identifiers with a base value of 127. The addresses of the thirty-second and thirty-third objects appear in the first two longwords of the second pointer table, whose address appears in the next-to-last longword in the base table. Therefore, objects associated with the second pointer table all have identifiers with a base value of 126. This decrease in the base field value as each table is allocated accounts for the unique identifier values for the objects within the job.

The value of JCB$W_OBJECT_FREE, as shown in the figure, is $FC3_{16}$ and points to the pointer table entry in which the next object created will appear. Subtracting 1 from $FC3_{16}$ produces the value $FC2_{16}$, which decodes to a base value of 126 and an index value of 2; therefore, the address of the next object created will be in the third longword (with a zero-based index) in the pointer table whose address appears in the 127th longword in the base table (again, with a zero-based index).

10.1.3 Translating Object Identifiers

Object translation — the decoding of an object identifier into the address of the object it identifies — occurs whenever the kernel must manipulate an object. In the process of translation, the kernel validates the object identifier and the address it translates into.

The internal subroutine KER$TRANSLATE_OBJECT (in module KERNELSUB) translates objects. This routine is called by all kernel procedures that manipulate kernel objects, such as KER$DELETE and KER$SIGNAL_EVENT. The routine expects two inputs: the identifier value to be translated and the address of the JCB for the job that owns the object. When control returns to the calling procedure, the address of the identified object is located in R1. The kernel uses that address to access the object.

Figure 10–8: Kernel Object Management Structures after the Creation of 34 Objects

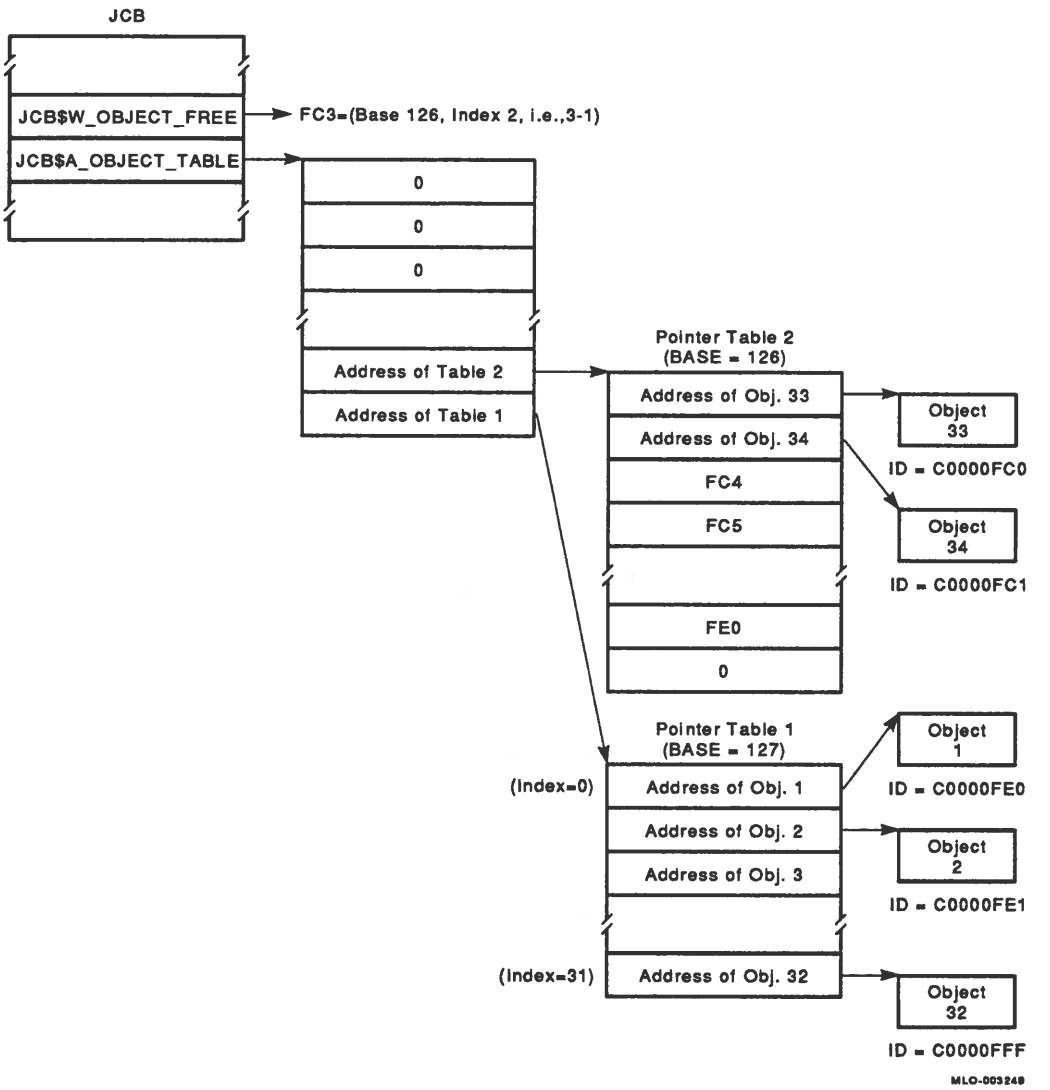


Figure 10–9 shows the brief code sequence used to validate and translate an object identifier. KER$TRANSLATE_OBJECT was designed to complete object translation — a common run-time operation — in as few instructions as possible. The data items and structures associated with object management exist to support that goal. The routine executes as follows:

1. The system and type bits in the identifier are tested. If they are set, then the identifier is valid. If not, control branches to the error exit for the routine.
2. The index field of the identifier is extracted.
3. The base field of the identifier is extracted.
4. The base value is used to obtain the address of the appropriate object pointer table in the base table. If the result is equal to 0, then the object identifier must be invalid, because it pointed to an empty entry in the base table; control branches to the error exit.
5. The sequence field in the identifier is compared to the sequence field in the translated object. If the two values do not match, the identifier is invalid, and control branches to the error exit. This mismatch means that an object pointer table has been reused.
6. The pointer table address and the pointer table index are used to obtain the system address of the object from the pointer table. If the value obtained from the table is 0 or greater, then the identifier must be invalid, because system addresses are interpreted as negative integers by the hardware (bit 31 set); control branches to the error exit.
7. Control is returned to the caller, with the address of the object in R1.

Figure 10–10 shows how KER$TRANSLATE_OBJECT is used in the kernel procedure KER$CLEAR_EVENT. The KER$CLEAR_EVENT procedure is passed the identifier of the event object in its argument list. It then passes that identifier to KER$TRANSLATE_OBJECT. If that routine succeeds, then R1 contains the address of the event object itself. That address is then used to access the fields in the object to validate its type and, finally, set its state to clear.

Figure 10–9: Kernel Object Translation with KER$TRANSLATE_OBJECT

```
KER$TRANSLATE_OBJECT::                  ; R0 = identifier, R7 = JCB
        CMPZV   #JID$V_TYPE,#2,R0,#3    ; system and type bits set in ID?
        BNEQ    10$                     ; if not, then invalid ID
        EXTZV   #JID$V_INDEX,#JID$S_INDEX,R0,R1
                                        ; extract index offset from ID
        EXTZV   #JID$V_BASE,#JID$S_BASE,R0,R2
                                        ; extract base offset from ID
        MOVL    @JCB$A_OBJECT_TABLE(R7)[R2],R2
                                        ; get address of pointer table
        BEQL    10$                     ; if the addr. is 0, then this
                                        ;    is a bad ID
        MOVL    (R2)[R1],R1             ; get address of object from table
        BGEQ    10$                     ; if result is >= 0, then it is
                                        ;    a prototype ID, not an address
        ASHL    #-JID$V_SEQUENCE,R0,R0  ; move sequence number to lower word
        BICL2   #^C<JID$M_SEQUENCE@-JID$V_SEQUENCE>,-
                R0                      ; mask out extraneous bits
        CMPL    R0,OBJ$L_SEQUENCE(R1)   ; compare sequence numbers
        BNEQ    10$                     ; if unequal, this is a bad ID
        MOVL    #1,R0                   ; set success indicator
        RSB                             ; return to caller

10$:    CLRL    R0                      ; set failure indicator
        RSB                             ; return with failure
```

Figure 10–10: Use of KER$TRANSLATE_OBJECT by KER$CLEAR_EVENT

```
KER$CLEAR_EVENT_S::
        .
        .
        .
        MOVL    EVENT(AP),R0            ; get event object ID from arg. list
        BSBW    KER$TRANSLATE_OBJECT    ; translate object ID to pointer
                                        ; address of object now in R1
        BLBC    R0,20$                  ; invalid ID -- return KER$_BAD_VALUE
        CMPB    #OBJ$K_EVENT,OBJ$B_TYPE(R1)
                                        ; is this an event object?
        BNEQ    30$                     ; not event -- return KER$_BAD_TYPE
        CLRB    EVT$B_STATE(R1)         ; set event state to cleared
        RETURN_STATUS   SUCCESS
20$:    RETRUN_STATUS   BAD_VALUE
30$:    RETURN_STATUS   BAD_TYPE
```

10.1.4 Deleting Objects

The deletion of a kernel object occurs when a job calls the KER$DELETE procedure, passing as an argument the variable that holds the identifier of the object to be deleted. In deleting an object, the kernel returns the pool block occupied by the object to the system pool and replaces the object's address in the pointer table with a prototype object identifier.

When a job exits (with the deletion of its master process), all the job's objects are deleted and the memory occupied by its base table and pointer tables is returned to the system.

The following sections describe these two levels of object deletion.

10.1.4.1 Deleting an Individual Kernel Object

When the KER$DELETE procedure is called to delete a kernel object, the following events occur:

- Object-specific actions are taken to clean up the object's associations within the job. For example, if the object is an event or a semaphore, any processes waiting on the object are unblocked with the KER$_BAD_VALUE status value.
- The object's pool block is returned to the system pool.
- The object's address is removed from the object pointer table and is replaced by a prototype object identifier, namely, the current value of JCB$W_OBJECT_FREE.
- The value of JCB$W_OBJECT_FREE is replaced by the adjusted value of the identifier of the deleted object. JCB$W_OBJECT_FREE then points to the point table entry just freed.

Different portions of KER$DELETE handle the object-specific aspects of deletion, but, in every case, the internal subroutine KER$FREE_OBJECT (in module ALLOCATE) is called to remove the object's address from the pointer table.

Figure 10–11 shows the portions of KER$DELETE relevant to deleting an event or semaphore object. The same code sequence applies to the deletion of all kernel objects — only the actions specific to each type of object, such as unblocking processes waiting on deleted events or semaphores, differ. The deletion sequence executes as follows:

1. The object identifier is copied from the argument list.

2. If the value passed as an identifier is not negative (system bit set), or if the type bit is clear, then the identifier is actually the address of a port object. Control branches to the sequence for port deletion (described in Section 10.2.4).
3. Control branches to the KER$TRANSLATE_OBJECT internal subroutine (described in Section 10.1.3) to return the address of the identified object. If the identifier was invalid, control branches to return the failure status KER$_BAD_VALUE to the caller.
4. Control branches to the local label DELETE_OBJECT, where the object's type field is examined. For a semaphore or event object, control then branches to the local label DELETE_EVENT/DELETE_SEMAPHORE.
5. The object identifier is again copied from the argument list.
6. The kernel macro FREE OBJECT generates a branch to the routine KER$FREE_OBJECT to remove the object's address from the appropriate object pointer tables.
7. Control branches to the internal subroutine FLUSH_QUEUE, which unblocks any process waiting for the event or semaphore being deleted.
8. The kernel macro FREE POOL generates a branch to a subroutine to return the object's pool block to the system pool.
9. Success status is returned to the caller of the KER$DELETE procedure.

The internal subroutine KER$FREE_OBJECT (invoked by the FREE kernel macro) is responsible for removing the object address from the object pointer table. The routine expects two inputs: the identifier of the object and the address of the JCB for the job that owns the object. KER$FREE_OBJECT executes as follows:

1. The base field of the identifier is extracted.
2. The index field of the identifier is extracted.
3. The sequence field of the identifier is extracted.
4. The base value is used to obtain the address of the appropriate object pointer table in the base table.

Figure 10–11: Deleting an Object with KER$DELETE

```
KER$DELETE_S::
        .
        .
        .
        MOVL    OBJECT_VALUE(AP),R0      ; get object ID
        BGEQ    30$                      ; if not negative, it's a port
        BBC     #JID$V_TYPE,R0,30$       ; if type bit clear, it's a port

        BSBW    KER$TRANSLATE_OBJECT     ; translate object id to pointer
        BLBC    R0,50$                   ; branch on failure
        BRW     DELETE_OBJECT            ; branch to sequence for object deletion
        .
        .
        .
DELETE_OBJECT:
        ; test object type and branch accordingly
        .
        .
        .
DELETE_EVENT:
DELETE_SEMAPHORE:
        MOVL    OBJECT_VALUE(AP),R0      ; get object id
        FREE    OBJECT                   ; free object pointer table entry
                bsbw KER$FREE_OBJECT
        BSBW    FLUSH_QUEUE              ; flush wait queue, i.e.,
                                         ;   unblock waiting processes
        MOVL    R6,R0                    ; set address of object
        FREE    POOL                     ; free the pool block
                bsbw KER$FREE_POOL
        RETURN_STATUS SUCCESS            ; return success status
                movzbl  #KER$_SUCCESS,R0
                REI
```

5. The pointer table address and the pointer table index are used to locate the entry for the object being deleted. The value of JCB$W_OBJECT_FREE is inserted into the pointer table entry, overwriting the address of the object being deleted. The entry now contains the low word of a prototype identifier value instead of the object's address.
6. The low word of the identifier of the deleted object is increased by 1 and written to JCB$W_OBJECT_FREE. This means that JCB$W_OBJECT_FREE now points to the pointer table entry just freed. When the next kernel object is created (assuming that no further deletions intervene), its address will appear in the pointer table entry of the object just deleted.

7. The sequence value from the deleted object's identifier is increased by 1 and inserted into the sequence field in the prototype identifier, now in the freed table entry.
8. Control is returned to KER$DELETE.

The interaction between the prototype object identifiers and the value of JCB$W_OBJECT_FREE — that is, their exchange of values on object creation and deletion — ensures that the kernel will reuse the pointer table entries of deleted objects before it uses fresh entries, which may require the allocation of another pointer table. This technique can save the allocation of pool blocks in certain situations. For example, a job that creates more than 32 objects but intersperses object creations and deletions may not require more than the single pointer table allocated at job creation. The technique also eliminates the need for the kernel to search the pointer tables for a free entry — JCB$W_OBJECT_FREE always points to the next free entry. When its value is 0, the current table is full and a new one must be allocated.

Deleting an object has no effect on the user-supplied variable that holds the object's identifier. After the deletion, however, the identifier stored in the object variable is no longer valid; attempting to use that identifier in an object operation will end in failure with the KER$_BAD_VALUE status. The identifier is rendered invalid because the object pointer table entry it translates to now holds a prototype identifier value, not the address of the object.

If the pointer table entry is later reused will also return KER$_BAD_VALUE, because the sequence number in the old identifier and the object whose address now occupies the table entry will not match.

10.1.4.2 Deleting Object Structures at Job Exit

When a job exits, the kernel deallocates the resources the job had acquired during its creation and execution. These resources include the page occupied by the base table and the set of pool blocks occupied by the job's objects and object pointer tables.

A job exits when its master process exits or is deleted (that is, process exit leads to deletion). Master process deletion takes a special path through KER$DELETE, in which all the resources occupied by objects and object-management structures belonging to the job are returned to the system. This process, in KER$DELETE, occurs as follows:

1. The procedure walks the base table from the last pointer table address to the first, processing each pointer table as its address is

accessed. Since the base table is filled in consecutively from the end, the traversal continues until a zero longword or the end of the table is encountered.

2. Each pointer table is traversed from beginning to end.
3. If a pointer table entry points to an object (that is, it contains an address, not a prototype identifier), the identifier for that object is fabricated and used in a recursive call to KER$DELETE to delete the object and free its pool block.
4. When the entire pointer table has been traversed and its associated objects freed, the pool block it occupies is returned to the system pool.
5. When the base table has been traversed and its associated pointer tables have been deleted, the system page it occupies is returned to the communication region.

Ultimately, the job's JCB is returned to the system pool, rendering the values of JCB$A_OBJECT_BASE and JCB$W_OBJECT_FREE invalid.

10.2 Creating, Managing, and Deleting Port Objects

Port objects act as holding areas for messages within a VAXELN system. The maximum number of ports objects that can exist at any given moment is determined by the **Ports** entry on the System Characteristics menu. By default, a system has a limit of 256 ports. A system must have at least two ports and can have up to a maximum of 32,767.

Each port object is accessed through a 128-bit (octaword) identifier unique within a DECnet network. Within an individual system, the port object identifier is used by the kernel to look up the address of the port object within a table of port object addresses. Within a network, node address information is used by the network service to route a message to the correct node on the network.

As with job-specific objects, the addresses of port objects are stored in a table in system space. Unlike other object address tables, the port address table is created at system initialization and has only one level. Ports are created with the KER$CREATE_PORT procedure. When a job creates a port, the kernel takes the following steps on the caller's behalf:

1. It obtains a pool block from the system pool to contain the port object.
2. It formulates a unique port identifier that represents an unused entry in the port address table.
3. It initializes the port object with appropriate state information.
4. It inserts the port object into the calling job's list of ports.
5. It inserts the address of the port object into the table entry represented by the port identifier.
6. It returns the identifier to the calling job in an octaword variable supplied by the caller.

When a job subsequently passes the port identifier variable in a kernel procedure call (such as KER$SEND or KER$CONNECT_CIRCUIT), the kernel uses the value to look up the address of the port object in the port address table. Using this address, the kernel can then perform the requested operation on the port, such as placing a message in its message queue. When a job deletes a port object, the kernel takes the following steps:

1. It uses the port identifier value to locate the port table entry containing the address of the port object to be deleted.
2. It removes the port object's address from the port address table.
3. It removes the port from the calling job's list of ports.
4. It returns the pool block that the port object had occupied to the system pool.

The sections that follow describe the data structures involved in the processing of port objects and discuss creating, using, and deleting port objects in more detail.

10.2.1 Structures for Managing Port Objects

The kernel maintains a number of data structures and data items to support the creation, use, and deletion of port objects:

- The values for the systemwide data items KERGW_PORT_SIZE, KERGA_PORT_BASE, and KER$GW_PORT_FREE. These values, stored in the kernel parameter and data blocks, are used to describe the size of the port address table, the address of the table, and the next free entry in the table, respectively.

- The port address table, which contains the addresses of the existing ports within the system.
- The port identifier, a value that, when translated, locates an entry in the port address table containing the address of a port object. The identifier also describes the network address of the node on which the port resides.
- The port object itself, which contains information that identifies the object as a port and defines its state.
- The job port list, which is maintained by each job to keep track of the ports it has created.

The following sections describe these data items and structures and how the kernel uses them.

10.2.1.1 Systemwide Data Items

The 16-bit value KER$GW_PORT_SIZE in the kernel parameter block contains the user-specified value representing the maximum number of ports that can exist simultaneously within the system. The value determines the size, in longwords, of the port address table. The value and the size of the port table are fixed for the life of the system.

The longword value KER$GA_PORT_BASE in the kernel data block contains the address of the port address table. This base address is established during system initialization when the address table is allocated from system memory. The value of KER$GA_PORT_BASE is in fact 4 less than the actual base of the table; this extra longword allows the table to be accessed as an array of longwords using a base index of 1 instead of 0.

The 16-bit value KER$GW_PORT_FREE in the kernel data block contains the offset into the address table of the entry that will hold the address of the next port object created; the value points to the next free entry in the port address table.

10.2.1.2 Port Address Table

The port address table contains the addresses of all the port objects created within the system. The length of the table in longwords is KER$GW_PORT_SIZE plus 1; this extra longword at the base of the table allows the table to be accessed using an index based on 1. The padding longword at the beginning of the table is never accessed.

The port table is created during system initialization (in module INITIAL). To create the port table, the kernel takes the following steps:

1. It subtracts 4 from the current system virtual address to create the base address of the port table. (At this stage of initialization, virtual addresses are being assigned to kernel data structures as they are created.) The resulting address is copied to the KER$GA_PORT_BASE cell in the kernel data block.
2. It determines the number of physical pages required to hold KER$GW_PORT_SIZE longwords, plus 1 for the padding longword, and allocates that number of pages.
3. It generates prototype port object identifiers by calling the internal routine KER$FREE_PORT (in module ALLOCATE), which inserts them into the table as placeholders. When the table is initialized, KER$GW_PORT_FREE points to the last longword in the table. The function of KER$FREE_PORT is described in Section 10.2.4.

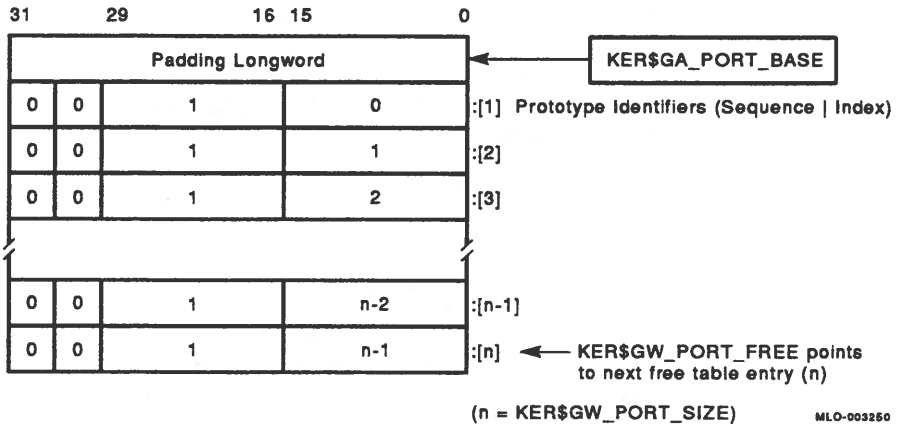
After the table is initialized, it appears as shown in Figure 10–12. In the figure, the values in the table are the prototype identifiers, which play a part in the creation of a port identifier, and the bracketed numbers represent the index values used to access the port table entry.

When port objects are created within the system, the table begins to fill with addresses from the end, because after initialization KER$GW_PORT_FREE points to the last entry in the table. As ports are created, the value of KER$GW_PORT_FREE decreases as it points to successively lower positions in the table (assuming, of course, that no ports are deleted in the meantime). When the value of KER$GW_PORT_FREE reaches 0, this signals that the table is full and that no more port objects can be created until another one is deleted.

The port object addresses in the table are accessed through VAX indexed instructions (in PC relative deferred-indexed addressing mode) using the index field in the port object identifier as the index into the port address table. For example, if R0 holds the index field of the port object identifier, the following instruction copies the address of the identified port into R1:

```
MOVL    @KER$GA_PORT_BASE[R0],R1
```

Figure 10–12: Port Address Table



The port object address obtained from this instruction — performed by the internal routine KER$TRANSLATE_PORT — is used by such port operations as KER$SEND and KER$CONNECT_CIRCUIT to manipulate ports.

10.2.1.3 Port Object Identifiers

All port-object procedures, such as KER$SEND, require their callers to supply the address of an octaword variable that contains or receives the identifier for the port. The port identifier and its address should not be confused with the port object itself, which resides in a system pool block, or with the address of the port object, which resides in the port address table.

Figure 10–13 shows the structure of a port object identifier, and Table 10–4 describes the significance of the bit fields within the 128-bit identifier.

Figure 10–13: Structure of a Port Object Identifier

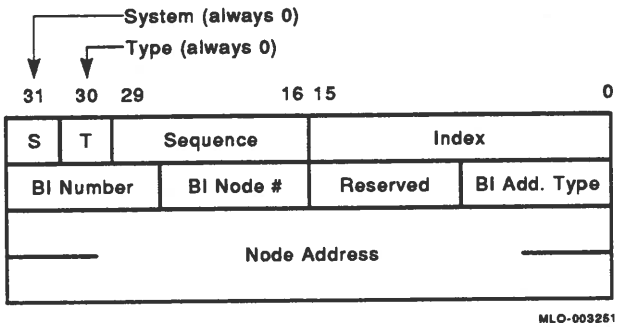


Table 10–4: Bit Fields Within the Port Object Identifier

Bit field	Size	Meaning
Index	16	The longword offset into the port address table to the address of the identified port.
Sequence	14	The generation number of the port in its port address table entry; used to detect mismatches between an identifier and a port object.
Type	1	When clear, indicates that the identified object is a port object.
System	1	Must be zero.
BI address type	8	Indicates whether the identified port is a port in a VAXBI-based closely coupled symmetric multiprocessing system.
BI node number	8	The VAXBI node number of the system on which the BI port resides. Used only on a KA800 system.
BI number	8	The VAXBI number of the system on which the BI port resides. Used only on a KA800 system.
Node address	64	The Ethernet or DECnet node address of the node on which the port resides.

The node address quadword in the identifier represents two separate values. The first longword is a constant value for all systems, $400AA_{16}$. The third word in the quadword contains the binary-coded decimal representation of a DECnet Phase IV node address; for example, the address 12.345 becomes 12633 (3159_{16}), using the following formula:

$$1024 * area + node$$

Alternatively, the same 48 bits can contain the Ethernet hardware address of the system as it was entered on the Network Node Characteristics Menu (for example, AA–00–03–00–00-0C). The final word in the quadword is reserved and must be 0.

Several of the bit fields within the identifier are represented within kernel code by symbolic values. Table 10–5 shows the names, current sizes, and significance of the symbolic values the kernel uses when manipulating port object identifiers.

Table 10–5: Assembly-Time Symbols Representing Port Identifier Bit Fields

Symbol	Size	Meaning
JID$V_SEQUENCE	16	The starting bit number of the sequence bit field.
JID$S_SEQUENCE	14	The size of the sequence bit field.
JID$M_SEQUENCE	N/A	A bit mask representing the sequence field.
JID$W_PORT_ID	N/A	The first longword in the port identifier, containing the index and sequence fields.
PID$K_LENGTH	16	The size in bytes of the port identifier.

The sequence field in the identifier plays a role in validating the port identifier in port operations by allowing the kernel to determine when port table entries are being reused. Each time a port that reuses a port table entry is created, the sequence number in the resulting identifier is increased by 1; this same value is stored within the port object itself. During the translation of the port identifier to the address of the port object, the two sequence numbers are compared. If they do not match, the identifier is invalid. This technique prevents the identifier associated with a deleted port object from being used to access the port whose address has subsequently reused the same entry in the port table.

10.2.1.4 Port Object Structure

A port object occupies one 128-btye system pool block, which is allocated from the system pool when the port is created. The port object therefore resides in system address space, where it is protected from uncontrolled access by user programs.

A port object should not be confused with its identifier, which resides in a job's address space as a 128-bit variable (a variable of type PORT in VAXELN languages). The object's address is recorded in the port address table, and its unique identifier is returned in the job's port variable.

The structure and function of a port object is described in Section B.18. For purposes of the present discussion, the 16 bytes comprising bytes 44 to 60 in the port object are of interest, because they correspond exactly to the port identifier; both contain the index, sequence, VAXBI node information, and the node address fields. During port creation in

the KER$CREATE_PORT procedure, the values for these fields, among others, are inserted into the port object. The procedure's last function before returning is to copy those 16 bytes to the caller's port variable to create the port's identifier. During subsequent port operations, the two 16-byte segments are compared to ensure that the identifier is valid for the associated port object.

10.2.1.5 Job Port Queue

Because port objects are managed across the system rather than by the job, a job maintains a doubly linked list containing the port objects that it has created. When the job creates a port, the kernel inserts the port into the job's port queue. The sole purpose of the list is to enable the kernel to locate and delete the ports that belong to a job that is being deleted.

The listhead for the queue resides in the JCB as a quadword containing the forward and backward links in the list. The forward address link appears in the JCB$A_PORT_FLINK field of the JCB; the backward link appears in the JCB$A_PORT_BLINK field. As ports are created and deleted by the job, they are inserted into and removed from the list using the INSQUE (Insert into Queue) and REMQUE (Remove from Queue) instructions.

10.2.2 Creating Port Objects

Port objects are created when a job calls the KER$CREATE_PORT procedure (in module CREATEPRT). As with job-specific kernel objects, the port creation procedure fills in the fields of the port object being created and calls a subroutine, this time KER$ALLOCATE_PORT, to generate part of the identifier. The job that calls KER$CREATE_PORT is the owner of the port. The address of the caller's JCB is recorded in a field in the port object to allow the verification of ownership in certain port operations, such as deletion.

Figure 10–14 shows the relevant segments of the KER$CREATE_PORT procedure. These code segments focus on the allocation of the port object and its identifier; the structure and function of a port object is described in Section B.18. KER$CREATE_PORT executes as follows:

1. The kernel macro ALLOCATE POOL generates a branch to a subroutine to allocate a pool block for the object.
2. The kernel macro REMOVE obtains the address of the pool block.

3. The kernel macro ALLOCATE PORT generates a branch to the subroutine KER$ALLOCATE_PORT. This routine uses the value KER$GW_PORT_FREE to generate index and sequence fields for the port identifier and returns.
4. The fields of the port object are initialized.
5. The port object is inserted into the calling job's list of ports. The listhead appears in the JCB$A_PORT_FLINK field in the job's JCB. This list allows the job to keep track of the ports it has created.
6. The index value is copied into the index field in the port object (PRT$W_INDEX).
7. The sequence value is copied into the sequence field in the port object (PRT$W_SEQUENCE).
8. The longword in the kernel data block indicating whether the system is running on a KA800 processor is copied to the port object (PRT$L_BIPORT).
9. The quadword in the kernel data block holding the node address of the system (KER$GQ_NODE_ADDRESS) is copied to the port object (PRT$Q_NODE_ADDRESS).
10. The address of the port object is inserted into the port address table using the index value as an index into the port table (KER$GA_ PORT_BASE).
11. The 16-byte identifier field in the port object, beginning with PRT$W_INDEX, is copied to the job's port variable, whose address was passed as an argument to KER$CREATE_PORT.
12. Success status is returned to the caller.

Figure 10–14: Creation of a Port Object

```
KER$CREATE_PORT_S::
        .
        . check access to job's port variable
        .
30$:    MOVL    #1,R0                   ; set number of pool blocks to allocate
        ALLOCATE POOL                   ; allocate port object block
                bsbw    KER$ALLOCATE_POOL
        BLBS    R0,40$                  ; return KER$_NO_POOL on failure
        RETURN_STATUS NO_POOL
                movzwl  #KER$_NO_POOL,R0
                rei
40$:    REMOVE  R6                      ; remove port object block from pool
                remqhi  (R1),R6
        ALLOCATE PORT                   ; allocate port table pointer
                bsbw    KER$ALLOCATE_PORT
        BLBS    R0,50$                  ; branch on success
        .
        . return KER$_NO_PORT on failure
        .
50$:    .
        . set up fields in the port object using address in R6
        .
        INSQUE  PRT$A_PORT_FLINK(R6),-  ; insert port into job's port list
                JCB$A_PORT_FLINK(R7)
        MOVW    R1,PRT$W_INDEX(R6)      ; set index of port object
        MOVW    R2,PRT$W_SEQUENCE(R6)   ; set sequence number of port object
        MOVL    W^KER$GL_PRT_BIPORT,-
                PRT$L_BIPORT(R6)        ; set VAXBI node info field
        MOVQ    W^KER$GQ_NODE_ADDRESS,- ;set node address to host node
                PRT$Q_NODE_ADDRESS(R6)  ;
        .
        .
        .
        MOVL    R6,@W^KER$GA_PORT_BASE[R1]
                                        ; store pointer to port object in table
        MOVC3   #PID$K_LENGTH,PRT$W_INDEX(R6),-
                @PORT_ADDRESS(AP)       ; store port object ID in job's port variable
        MOVL    R6,R1                   ; set address of port object
        RETURN_STATUS SUCCESS           ; return success
                movzbl  #KER$_SUCCESS,R0
                rei
```

The internal subroutine KER$ALLOCATE_PORT returns the information KER$CREATE_PORT requires to create the port identifier and store the address of the port object in the port address table. KER$ALLOCATE_PORT executes as follows to allocate a port for the KER$CREATE_PORT procedure:

1. The value of KER$GW_PORT_FREE is obtained. This value represents the index value of the next free entry in the port address table.
2. If the value of KER$GW_PORT_FREE is 0, the table is full, so control branches to return an error status from the subroutine. KER$CREATE_PORT will return the status value KER$_NO_PORT in response to the failure of KER$ALLOCATE_PORT.

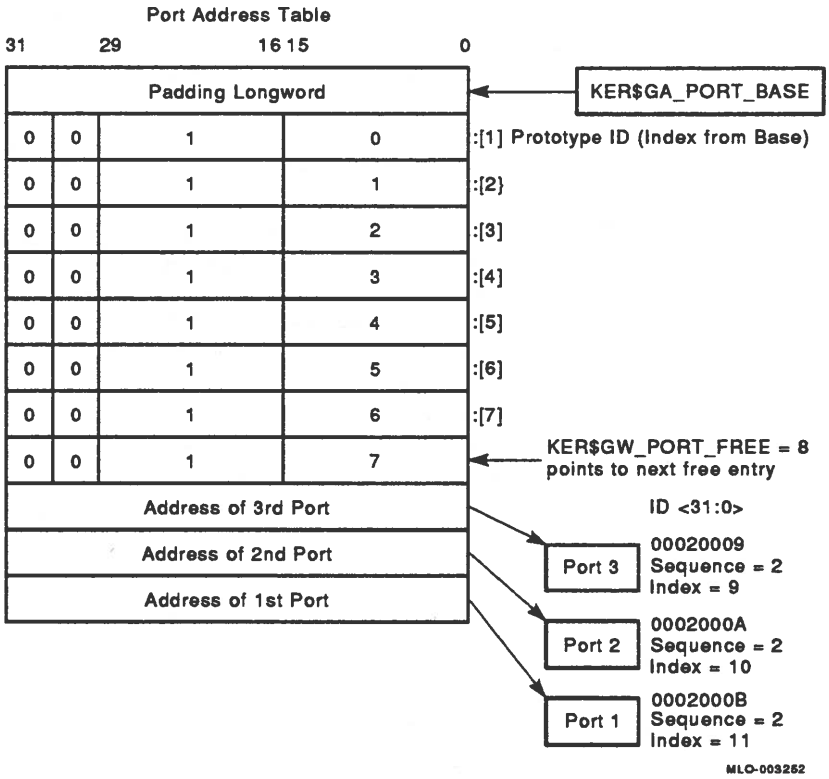
   If KER$GW_PORT_FREE is not 0, then a port can be created.
3. Using the current value of KER$GW_PORT_FREE as an index into the port table, the index field in the prototype identifier in the current table entry is copied to KER$GW_PORT_FREE. In other words, KER$GW_PORT_FREE now points to the next free table entry after the current one; this value will be used when the next port is created.
4. Again using the original value of KER$GW_PORT_FREE as an index into the port table, the sequence field in the prototype identifier in the current table entry is obtained. This value represents the sequence value for the last port, if any, whose address appeared in the current table entry. If this is the first time the entry will be used, then the sequence value will be 1, its initialized value.
5. The sequence value is increased by 1. This value will be the sequence value for the port being created. The value will appear in both the port object and its identifier and will be used to ensure a match between the two during later port operations.
6. The bit above the sequence field is cleared, just in case increasing the sequence value caused it to overflow its 14-bit limit.
7. Control returns to KER$CREATE_PORT.

KER$ALLOCATE_PORT returns the index and sequence values for the new identifier. The KER$CREATE_PORT procedure uses these values to create the first longword of the port identifier and to insert the address of the new port object into the current entry in the port address table.

Figure 10–15 shows the configuration of the port data structures after the creation of three ports in a system that can have a maximum of ten ports (KER$GW_PORT_SIZE equals 10). The value of KER$GW_PORT_FREE, as shown in the figure, is 8 and indicates that the address of the next port object created will appear in the eighth field in the table (not counting the padding longword and assuming that no existing port is deleted in the meantime). In the identifiers generated for these port objects, the sequence value has been increased to 2 from the

prototype value of 1; this sequence value also appears in the PRT$W_SEQUENCE field in the port object itself to associate the port with its identifier in subsequent port operations.

Figure 10–15: Port Address Table after the Creation of 3 Ports



10.2.3 Translating Port Object Identifiers

Port object translation — the decoding of the port identifier into the address of the port — occurs whenever the kernel must manipulate a port object. In the process of translation, the kernel validates the port identifier by ensuring that it matches the identifier portion of the port object itself.

Port objects are translated by the internal subroutine KER$TRANSLATE_PORT (in module KERNELSUB), which is called by all kernel procedures that manipulate port objects, such as KER$SEND and KER$CONNECT_CIRCUIT. The routine expects as input the address of the port identifier variable to be translated into a port address. This identifier variable will be stored in an octaword owned by the caller of the port procedure. When control returns from the subroutine, R1 will contain the address of the identified port object.

Figure 10–16 shows the brief code sequence used to validate and translate a port identifier. KER$TRANSLATE_PORT was designed to complete port translation — a common run-time operation — in as few instructions as possible. The data items and structures associated with port management exist to support that goal. The routine executes as follows:

1. The node address in the port object identifier is compared to the system's node address, stored in the global quadword KER$GQ_NODE_ADDRESS. If the node addresses do not match, then the port being translated is a port on another node and cannot be translated on this node.
2. If the identifier points to a port on a remote node, the status KER$_REMOTE_PORT is returned to the calling port procedure. In some cases, such as KER$RECEIVE, the procedure will return the status value KER$_NO_SUCH_PORT to its caller to indicate that the requested operation cannot be carried out on a remote port. In other case, such as KER$SEND, the KER$_REMOTE_PORT status indicates that the network service must become involved in the requested operation.
3. The index value (a word) is obtained from the first longword of the identifier (JID$W_PORT_ID). If that value is 0, the routine returns with a failure status, because valid index values start at 1.
4. The port table index value from the identifier is compared with the maximum number of entries in the port address table, KER$GW_PORT_SIZE. If the index exceeds the size of the table, the routine returns with a failure status.

5. The index value is used to obtain the address of the identified port object from the port address table. If the value obtained is that of the prototype identifier (bit 31 clear), the port object has been deleted, and the routine returns a failure status.
6. The first longword of the caller's port identifier is compared with the 16-bit identifier value in the port object itself to validate the identifier.
7. If the identifiers match, the identifier is valid, and the routine returns with the address of the port object in R1. If the two identifiers do not match — probably because they have different sequence values — the identifier is invalid, and the routine returns with a failure status. In such cases, the mismatch indicates that the identifier was associated with a port object that was previously deleted; the port table entry that the deleted port's address once occupied now holds the address of another port.

Figure 10–16: Port Object Translation with KER$TRANSLATE_PORT

```
KER$TRANSLATE_PORT::
        CMPL    JID$Q_NODE_ADDRESS(R0),-
                W^KER$GQ_NODE_ADDRESS   ; test 1st longword of node address
                                        ;   to see whether this is a local port
        BNEQ    15$                     ; if no match, not a local port
        CMPL    JID$Q_NODE_ADDRESS+4(R0),-
                W^KER$GQ_NODE_ADDRESS+4 ; test 2nd longword of node address
                                        ;   to see whether this is a local port
        BEQL    20$                     ; if no match, not a local port
15$:    MOVZWL  #KER$_REMOTE_PORT,R0    ; set remote port failure indicator
        RSB                             ; and return

20$:    MOVZWL  JID$W_PORT_ID(R0),R1    ; get table index from port ID
        BEQL    10$                     ; if it's 0, something's wrong
        CMPW    R1,W^KER$GW_PORT_SIZE   ; index within the table limit?
        BGTRU   10$                     ; if over the limit, something's wrong
        MOVL    @W^KER$GA_PORT_BASE[R1],R1
                                        ; get the address of the port object
        BGEQ    10$                     ; if 0 or greater, not a system address
        CMPL    JID$W_PORT_ID(R0),-
                PRT$W_INDEX(R1)         ; index and sequence number match
                                        ;   between port and ID?
        BNEQ    10$                     ; if they don't match, invalid ID
        MOVL    #1,R0                   ; set success indicator
        RSB                             ; and return
```

Figure 10–16 Cont'd. on next page

Figure 10–16 (Cont.): Port Object Translation with KER$TRANSLATE_PORT

```
10$:    CLRL    R0                      ; set failure indicator
        RSB                             ;
```

Figure 10–17 shows how KER$TRANSLATE_PORT is typically used by a kernel procedure that performs port operations, in this case, KER$RECEIVE. The procedure is passed the address of the caller's port identifier in its argument list. It first tests whether it can read that address in the caller's memory. Next, control branches to KER$TRANSLATE_PORT. If that routine succeeds, then R1 contains the address of the port object itself. That address is then used to access the fields in the port object. The one port operation shown involves testing whether the calling job is in fact the owner of the identified port.

Figure 10–17: Use of KER$TRANSLATE_PORT by KER$RECEIVE

```
KER$RECEIVE_S:
        .
        . process other arguments in the argument list
        .
        MOVL    SOURCE_ADDRESS(AP),R0   ; get address of source port ID
        IFN_READ #PID$K_LENGTH,(R0),50$ ; if not readable, return error
        .
        .
        .
        BSBW    KER$TRANSLATE_PORT      ; translate source port ID
        BLBC    R0,30$                  ; branch to return failure status
        CMPL    R7,PRT$A_OWNER(R1)      ; is the current job the owner of port?
        .
        . complete operation using port object address in R1
        .
```

10.2.4 Deleting Port Objects

The deletion of a port object occurs when a job calls the KER$DELETE procedure, passing as an argument the address of variable that holds the identifier of the port to be deleted. In deleting a port object, the kernel returns the pool block occupied by the object to the system pool, replaces the object's address in the port address table with a prototype port identifier (returning it to the list of free table entries), and removes the port from the calling job's list of ports.

When an job exits (with the deletion of its master process) all the ports created — and thus owned — by that job must be deleted from the system as well.

The following sections describe these two levels of port object deletion.

10.2.4.1 Deleting an Individual Port Object

When the KER$DELETE procedure is called to delete a port object, the following events occur:

1. Action is taken to dissociate the port object from any unreceived messages it contains, any circuit connection it belongs to, and any processes waiting for it. For example, any processes waiting on the port are unblocked with the KER$_BAD_VALUE status value.
2. The port's address is removed from the port address table and is replaced by a prototype object identifier.
3. The port's pool block is returned to the system pool.

The process of removing the port's address from the kernel's port address table is performed by the internal subroutine KER$FREE_PORT (in module ALLOCATE). The routine expects two inputs: the index and sequence values from the first longword of the port identifier. The kernel macro FREE PORT within KER$DELETE generates a branch to KER$FREE_PORT, which executes as follows:

1. The sequence value is shifted into the upper word of the identifier to create the sequence field in the prototype identifier.
2. The value of KER$GW_PORT_FREE is inserted into the low word of the identifier to create the index field in the prototype identifier.
3. The prototype identifier is copied into the port address table to overwrite the address of the port being deleted. The entry now contains a prototype identifier value instead of the port's address.

4. The index value of the port being deleted is copied into KER$GW_PORT_FREE. This means that KER$GW_PORT_FREE now points to the port table entry just freed. When the next port object is created (assuming that no further deletions intervene), its address will appear in the port table entry of the port just deleted. KER$GW_PORT_FREE will now contain the prototype identifier of the originally deleted port.
5. Control returns to KER$DELETE.

The interaction between the prototype port identifiers and the value of KER$GW_PORT_FREE — that is, their exchange of values on port creation and deletion — ensures that the kernel will reuse the port table entries of deleted port objects before it uses fresh entries. This way, the beginning of the port address table (where the address of the last port created will appear) will be reached only when the table is actually full. The technique eliminates the need for the kernel to search the entire table for a free entry — KER$GW_PORT_FREE always points to the next free entry. When its value is 0, the table is full.

Deleting a port object has no effect on the variable that holds the port's identifier. After the deletion, however, the identifier stored in the port variable is no longer valid; attempting to use that identifier in a port operation will end in failure. The identifier is rendered invalid because the port table entry to which it translates now holds a prototype identifier value, not the address of the port.

10.2.4.2 Deleting Port Objects at Job Exit

When a job exits, the kernel deallocates the resources the job had acquired during its creation and execution. As described in Section 10.1.4.2, the kernel deletes an exiting job's jobwide kernel objects by traversing its object pointer tables and deleting each object whose address it encounters.

A job's port objects must be deleted as well, but deleting ports differs from normal object deletion because the addresses of port objects are held in a systemwide table controlled by the kernel for all jobs. Moreover, walking the port address table in search of only the ports owned by the exiting job could be a time-consuming process in a large system. Instead, the kernel requires that each job keep track of its own ports by maintaining a doubly linked list of the ports it has created. The listhead of the port list appears in the job's JCB as the quadword represented by the fields JCB$A_PORT_FLINK and JCB$A_PORT_

BLINK. When the job is deleted, the kernel can access this list and quickly locate the ports owned by the job being deleted.

As with normal objects, a job's port objects are deleted when KER$DELETE is called to delete the master process in a job. The deletion of the job's ports occurs as follows:

1. The procedure obtains the address of the first port object in the job's port queue from the JCB$A_PORT_FLINK in the JCB.
2. The address of the 16-byte copy of the port's identifier value is passed as an argument in a recursive call to KER$DELETE. This deletes the port object as described in Section 10.2.4.1. Deleting the port removes it from the job's port queuem, updating the listhead in JCB$A_PORT_FLINK for the next iteration of the deletion loop.
3. When the job's queue of ports has been exhausted, the procedure continues with the deletion of the master process.

The port address table itself exists for the life of the system. The only effect on the table is the removal of the deleted job's port addresses.

Chapter 11

Job and Process Synchronization

The kernel provides the VAXELN programmer with a set of procedures and data structures that synchronize jobs and processes around the occurrence of specified events or the availability of certain resources. The programmer first identifies explicit synchronization points within the design of the application and selects appropriate kernel objects — areas, devices, events, ports, processes, or semaphores — to represent those points of synchronization. Next, coding calls to the KER$WAIT and KER$SIGNAL procedures allows processes to respond to events or the availability of resources.

When a process waits for a kernel object, the KER$WAIT procedure associates a structure called a wait control block (WCB) with the object being waited for by inserting it at the end of its wait queue. The WCB represents the waiting process and describes the conditions under which the wait will be satisfied. If the state of the object does not require a wait (for example, an event is already signaled), the process continues immediately. Otherwise, the process is placed into the waiting state, and the scheduler is invoked to select a new process to run.

When a process signals an object, the KER$SIGNAL procedure examines the WCBs associated with the signaled object to find a process whose wait has been satisfied by the signal. If one is found, KER$SIGNAL updates the state of the signaled object, dequeues the process's WCBs, and removes the process from the waiting state.

This chapter describes the kernel's support for job and process synchronization. It first treats, in Section 11.1, the data structures involved in synchronization, including WCBs and kernel objects. Section 11.2 follows with a description of the KER$WAIT_ALL and KER$WAIT_ANY procedures themselves, and Section 11.3 describes how a wait is satisfied with the KER$SIGNAL and KER$SIGNAL_DEVICE procedures.

This last section also describes three subroutines used by the kernel to support synchronization: KER$TEST_WAIT, KER$SATISFY_WAIT, and KER$UNWAIT.

Certain terms and phrases that are used frequently throughout the chapter should be understood from the outset:

- KER$WAIT: the kernel's wait code, entered through the unique entry points KER$WAIT_ALL and KER$WAIT_ANY. This term is used to refer to the bulk of the wait logic that is common to both procedures.
- Block: to enter the waiting state.
- Unblock: to leave the waiting state. An unblocked process returns first to the ready state; from there it returns to the running state as permitted by the overall scheduling state of the system.
- Wait conditions: the set of objects a process wishes to acquire or to synchronize with.
- Wait-all wait: a waiting state entered through a call to KER$WAIT_ALL. In this state, a process waits for all of a set of conditions to be satisfied. Even though some of a process's wait conditions may be momentarily satisfied, if all them are not satisfied at the same time, the process remains in its waiting state.
- Wait-any wait: a waiting state entered through a call to KER$WAIT_ANY. In this state, a process waits for any of a set of conditions to be satisfied. As soon as any one of the wait conditions is satisfied, the process unblocks.
- Wait condition satisfied: the state of a kernel object that allows a process whose wait is completely satisfied to unblock.
- Completely satisfied wait: the state in which all of a process's wait conditions are satisfied. For wait-all waits, this means all of the wait conditions are satisfied. For wait-any waits, this means at least one of the wait conditions is satisfied.
- Potentially satisfied wait: the state in which one or more of a process's wait conditions are satisfied. This phrase normally refers to the signaling of an individual kernel object which may or may not result in a process being unblocked.
- Signal: an action that causes a kernel object to enter a state that potentially satisfied a process's wait conditions.

- Object wait queue: a list of WCBs linked to an object being waited for. A WCB is inserted at the end of the wait queue when a process waits for the object. WCBs can be removed from anywhere in the queue when the object is signaled. WCBs are removed from the object wait queue when the wait of the associated process is completely satisfied.

11.1 Data Structures for Job and Process Synchronization

The wait for a kernel object involves creating a network of data structures linked through a number of queues. This section describes these data structures:

- The wait control block (WCB). This structure, established by the KER$WAIT procedure, represents the context of a process's wait for a kernel object or a time value. Chained together, multiple WCBs represent the multiple wait conditions that a caller can specify to the KER$WAIT_ANY and KER$WAIT_ALL kernel procedures.
- Synchronization-related kernel objects. The structures — area (ARA), device (DEV), event (EVT), port (PRT), process (PCB), and semaphore (SEM) — are the entities for which a process can wait.[1]
- The kernel vectors for the KER$WAIT procedures. Located in the kernel vector block, these small procedures reexecute a wait request that has been interrupted by the delivery of an asynchronous exception to a waiting process.
- The timer queue. This queue holds the WCBs of processes that are waiting for an absolute or interval time to expire.

The following discussions of these data structures focus on their individual contributions to the task of placing a process into the waiting state. Section 11.2 then shows how KER$WAIT builds these individual components into the relationships that represents a wait.

[1] The kernel uses message objects in wait and signal operations to synchronize communication among KA800 processors in closely coupled symmetric multiprocessing systems. This use of message objects is reserved and is unsupported for user programs.

11.1.1 Wait Control Block

The wait control block (WCB) is the currency in which synchronization transactions are conducted. Each object that a process waits for is represented by a WCB, and this collection of WCBs defines the conditions whereby the wait will be satisfied.

To represent a wait condition, a WCB is involved in the following relationships:

- As the representative of an object a process is waiting for, it is linked into a list of WCBs permanently attached to the process's process control block (PCB).
- As the representative of a process in the waiting state, it is linked into a queue of processes waiting for the same object.
- As the representative of a wait's timeout value, it is linked into the kernel's timer queue.

Fields in the WCB shown in Figure 11–1 and described in Table 11–1 record the details of the structure's relationship to the process it represents, to the object being waited for, and to other WCBs owned by the waiting process.

Figure 11–1: Structure of a Wait Control Block

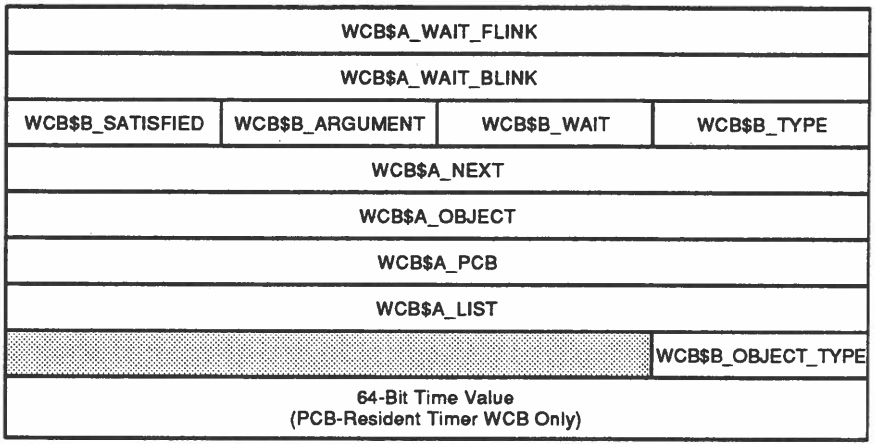


Table 11–1: WCB Fields

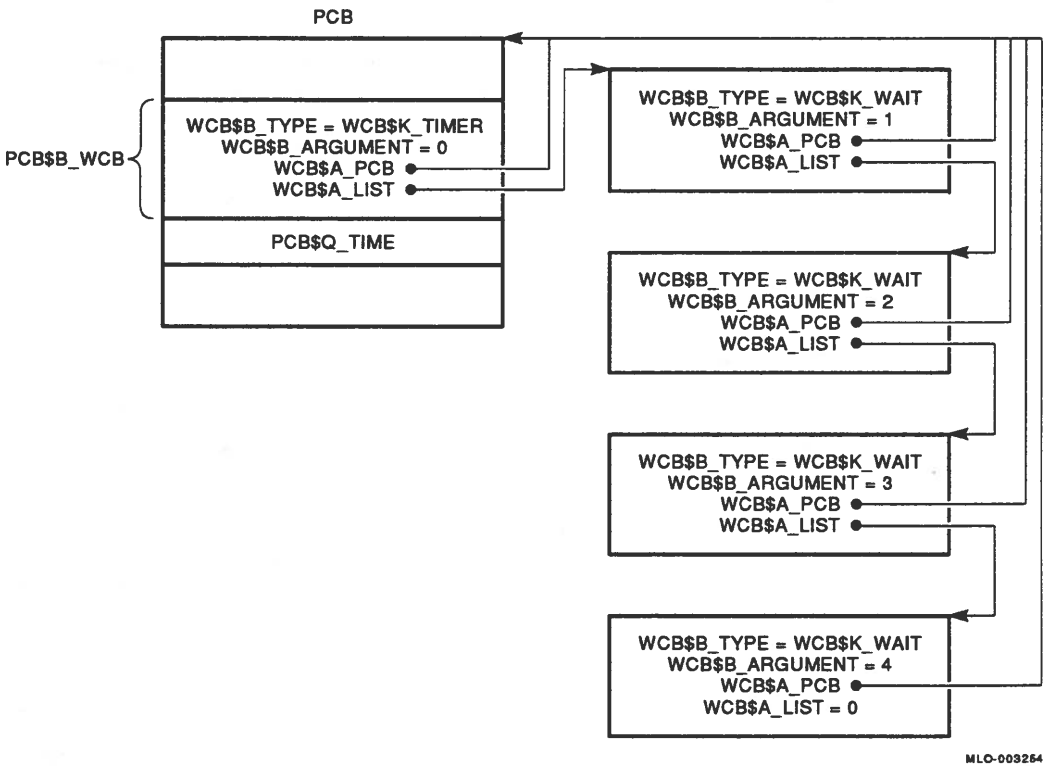
Field	Meaning
WCB$A_WAIT_FLINK WCB$A_WAIT_BLINK	The forward and backward links to the next and previous WCBs waiting for an object. These fields are used to queue the WCB to the object being waited for. The listhead for the queue resides in the wait queue links in the object data structure (see Section 11.1.3). Every process waiting for an object has one of its WCBs linked into the object's wait queue using these WCB link fields.
WCB$B_TYPE	The type of WCB. A WCB for a timed wait, called a timer WCB, is type WCB$K_TIMER; a WCB for an object wait, called an object WCB, is type WCB$K_WAIT.
WCB$B_WAIT	A bit field recording characteristics of the wait. The setting of the low bit in this byte reflects whether the WCB was created by a call to KER$WAIT_ANY (bit 0 clear) or KER$WAIT_ALL (bit 0 set). In timer WCBs, the setting of the second bit (WCB$V_WAIT_DELTA) indicates the type of timed wait requested. If the bit is clear, the wait is for an absolute time; if the bit is set, the wait is for an interval, or delta, time. This information is required for correcting interval waits after calls to KER$SET_TIME (see Section 5.3.6.1).
WCB$B_ARGUMENT	The argument in the KER$WAIT call to which the WCB corresponds. Timer WCBs are marked as argument 0. The first object argument in the call is marked as 1, and so on for each object specified. This value is returned to the caller in the optional result argument to indicate which object satisfied a wait condition.
WCB$B_SATISFIED	The state of a wait for a process object. When the wait on a process is satisfied (the process terminates), the low bit in this byte is set to show that this portion of a wait has been satisfied.

Table 11–1 (Cont.): WCB Fields

Field	Meaning
WCB$A_NEXT	A pointer to the next WCB associated with the current wait. Since all WCBs allocated for a process need not be used in every wait (for example, the wait may be for one object, or no timer value may be involved), this singly linked list stores only the addresses of the WCBs required for the current wait. The field is initialized by the KER$WAIT procedure. The list is walked by the kernel subroutines that test and satisfy wait conditions. The PCB$A_FIRST_WCB field in the associated PCB points to this list.
WCB$A_OBJECT	The address of the object being waited for.
WCB$A_PCB	The address of the PCB that owns the WCB. When a wait has been satisfied, this address is used to locate the PCB of the process to be unblocked.
WCB$A_LIST	A pointer to the next WCB in a process's list of WCBs. The singly linked list of WCBs built through this field is created during process creation, originates with the PCB-resident timer WCB, and includes every WCB allocated for a process. At process deletion, the list is walked by the KER$DELETE procedure to free the pool blocks occupied by a process's WCBs.
WCB$B_OBJECT_TYPE	The type of object for which this WCB represents a wait. This field, a copy of the type field in the object being waited for, is used in testing whether a wait has been satisfied or in satisfying a wait. These operations vary according to the object type.
WCB$Q_TIME	The time value for a timed wait. The 64-bit binary time value appears only in the timer WCB resident in the PCB. This value represents the absolute system time at which the wait expires.

When the kernel creates a process (either a master or subprocess), it creates and initializes five WCBs (including the timer WCB), as shown in Figure 11–2. The four WCBs not resident in the PCB are allocated from a single system pool block. Additional WCBs are allocated four at a time (one pool block's worth), as required, by the KER$WAIT procedure.

Figure 11–2: Relationship of WCBs to the PCB



The KER$CREATE_JOB, KER$CREATE_PROCESS, and KER$WAIT procedures initialize the following fields when creating a WCB:

- WCB$B_TYPE is set to either WCB$K_TIMER or WCB$K_WAIT. Only the first, PCB-resident WCB is marked as a timer WCB.
- WCB$B_ARGUMENT is set to the number of the wait argument to which the WCB will correspond. The timer WCB is set to 0; the next WCB is 1, the next 2, and so on, up to the limit of 250 WCBs for object wait. (Including the timer WCB, a process can have up to 251 WCBs.) When KER$WAIT sets up the WCBs for a wait, it associates each successive WCB with each successive wait argument, reserving the timer WCB for a timeout argument.

- WCB$A_PCB is set to contain the address of the PCB that owns the WCB.
- WCB$A_LIST is set to point to the next WCB in the list of the process's WCBs. In the last WCB, this field is set to zero to terminate the list.

The values set in these fields endure for the life of the process. The remaining fields in the WCB are set by the KER$WAIT procedure, as described in Section 11.2.

11.1.2 Process Control Block

The process control block (PCB) has a dual nature in wait operations. It represents both a process that wishes to wait and a process that is waited for. This section describes the aspects of the PCB that relate to its role in allowing a process to wait for other objects. Section 11.1.3.3 focuses in turn on the portions of the PCB that function in its role as an object for another process's wait.

The PCB contains several elements that associate it with the WCBs that represent the conditions of its wait. These elements are shown in Table 11–2. Figure 4–3, which illustrates the PCB, shows where these elements appear in the structure.

Table 11–2: PCB Fields to Support Process Waiting

Field	Meaning
PCB$A_FIRST_WCB	The address of the first of the WCBs that define the current wait conditions for the process. This address is set by the KER$WAIT procedure and passes the address of the start of the active WCB list to several kernel subroutines. For a timed wait, this field contains the address of the timer WCB; for waits without a time value, the address of the first object WCB is used.

Table 11–2 (Cont.): PCB Fields to Support Process Waiting

Field	Meaning
PCB$B_WCB	The PCB-resident WCB. This WCB is dedicated for use as a timer WCB; its 64-bit time value appears in its WCB$Q_TIME (PCB$Q_TIME) field. Its WCB$A_WAIT_FLINK and WCB$A_WAIT_BLINK fields link the timer WCB into the system timer queue; its WCB$B_ARGUMENT field is set to 0 to signify its role as the timer WCB.
PCB$Q_TIME	The WCB$Q_TIME field for PCB$B_WCB. The 64-bit absolute time value used for timed waits. The time values for interval waits are converted to absolute times before being stored here.

11.1.3 Kernel Objects

Some kernel objects — the event and the semaphore — exist solely to support synchronization. This section describes all aspects of their creation and function. Other objects — the area, the device, and the port — participate in synchronization operations as resources for which processes wait. This section describes only those aspects of these kernel objects that relate to synchronization. Discussions of their other aspects are deferred to other sections. The common aspects of object creation, such as the allocation of object identifiers, are described in Chapter 10.

11.1.3.1 Event Object

The event object records whether a user-defined event has occurred. If the event has not occurred, the event's state is said to be cleared. If the event has occurred, the event's state is said to be signaled. A process's wait for an event object is satisfied when the event is signaled. When this occurs, all waits for the event are potentially satisfied. Contrast this with the semaphore and area objects, for which a signal allows the wait of only one process to be satisfied.

Figure 11–3 shows the structure of an event object, and Table 11–3 describes its fields.

Figure 11–3: Structure of an Event Object

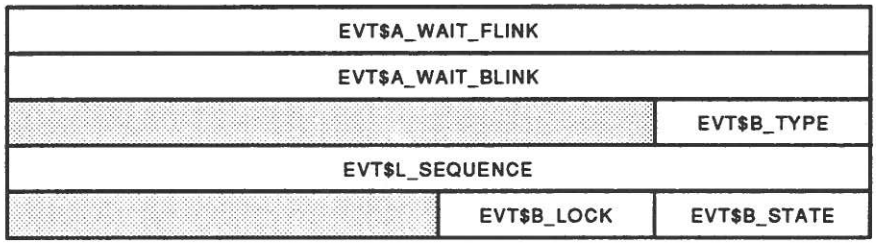


Table 11–3: Event Fields

Field	Meaning
EVT$A_WAIT_FLINK EVT$A_WAIT_BLINK	The listhead for the queue of processes waiting for this object. The WCB representing a waiting process is inserted into this queue by the KER$WAIT procedure. When the event is signaled, processes whose waits are completely satisfied are removed from the queue and unblocked.
EVT$B_TYPE	The object type: OBJ$K_EVENT.
EVT$L_SEQUENCE	The object sequence number. This value must match the sequence field in the object identifier during object translation.
EVT$B_STATE	The current state of the object, either cleared (0) or signaled (1). Signaling and clearing the event change this field's value.
EVT$B_LOCK	An interlock bit indicating that the event object is in use. This field is used exclusively by the error-logging subsystem in awakening the ERRFORMAT job. See Section 7.1.3.2.

Event objects are created by the KER$CREATE_EVENT kernel procedure (in module CREATEEVT). The procedure expects three arguments: the optional address to receive a status value, the address to receive the object's identifier, and a value representing the initial status of the event (0 for cleared, 1 for signaled). KER$CREATE_EVENT creates an event object with the following steps:

1. A pool block is allocated to hold the event object. If no pool is available, KER$_NO_POOL status is returned.
2. An object identifier is allocated for the object by calling the subroutine KER$ALLOCATE_OBJECT. The routine also returns the address of an entry in an object pointer table and the current sequence number for the object. If the job's object pointer tables are filled to capacity, the KER$_NO_OBJECT status is returned.
3. The wait queue listhead is initialized. EVT$A_WAIT_FLINK and EVT$A_WAIT_BLINK are both set to point to EVT$A_WAIT_FLINK to signify that the queue is empty.
4. EVT$B_TYPE is set to OBJ$K_EVENT.
5. The initial state is set in EVT$B_STATE as specified by the caller.

6. The interlock bit in EVT$B_LOCK is cleared.
7. The object address is stored in the allocated entry in the object pointer table.
8. The allocated sequence number is set in EVT$L_SEQUENCE.

When an event object is signaled with the KER$SIGNAL procedure, the event's state is set to signaled, and all processes queued to the event whose waits are thereby satisfied are returned to the ready state. The signaling of an event object is described in more detail in Section 11.3.1. An event object can be returned to the cleared state with the KER$CLEAR_EVENT kernel procedure, which simply clears the low bit of EVT$B_STATE.

When an event object is deleted, the waits of all processes in the object's queue are immediately unblocked with KER$_BAD_VALUE status.

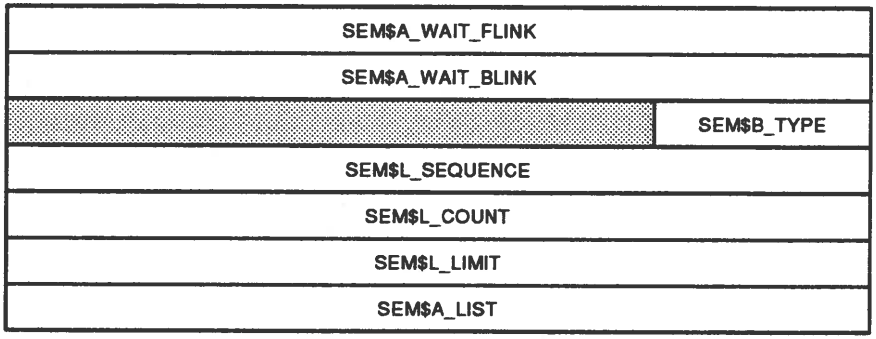
11.1.3.2 Semaphore Object

The semaphore object represents a resource to which access must be limited to one or a limited number of processes. Access to a semaphore is controlled by a pair of values: a maximum count and a current count. A semaphore with a maximum count of 1, a binary semaphore, provides exclusive access to the resource the semaphore protects. A semaphore with a count greater than 1, called a counting semaphore, allows metered access to the resource. The semaphore's current count defines the state of the semaphore; this count determines how many accesses are allowed at a given moment.

A successful wait for a semaphore causes its count to decrease by one, down to a minimum of zero. Signaling a semaphore causes its count to increase by one, up to its maximum count. A wait for a semaphore object is satisfied when the semaphore's count becomes greater than 0. Signaling a semaphore beyond its maximum count returns KER$_COUNT_OVERFLOW status and usually indicates an error in programming logic.

Figure 11–4 shows the structure of a semaphore object, and Table 11–4 describes its fields.

Figure 11–4: Structure of a Semaphore Object



MLO-003255

Table 11–4: Semaphore Fields

Field	Meaning
SEM$A_WAIT_FLINK SEM$A_WAIT_BLINK	The listhead for the queue of processes waiting for this semaphore. A WCB representing a waiting process is inserted into this queue by the KER$WAIT procedure. When the semaphore is signaled, the first process whose wait is completely satisfied is removed from the queue and unblocked.
SEM$B_TYPE	The object type: OBJ$K_SEMAPHORE.
SEM$L_SEQUENCE	The object sequence number. This value must match the sequence field in the object identifier during object translation.
SEM$L_COUNT	The number of processes that can yet access the resource protected by the semaphore. If this value is greater than zero, a process waiting on the semaphore can acquire immediate access to the protected resource. A satisfied wait for a semaphore decreases the count by one. Signaling a semaphore, in turn, increases its count by 1. A count of 0, then, means that a process will have to wait for the semaphore to be signaled before gaining access to the resource it protects.
SEM$L_LIMIT	The maximum number of processes that may have access to the semaphore. When the semaphore count equals this limit, all the units of the protected resource are available.

Semaphore objects are created by the KER$CREATE_SEMAPHORE kernel procedure (in module CREATESEM). The procedure expects four arguments: the optional address to receive a status value, the address to receive the object's identifier, a value specifying the initial count for the semaphore, and a value specifying its maximum count. KER$CREATE_SEMAPHORE creates a semaphore object with the following steps:

1. The initial value is compared to the maximum value. If the initial value is greater than the maximum, KER$_BAD_VALUE status is returned.
2. A pool block is allocated to hold the event object. If no pool is available, KER$_NO_POOL status is returned.

3. An object identifier is allocated for the object by calling the subroutine KER$ALLOCATE_OBJECT. The routine also returns the address of an entry in an object pointer table and the current sequence number for the object. If the job's object pointer tables are filled to capacity, KER$_NO_OBJECT status is returned.
4. The wait queue listed is initialized. SEM$A_WAIT_FLINK and SEM$A_WAIT_BLINK are both set to point to SEM$A_WAIT_FLINK to signify that the queue is empty.
5. SEM$B_TYPE is set to OBJ$K_SEMAPHORE.
6. The specified initial and maximum counts are set in SEM$L_INITIAL and SEM$L_MAXIMUM, respectively.
7. The object address is stored in the allocated entry in the object pointer table.
8. The allocated sequence number is set in EVT$L_SEQUENCE.

When a semaphore object is signaled with the KER$SIGNAL procedure, the value of SEM$L_COUNT is increased by one, and the first process in the queue whose wait is thereby satisfied is returned to the ready state. The signaling of a semaphore object is described in more detail in Section 11.3.1.

When a semaphore object is deleted, the waits of all processes in the object's queue are unblocked with KER$_BAD_VALUE status.

The VAXELN run-time library uses a semaphore object to implement an optimized synchronization object called the mutex (for mutual exclusion semaphore). The mutex consists of a binary semaphore and a signed-word count value initialized to –1. Locking the mutex amounts to increasing the count by one. If the new count does not equal 0, the mutex has already been locked, and the calling process is forced to wait on the mutex semaphore (that is, KER$WAIT_ANY is called by the lock routine on behalf of the locking process). Unlocking the mutex means to decrease the mutex count by 1. If the new count is not –1, the mutex semaphore is signaled (that is, KER$SIGNAL is called by the unlocking routine on behalf of the unlocking process). The mutex offers optimized performance by requiring the call to KER$WAIT_ANY and KER$SIGNAL only when ownership of the mutex is in contention.

11.1.3.3 Process Object

A process object represents an independent thread of execution within a job. A wait for a process is satisfied when the process terminates by exiting or being deleted.

When one process waits for another process, the KER$WAIT procedure inserts a WCB to represent the waiting process at the end of the wait queue of the process being waited for. The listhead for the queue of WCBs is located at the PCB$A_WAIT_FLINK and PCB$A_WAIT_BLINK fields in the PCB of the process being waited for.

When that process terminates, the process-deletion logic in KER$DELETE sets the low bit in the WCB$B_SATISFIED field in every WCB in the terminating process's wait queue. It then tests the wait conditions of every process waiting for the deleted process. If the deletion of the process satisfies the wait, the waiting process is unblocked. If the deletion of the process does not satisfy the wait (this may be the case for wait-all waits), the waiting process remains in its waiting state; however, the setting of the WCB$B_SATISFIED bit in the waiting process's WCB will signify for later tests that this portion of the wait has been satisfied.

A process can be forced to exit with the KER$SIGNAL procedure. This procedure raises the asynchronous exception KER$_QUIT_SIGNAL in the signaled process. If that process does not handle the exception, it is forced to exit, and the waits of all processes waiting for it are potentially satisfied. Process signaling is described in Section 6.5.2.1.

The creation of the process object itself is described in Section 4.5. Process deletion is described in Section 4.6.

11.1.3.4 Area Object

An area object represents a region of physical memory that can be shared among multiple jobs by being mapped into their respective P0 address spaces. Each job that maps the same area has an area object that points to a jobwide structure, the area control block (ACB), that synchronizes access to the shared memory. The structure and function of the area object and the ACB are described in Sections B.3 and B.1, respectively. This section looks briefly at the synchronization aspects of areas.

When a process waits for an area object, it is in fact waiting on a binary semaphore built into the area's ACB by the KER$CREATE_AREA procedure. This internal semaphore is defined by its current and maximum counts, maintained in the ACB's ACB$L_COUNT and ACB$L_LIMIT fields. When the ACB is created, these fields are initialized to 1, meaning that at most one process may access the area and that the area is initially available. As with semaphore objects, gaining access to the area causes the current count to decrease by 1. Signaling the area with KER$SIGNAL causes the count to increase by 1.

The ACB, not the area object, also contains the listhead (ACB$A_WAIT_FLINK and ACB$A_WAIT_BLINK) for the queue of WCBs that represent the processes (most likely in separate jobs) waiting for the area. When a process waits for the area object, a WCB for the process is inserted in the associated ACB's wait queue. When the area is signaled, the first process in the queue whose wait is thereby satisfied is returned to the ready state.

The run-time library also supplies a mutex, called the area-lock variable, that optimizes access to areas. The lock mutex consists of the ACB's internal semaphore and a signed-word count value. The function of a mutex is described in Section 11.1.3.2.

11.1.3.5 Port Object

A port object represents a storage area for unreceived messages. A wait on a port object is satisfied when a message is inserted into the port's message queue. The structure and function of the port object are described in Section B.18. This section looks briefly at the synchronization aspects of ports.

When a process waits for a port, a WCB that represents the waiting process is inserted into the wait queue at the listhead defined by PRT$A_WAIT_FLINK and PRT$A_WAIT_BLINK in the port object. The waits of all processes waiting for the port are potentially satisfied when the port object changes state in one of the following ways:

- A message arrives in the port. All processes whose waits are completely satisfied by the arrival of the message are unblocked.
- The port's partner in a circuit disconnects its part of the circuit. All processes whose waits are completely satisfied by the disconnection of the circuit are unblocked.
- The port is deleted. In this case, all processes waiting for the port are immediately unblocked with KER$_BAD_VALUE status.

A process may enter a waiting state for a port by explicitly calling the KER$WAIT procedure, or the kernel may implicitly put it into a wait for a port after a call to KER$ACCEPT_CIRCUIT (waiting for a circuit connection), KER$CONNECT_CIRCUIT (waiting for a circuit connection to be accepted), or KER$SEND (waiting for a port to become nonfull).

11.1.3.6 Device Object

A device object represents a channel to a hardware device connected to the target computer. A wait on a device object is satisfied when the device object is signaled with the KER$SIGNAL_DEVICE kernel procedure. A device is signaled from an interrupt service routine associated with the device to indicate that an interrupt has been serviced. The structure and function of the device object is described in Section B.5. This section looks briefly at the synchronization aspects of devices.

Waiting for a device to be signaled allows a device driver process to synchronize its execution with the device and its interrupt service routine. When a process waits for a device, a WCB that represents the waiting process is inserted into the wait queue at the listhead defined by DEV$A_WAIT_FLINK and DEV$A_WAIT_BLINK in the device object.

When a device is signaled with KER$SIGNAL_DEVICE, the device object is inserted into the global device queue, located at KER$GQ_DEVICE_LIST, using the DEV$A_FORK_FLINK and DEV$A_FORK_BLINK fields to form the link with the queue. An IPL 8 software interrupt is then requested to service the queue. The function of KER$SIGNAL_DEVICE is described fully in Section 11.3.2.

When it executes, the IPL 8 interrupt service routine, KER$DEVICE_SIGNAL in module SIGNALDEV, removes the device object from the system device queue and sets the low bit in DEV$B_STATE to show that the device has been signaled. Finally, it scans the list of WCBs queued to the device and unblocks the first waiting process whose wait conditions are completely satisfied by the device signal. KER$DEVICE_SIGNAL is also described in Section 11.3.2.

11.1.4 KER$WAIT Kernel Vectors

Kernel vectors are brief procedures that represent the global entry point for public and internal kernel procedures. These vectors, discussed fully in Chapter 8, transfer control to the procedure's internal entry point, check the procedure's completion status, and return control to the caller.

The kernel vectors for KER$WAIT_ALL and KER$WAIT_ANY perform a special role in allowing asynchronous exceptions to be delivered to waiting processes or to processes about to enter the waiting state. These vectors execute a special test immediately after control returns to the vector from the KER$WAIT procedure. Figure 11–5 shows the kernel vector for KER$WAIT_ANY, which tests the value of R0.

Figure 11–5: Kernel Vector for KER$WAIT_ANY

```
KER$WAIT_ANY::                      ; define kernel service entry
        .WORD   KER$WAIT_ANY_M      ; KER$WAIT_ANY entry mask
30086$: CHMK    #CHMK_CODE          ; CHMK to procedure, return via REI
        TSTL    R0                  ; wait condition satisfied?
        BEQL    30086$              ; if eql, no - reexecute the wait
        TSTL    8(AP)               ; did caller request the wait result?
        BEQL    30087$              ; if eql, no - so don't return one
        MOVL    R1,@8(AP)           ; return the wait result
30087$:
        BRW     KER$RETURN_STATUS
                                    ; return kernel service status
```

If the procedure succeeded, the value of R0 will be 1; if the procedure failed, the value will be greater than 1. In both cases, execution will proceed through the vector to return the status to the calling process. A value of 0 in R0, however, indicates that the process's wait was interrupted by the delivery of an asynchronous exception. The Change Mode to Kernel (CHMK) instruction is therefore reexecuted to allow the process to return to the waiting state it was in before the delivery of the asynchronous exception, if the conditions of the wait were not satisfied in the meantime.

The 0 value is placed into R0 either by KER$WAIT itself, when it discovers that an asynchronous exception is pending against the calling process, or by the internal subroutine KER$UNWAIT, when it unblocks the waiting process to allow the asynchronous exception to be delivered. The role of the KER$WAIT procedure in clearing R0 is described in Section 11.2, and the role of KER$UNWAIT is described in Section 11.3.3.3. Section 6.5 deals with the asynchronous exception mechanism itself.

11.1.5 Timer Queue

When a process requests a timed wait by supplying a time value to the KER$WAIT procedure, a timer WCB representing the process is inserted into a global list called the timer queue, whose listhead is located at KER$GQ_TIME_QUEUE. The PCB-resident WCB, PCB$B_WCB, and its associated 64-bit time value, PCB$Q_TIME, are used for this purpose.

The WCBs in the timer queue are ordered according to the time values they contain — the WCB with the shortest wait appears first in the queue, the longest appears last. The ordering is based on absolute time values.

If the timed wait was requested as a interval (relative or delta) time, KER$WAIT converts it to an absolute time, by adding the specified interval time to the current system time, and writes it to the WCB$Q_TIME field in the timer WCB. The second bit (WCB$V_WAIT_DELTA) in WCB$B_WAIT is also set; this allows the time values of relative-wait entries to be adjusted if the system time is changed.

If the wait was requested as an absolute time, the specified time value is simply copied into WCB$Q_TIME. The timer WCB is then inserted at the appropriate point in the timer queue, and the calling process is placed into the waiting state.

Whenever the interval clock interrupt service routine finds that the first entry in the timer queue has come due (its time value is less than or equal to the system time), a software interrupt is requested to activate the software timer ISR. This routine walks the queue of WCBs in the timer queue and unblocks every process whose timer has expired.

The role of KER$WAIT in establishing timed waits is dealt with in Section 11.2. Other time-related kernel functions are described in detail in Section 5.3.

11.2 KER$WAIT Procedures

Two kernel procedures, KER$WAIT_ALL and KER$WAIT_ANY, enable a process to synchronize its execution with one or more events or resources represented by VAXELN kernel objects. Calling the KER$WAIT_ANY procedure specifies that the calling process should unblock when any one of its wait conditions is satisfied. Calling the KER$WAIT_ALL procedure specifies that the calling process should unblock when all of its wait conditions are satisfied. (When a wait condition is satisfied depends on the object being waited for: waits on area, device, event, process, and semaphore objects are satisfied when the object is signaled; waits on port objects are satisfied when a message arrives in the port. A wait on a process is also satisfied when the process terminates.)

The two procedures are separate entry points to the WAIT module, where they share the majority of their code. On entry into one of the procedures, a flag is set to indicate which procedure was called, and control then branches to common code.

The goal of the KER$WAIT procedures is to establish a set of WCBs to represent the conditions of the calling process's wait, test the conditions of the wait, and, if the wait is not completely satisfied, insert the WCBs into kernel object wait queues and place the calling process into the waiting state. If the procedure's test finds that the conditions of the wait are already satisfied, the WCBs are not inserted into the wait queues, and the calling process is allowed to continue execution without blocking.

This section describes the operation of the KER$WAIT procedures by dividing it into the following functional stages:

1. The procedure is entered, the wait-all/wait-any flag is set, and control is transferred to common code. See Section 11.2.1.
2. The WCBs for the wait are established. See Section 11.2.2.
3. If the caller specified a timeout value, the timer WCB is established. See Section 11.2.3.
4. The address of the first WCB in the wait list is saved in the PCB. See Section 11.2.4.
5. The WCBs are used to test the wait conditions. See Section 11.2.5.
6. If the wait conditions are not completely satisfied, a test for a pending asynchronous exception is performed. If one is present, entry to the waiting state is postponed. See Section 11.2.6.

7. The WCBs are inserted into the appropriate wait queues. See Section 11.2.7.
8. The scheduler is called to remove the process from execution and select a new process to run. Section 11.2.8.

The KER$WAIT procedure executes in kernel mode and expects at least three arguments: the address of an optional status variable, the address of an optional result variable, and the address of an optional timeout value for the wait. These arguments can be followed by the object identifier values for up to 250 kernel objects (for port objects, the address rather than the value of the identifier is passed). If the result argument is specified, the kernel will return to the caller the argument number of the object that first satisfied the wait.

11.2.1 Step 1 — Enter the Procedure

If KER$WAIT is entered through the KER$WAIT_ALL entry point, the lower bit in a general register is set as a wait-all flag. When WCBs are established for the wait, this flag is copied into the WCB$B_WAIT field to record the nature of the wait for later tests. If the procedure is entered through the KER$WAIT_ANY entry point, the wait flag is cleared.

Once the flag is set, control branches to common code within the WAIT module. Here, the procedure first obtains the address of the PCB-resident timer WCB, PCB$B_WCB. It next determines whether any objects have been specified in the call by subtracting the minimum number of arguments for the call (3) from the number of arguments actually specified in the call (pointed to by the argument pointer). If the result is 0, no objects were specified, and execution branches to step 3 to determine whether a timeout value was specified. If objects were specified, the process of establishing WCBs begins.

11.2.2 Step 2 — Establish WCBs for the Wait

If the caller specified kernel objects to be waited for, KER$WAIT must establish a WCB to test the object's condition and to possibly be inserted into each object's wait queue. Establishing a WCB means setting its dynamic fields, those that vary from wait to wait or can change during a wait: WCBB_WAIT, WCBB_SATISFIED, WCBA_NEXT, WCBA_OBJECT and WCB$A_OBJECT_TYPE. If the number

of specified objects exceeds the number of WCBs allocated to the process, a local subroutine is called to allocate additional WCBs from a system pool block, four at a time, as needed.

The following steps execute in a loop until all the objects specified in the call have been processed into corresponding WCBs:

1. The object identifier is obtained from the argument list. If the system or type bits in the identifier are set, the argument is an identifier for a jobwide kernel object. If either bit is clear, then the object argument is assumed to be the address of an identifier for a port object.

   Jobwide kernel objects are treated as follows:

   a. The kernel subroutine KER$TRANSLATE_OBJECT is called to return the address of the kernel object. If the translation fails, KER$_BAD_VALUE is returned to the caller.

   b. Execution continues, using the address of the object.

   Port identifiers are treated as follows:

   a. The LOCK macro is executed to ensure exclusive access to the port address table.

   b. The kernel subroutine KER$TRANSLATE_PORT is called to return the address of the port object. If the port is not found, KER$_BAD_VALUE is returned to the caller.

   c. The address of the caller's JCB is compared to the value of PRT$A_OWNER, the address of the JCB of the job that created the port. In most cases, if the two values do not match, KER$_BAD_VALUE is returned. Normally, a process is allowed to wait only for ports it created; however, if the caller was in kernel mode before calling KER$WAIT, and if the port is not connected in a circuit, the process may wait on a port not owned by its job.

   d. Execution continues, using the address of the port object.

2. The validity of the object for wait operations is tested. The value in the OBJ$B_TYPE field of the object is compared to the setting in a bit mask located at KER$GL_WAIT_OBJECT within the WAIT module. The mask contains a set bit for each object that can be waited for. If the current object's bit is not set in the mask, KER$_BAD_TYPE is returned to the caller.

3. The WCB$A_NEXT field of the current WCB is initialized by copying the value of WCB$A_LIST into it. The first time through the loop, the current WCB is the PCB-resident timer WCB. The initialization of this WCB is not completed unless the caller specified a timeout argument.

   If the value of WCB$A_LIST is 0 at this point, the last WCB currently allocated to the process has been reached. Before continuing, KER$WAIT calls the local subroutine KER$EXPAND_PROCESS_WAIT to allocate four additional WCBs and link them into the process's WCB list through their WCB$A_LIST fields. If the subroutine succeeds, this step is reexecuted. If the subroutine fails, KER$_NO_POOL is returned to the caller.
4. The address of the next WCB is obtained from the WCB$A_LIST field of the current WCB. The first time through the loop, this is the address of the first object WCB in the list.
5. The wait flag stored in a general register is written to the low bit of WCB$B_WAIT to establish whether the WCB was established by a call to KER$WAIT_ALL or KER$WAIT_ANY.
6. The low bit of WCB$B_SATISFIED is cleared. If this WCB represents a wait for a process object, the setting of this bit indicates whether this wait has been satisfied in a wait-all wait.
7. The address of the object, returned by the translation subroutines, is written to WCB$A_OBJECT. This address is used later to queue the WCB to the object.
8. The type field in the object is copied to the WCB$B_OBJECT_TYPE field of the WCB. This field will be used later in testing and satisfying the wait for this object.
9. The loop continues by returning to the first step until all the specified objects have been processed.

11.2.3 Step 3 — Establish the Timer WCB

Once the object WCBs have been established, KER$WAIT checks to see whether a timeout argument was specified. If none was specified, and no objects were specified, KER$_SUCCESS and a wait result of 0 are returned to the caller. If no timeout was specified but at least one object was, execution branches to the next step.

If a timeout value was specified, the timer WCB must be initialized. First, the time value is tested. If it is an absolute time (0 or greater), its value is copied to the WCB$Q_TIME (PCB$Q_TIME) field of the timer WCB.

If the timeout value is specified as an interval time (a negative value), it is transformed to the absolute system time of the timeout by negating its value and adding it to the value of KER$GQ_SYSTEM_TIME, the current system time. Then the second bit in WCB$B_WAIT (WCB$V_WAIT_DELTA) is set to show that the timeout was specified as a relative (delta) time. This allows the KER$SET_TIME procedure to adjust its time value if the system time is reset, preserving the relative nature of its wait (see Section 5.3.6.1).

11.2.4 Step 4 — Save the Address of the First WCB

Once all the WCBs for the wait have been established, KER$WAIT saves the address of the first WCB in the list by writing it to the PCB$A_FIRST_WCB field in the waiting process's WCB. If the wait has a timeout, this address is that of the timer WCB; otherwise, it is the address of the first object WCB in the list. The value of PCB$A_FIRST_WCB can be used by other kernel routines to locate the process's established WCBs.

The WCBs for this wait are now connected in a circular list by their WCB$A_NEXT fields. If there is only one WCB in the wait, either the timer WCB or the first object WCB, this field points to that WCB. If there are multiple WCBs, the last WCB completes the list by pointing back to the first WCB (either the timer WCB or the first object WCB), whose address is saved in PCB$A_FIRST_WCB.

11.2.5 Step 5 — Test the Wait Conditions

The process that called KER$WAIT is placed into the waiting state only if its wait is not already completely satisfied at the time of the call. If the caller specified a timeout value of 0, then the wait is is immediately completely satisfied (nonblocking), regardless of the wait conditions of any objects specified. (Specifying a timeout value of 0 acts a "test wait" by allowing a process to determine whether its wait conditions are satisfied at the time of the call to KER$WAIT without having to risk entering the waiting state.)

If the caller specified only a (nonzero) timeout value for the wait, no objects must be tested. Instead, in later steps, the timer WCB will be inserted into the timer queue, and the process will enter the waiting state.

If the process is waiting for at least one object, KER$WAIT calls the internal subroutine KER$TEST_WAIT (in module KERNELSUB). This subroutine, described in Section 11.3.3.1, walks the list of WCBs, tests the objects they represent, and, if the wait is completely satisfied, returns the argument number of the object that satisfied the wait.

If the specified wait is completely satisfied, the KER$SATISFY_WAIT procedure is called (see Section 11.3.3.2), and the argument number and KER$_SUCCESS are returned to the caller. If the wait is not yet satisfied, execution continues with the next step, unless the caller also specified a timeout value of 0. In that case, the "test" wait is immediately satisfied, and the argument number 0 (timeout) and KER$_SUCCESS are returned to the caller.

11.2.6 Step 6 — Test for a Pending Asynchronous Exception

The kernel will not place a process into the waiting state if an asynchronous exception is pending against the process and the process is capable of accepting the asynchronous exception. If these conditions are met, the process's wait is deferred until the asynchronous exception has been delivered. If the process handles the exception, it reenters the KER$WAIT code and possibly enters the waiting state. The use and delivery of asynchronous exceptions are described in Section 6.5.

KER$WAIT tests whether an asynchronous exception is pending by examining the PCB$B_REASON field of the current PCB. If none of the asynchronous exception bits is set, execution continues with the next step to place the process into the waiting state.

If the test reveals that an asynchronous exception is pending against the process, KER$WAIT next determines whether the process is capable of accepting the exception by testing the following conditions; if any test fails, execution continues with the next step:

- The delivery of asynchronous exceptions must be enabled for the process. Namely, the PCB$V_SIGNAL_DISABLE bit in PCB$B_REASON must be clear.

- The IPL of the process prior to the call to KER$WAIT must be 0. If the previous IPL was not 0, then KER$WAIT was called implicitly by the kernel on behalf of a process; that is, the process is entering an implicit waiting state. Therefore, the kernel defers the delivery of the asynchronous exception until the kernel procedure requesting the implicit wait has completed.
- The hardware must be able to deliver the asynchronous exception. Namely, the access mode of the process before the call to KER$WAIT must not be more privileged than the process's base access mode (JCB$B_MODE). The REI instruction that exits the KER$WAIT code also performs a comparable test to determine whether to request an IPL 2 software interrupt, which initiates the delivery of the asynchronous exception (see Section 6.5.1.1). Therefore, if the asynchronous exception cannot be delivered by the hardware, the process is allowed to enter the waiting state.

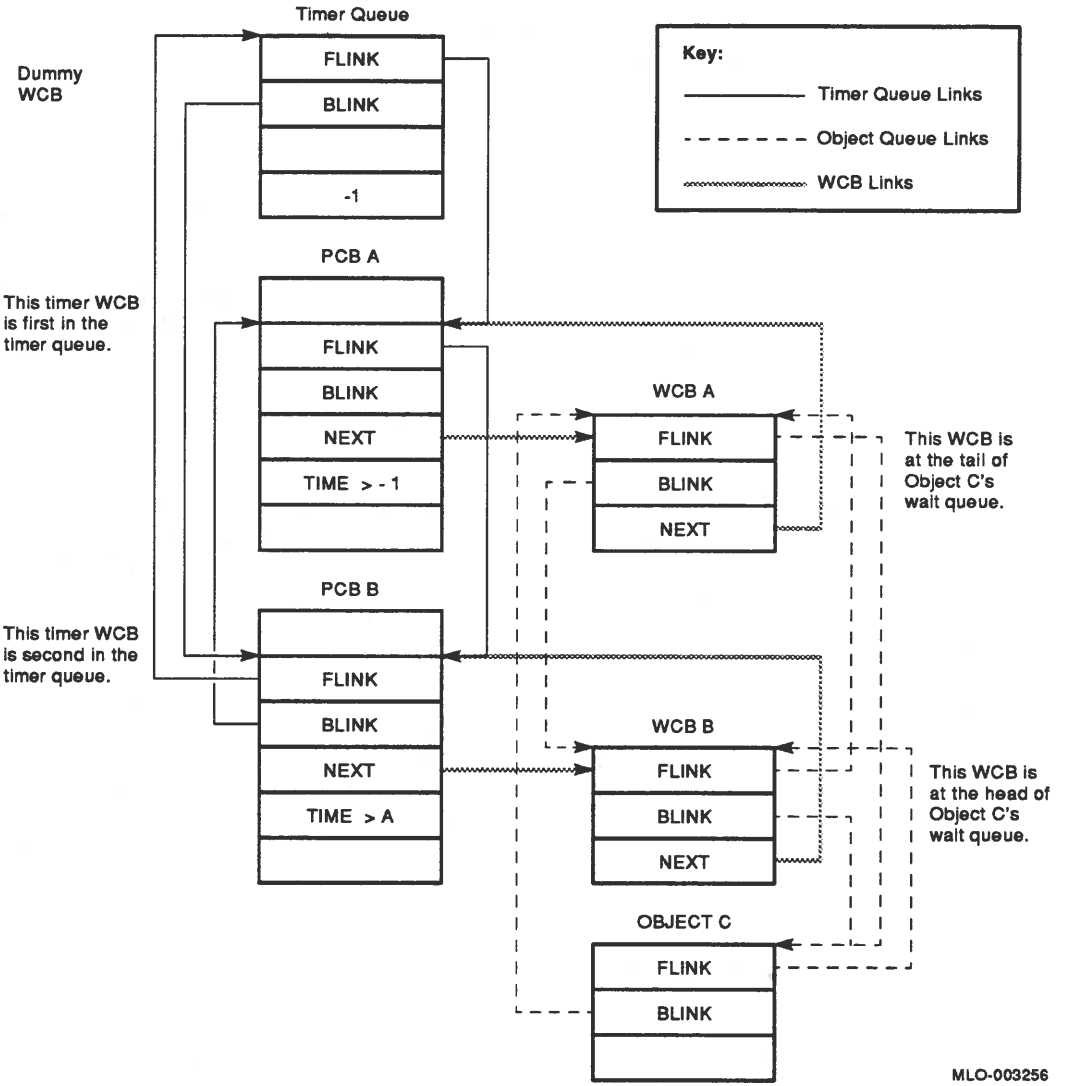
If the asynchronous exception can be delivered, control branches to a special exit sequence that clears general register R0 and then executes an REI instruction. The REI microcode initiates the delivery of the asynchronous exception. If the process handles the exception, it continues normal execution at the instruction following the CHMK instruction in the kernel vector for either KER$WAIT_ALL or KER$WAIT_ANY. As described in Section 11.1.4, the next instruction in the vector tests the value of R0. Because its value is 0, the vector reexecutes the CHMK instruction to reenter the KER$WAIT code. This time, with the asynchronous exception cleared, the process should succeed in entering the waiting state if its wait was not satisfied in the meantime.

11.2.7 Step 7 — Insert the WCBs into Wait Queues

The conditions of a process's wait are represented by its WCBs and their connections to the objects for which the process is waiting. The KER$WAIT procedure establishes those connections by inserting each WCB for the wait at the tail of the wait queue for its associated object. Timer WCBs are inserted at the appropriate position in the kernel's timer queue.

Figure 11–6 illustrates the network of connections established for two waiting processes. For the sake of clarity, each process is shown waiting for a timeout and for the same kernel object. To enter the WCBs into the appropriate queues, KER$WAIT walks the list of WCBs from the first to the last.

Figure 11–6: Two Processes in the Waiting State



If the wait has specified a timeout, KER$WAIT inserts the timer WCB into the timer queue as follows:

1. The address of the timer queue listhead is obtained from KER$GQ_TIME_QUEUE.
2. The address of the first entry in the queue is obtained.
3. The value in the timer WCB's time field is compared to the same field in the queued WCB.

   If the time value in the current WCB is less than that of the queued WCB, then the proper location has been found.

   Otherwise, the address of the next queue entry is obtained, and the comparison is repeated until the proper location or the end of the queue is found.
4. The INSQUE instruction is used to insert the current timer WCB before the WCB with a greater time value or at the end of the timer queue.

Next, KER$WAIT begins its scan of the list of object WCBs for the wait and inserts the WCBs in the appropriate object queues. The scan completes when the last WCB is queued. To queue an WCB to an object, KER$WAIT first obtains the address of the object from the WCB$A_OBJECT field of the current WCB. This address also points to the listhead for the object's queue.

Next, the type field in the object is checked. If the object is not an area object, the WCB is inserted at the tail of the object queue by the INSQUE instruction. If the object is an area, the address of its associated area control block (ACB), which contains the wait queue for the area, is obtained. The address of the ACB also points to the area's wait queue. The WCB is then inserted at the tail of the area queue with the INSQUE instruction.

11.2.8 Step 8 — Remove the Process from Execution

Once its WCBs have been queued to the object queues and the timer queue, the calling process is placed into the waiting state until conditions of its wait are satisfied. To inform the scheduler that the process is no longer eligible for execution, KER$WAIT places the process into the waiting state. The waiting process can unblock under the following circumstances:

- The conditions of its wait are satisfied. The kernel procedure that satisfies the process's wait, such as KER$SIGNAL or KER$SEND, removes the process from the waiting state and places it in the ready state, so that it is once again eligible to execute.
- The wait times out. When the kernel's hardware and software interval-clock ISRs discover that a process's wait has expired, the process is unblocked, regardless of the status of its other wait conditions.
- The waiting process receives an asynchronous exception. When an asynchronous exception is posted against a waiting process, the kernel moves the process from the waiting to the ready state. When the process next executes, the asynchronous exception is delivered. If the process handles the exception, it returns to its previous waiting state, unless its wait conditions have been satisfied in the meantime. In that case, it continues to execute.

To place a process into the waiting state, KER$WAIT first sets PCB$B_STATE field in the PCB to PCB$K_WAITING. This informs the scheduler that the process is ineligible to execute. Next, the scheduler subroutine KER$WAIT_PROCESS is called to remove the process from execution.

KER$WAIT_PROCESS (in module SCHEDPRO) immediately executes a SVPCTX instruction to save the hardware context of the process entering the waiting state and to switch execution to the interrupt stack. The scheduler then takes over to find the next process to place into execution.

11.3 Satisfying a Process Wait

A process placed into the waiting state stays there until its wait is satisfied, its wait times out, or the process itself is deleted. This section focuses on the kernel mechanisms for satisfying a wait on objects that are explicitly signaled. The process timeout mechanism is described in Section 5.3.5, and process deletion is described in Section 4.6.

Signaling an area, device, event, process, or semaphore object causes the object itself to change state and sets in motion a series of kernel operations that can cause one or more processes to leave the waiting state and enter the ready state. The KER$SIGNAL procedure signals area, event, process, and semaphore objects, and is described in Section 11.3.1. A separate procedure, KER$SIGNAL_DEVICE, signals device objects and is described in Section 11.3.2. Since KER$SIGNAL_

DEVICE is usually called from device interrupt service routines at elevated IPL, it is provided as a separate, optimized routine.

KER$SIGNAL and KER$SIGNAL_DEVICE both employ three kernel subroutines to test a process's wait conditions, make necessary alterations to the objects that satisfied the wait, and remove the process from the waiting state: KER$TEST_WAIT, KER$SATISFY_WAIT, and KER$UNWAIT, respectively, which are described in Section 11.3.3. These same routines are used by the KER$SEND routine to satisfy a process's wait on a port object.

11.3.1 KER$SIGNAL Procedure

Signaling an area, event, process, or semaphore object may change the object's state and allow one or more processes waiting for the object to leave the waiting state. The meaning of a call to KER$SIGNAL depends on the program's synchronization scheme. In general, a signal signifies one of the following: that a process no longer requires access to the resource it has acquired by waiting for it (area and semaphore objects), that a thread of execution is no longer required (process object), or that a specific synchronization point in the program has been reached (event and semaphore objects).

Whether the signal will enable a waiting process to unblock depends on the conditions of its wait. For a wait-any wait, only one signal will enable the process to unblock. For a wait-all wait, all the process's other wait conditions must also be satisfied at this time for this one signal to unblock it.

More specifically, signaling these objects has the following effects:

- Area object. Signaling an area increases its internal semaphore count by one and unblocks at most one waiting process.
- Event object. Signaling an event changes its state to signaled (signaling a signaled event has no effect) and unblocks all processes waiting for the event whose waits are otherwise satisfied.
- Process object. Signaling a process raises the asynchronous exception KER$_QUIT_SIGNAL for the process. If the process does not handle the exception, it is forced to exit. All processes waiting for the process are then unblocked if their waits are otherwise satisfied. (A wait condition for a process is also satisfied when the process terminates normally or is deleted.)

- Semaphore object. Signaling a semaphore increases its count by one and unblocks at most one waiting process.

KER$SIGNAL (in module SIGNAL) executes in kernel mode and expects two arguments: the address of an optional status variable and the object identifier for the object to be signaled. The procedure first calls the KER$TRANSLATE_OBJECT subroutine to obtain the address of the object to be signaled. For all objects except areas, the address of the object is also the address of the object's wait queue. For area objects, the address of the wait queue is the address of its associated area control block.

The procedure then dispatches execution based on the value in the object's OBJ$B_TYPE field. If the object is not valid for a signal operation, KER$_BAD_TYPE status is returned to the caller.

The following sections describe how KER$SIGNAL processes a signal for each kind of object.

11.3.1.1 Signaling an Area Object

When an area object is signaled, at most one waiting process can be unblocked. KER$SIGNAL takes the following steps to signal an area object:

1. The address of the area's associated, systemwide ACB is obtained from the ARA$A_ACB field in the area object.
2. The current count of the area's internal semaphore (ACB$L_COUNT in the ACB) is compared to its maximum count (ACB$L_LIMIT), which is always 1. If the two counts are equal, then the area has already been signaled, and KER$_COUNT_OVERFLOW is returned to the caller.
3. The LOCK macro is executed to ensure this thread's exclusive access to the global area queue.
4. The value of ACB$L_COUNT is increased by one to signify that the area object is now available to another process.
5. The KER$TEST_WAIT subroutine is called for each WCB in the area's wait queue until it finds an associated process whose wait is completely satisfied by this signal. Only one process can be unblocked by the signal. If a satisfied wait is not found, the procedure exits without unblocking a process.

6. If a process is found whose wait is completely satisfied, the KER$SATISFY_WAIT subroutine is called to decrease the area semaphore's count by 1. If this process was in a wait-all wait, the state of all the objects it was waiting for could be changed by KER$SATISFY_WAIT.
7. The KER$UNWAIT subroutine is called to unblock the process that is gaining access to the area and remove its WCBs from their respective wait queues.
8. KER$_SUCCESS is returned to the caller.

11.3.1.2 Signaling an Event Object

When an event object is signaled, every process waiting for the event can be unblocked. KER$SIGNAL takes the following steps to signal an event object:

1. The low bit in the EVT$B_STATE field is set to signify that the event's state is now signaled. It remains signaled until the state bit is cleared with the KER$CLEAR_EVENT procedure (see Figure 10–10).
2. The KER$TEST_WAIT subroutine is called for each WCB in the event's wait queue to search for associated processes whose waits are completely satisfied by this signal. Every process whose wait is now completely satisfied will be unblocked. If no satisfied waits are found, the procedure exits without unblocking a process.
3. For each process with a satisfied wait, the KER$SATISFY_WAIT subroutine is called. The subroutine makes no change to the event object itself to show that a process's wait on it is satisfied. For a process in a wait-all wait, the state of all the object's it was waiting for could be changed by KER$SATISFY_WAIT.
4. For each process whose wait is satisfied, the KER$UNWAIT subroutine is called to unblock the process and remove its WCBs from their respective wait queues.

11.3.1.3 Signaling a Process Object

When a process object is signaled, every process waiting for that process can be unblocked. A process signal depends on the delivery of a KER$_QUIT_SIGNAL asynchronous exception to the signaled process; this delivery mechanism is described in detail in Section 6.5. To set the delivery in motion, KER$SIGNAL simply sets the PCB$V_SIGNAL_QUIT bit in the PCB$B_REASON field of the PCB of the process being signaled. It then calls the KER$SIGNAL_AST subroutine, which sets the ASTLVL values required to allow the hardware to deliver the asynchronous exception. KER$SIGNAL then returns success to its caller.

When the signaled process receives the asynchronous exception, the kernel searches for a condition handler established by the signaled process. If no handler is found, or if the handler does not continue from the signal, the kernel calls KER$EXIT to force the signaled process to exit. KER$EXIT in turn calls KER$DELETE to delete the process.

KER$DELETE then takes the following steps to process the signal and possibly unblock any processes waiting for the signaled process to terminate:

1. An attempt is made to remove a WCB from the process object's wait queue. The WCB represents a waiting process. If no WCBs remain in the queue, the operation is complete.
2. The low bit in WCB$B_SATISFIED is set to signify that the process has terminated.

   This bit is set to record the fact that this portion of a waiting process's wait has been satisfied by the termination of the signaled process. This fact must be preserved for processes whose wait-all waits are not completely satisfied when the process they are waiting for is deleted. Because the PCB for the deleted process no longer exists, it cannot be used in subsequent tests for wait-all waits; the satisfied bit in the WCB must be used instead.
3. The KER$TEST_WAIT subroutine is called for each WCB in the process's wait queue to search for associated processes whose waits are satisfied by the termination of this process. Every process whose wait is now completely satisfied will be unblocked. If no satisfied waits are found, KER$DELETE deletes the process without unblocking another process.
4. For each process with a satisfied wait, the KER$SATISFY_WAIT subroutine is called. The subroutine makes no change to the process object itself to show that a process's wait for it is satisfied.

5. For each process whose wait is satisfied, the KER$UNWAIT subroutine is called to unblock the process and remove its WCBs from their respective wait queues.

11.3.1.4 Signaling a Semaphore Object

When a semaphore object is signaled, at most one waiting process can be unblocked. KER$SIGNAL takes the following steps to signal a semaphore object:

1. The current count of the semaphore (SEM$L_COUNT) is compared to its maximum count (SEM$L_LIMIT). If the two counts are equal, then the semaphore has already been signaled to its maximum count, and KER$_COUNT_OVERFLOW is returned to the caller.
2. The value of SEM$L_COUNT is increased by one to signify that the semaphore object is now available to another process.
3. The KER$TEST_WAIT subroutine is called for each WCB in the semaphore's wait queue until it finds an associated process whose wait is completely satisfied by this signal. Only one process can be unblocked by the signal. If a satisfied wait is not found, the procedure exits without unblocking a process.
4. If a process is found whose wait is completely satisfied, the KER$SATISFY_WAIT subroutine is called to decrease the semaphore's count by 1. If this process was in a wait-all wait, the state of all the objects it was waiting for could be changed by KER$SATISFY_WAIT.
5. The KER$UNWAIT subroutine is called to unblock the process that is gaining access to the semaphore and remove its WCBs from their respective wait queues.
6. KER$_SUCCESS is returned to the caller.

11.3.2 KER$SIGNAL_DEVICE

Waiting for a device object allows a device driver process to synchronize its execution with an interrupt service routine (ISR). The driver process blocks its execution until a device interrupt arrives and is serviced. Within the ISR, a call to KER$SIGNAL_DEVICE results in the ultimate unblocking of the waiting driver process.

Unlike the KER$SIGNAL procedure, KER$SIGNAL_DEVICE (in module SIGNALDEV) does not directly unblock the waiting process; rather, it inserts the signaled device object into the global device queue to be serviced at IPL 8, instead of device IPL. Deferring the work of unblocking the waiting process minimizes the time IPL must remain at device level. This mechanism is roughly equivalent to the role of fork processing under the VMS operating system.

Because it normally executes at device IPL, KER$SIGNAL_DEVICE contains only a handful of instructions — the minimum number to insert the signaled device into the device queue. In fact, the VAXELN Pascal compiler generates a JSB subroutine call to a special kernel vector, KER$SIGNAL_DEVICE_R2. This vector then executes a BRW instruction to transfer control to the start of the actual code for KER$SIGNAL_DEVICE, KER$SIGNAL_DEVICE_R2_S. Control never returns to the kernel vector; the RSB that exits the kernel code returns control directly to the ISR.

VAX languages whose processors do not generate JSB linkages must call KER$SIGNAL_DEVICE through another kernel vector, KER$SIGNAL_DEVICE. This kernel vector then uses the BSBW instruction to branch to the location KER$SIGNAL_DEVICE_S. Code at this entry point executes five instructions to move the arguments expected by the main routine code at KER$SIGNAL_DEVICE_R2_S into general registers, then branches to that location. The RSB that exits the routine will then return control to the kernel vector at KER$SIGNAL_DEVICE, which executes a RET instruction to return control to the ISR. Kernel vectors and their linkages to kernel routine code are described in Chapter 8.

The register input expected by KER$SIGNAL_DEVICE_R2_S is the relative device number of the device object to be signaled and the address of the interrupt dispatch block (IDB) for the device, which contains the array of relative device object addresses. (These register arguments are automatically established by the VAXELN Pascal compiler; for other languages, they are established at the entry point KER$SIGNAL_DEVICE_S). The structure and function of the IDB are described in Section B.9.

Executing in kernel mode at device IPL, KER$SIGNAL_DEVICE_R2_S executes as follows:

1. The address of the signaled device object is obtained by indexing the device list located at IDB$A_DEVICE_LIST by the relative device number. If no relative device number was specified, the first

device in the list is used. If a null address is obtained, the routine returns KER$_BAD_VALUE.

2. The interlocked instruction BBSSI is executed to set the device lock in the low bit of DEV$B_LOCK. If the lock is already set, then the device has already been signaled, and the routine exits with success status.
3. The interlocked instruction INSQTI is executed to insert the device object at the tail of the kernel's device queue, located at KER$GQ_DEVICE_QUEUE. If the insertion fails because of another interlock, the instruction is repeated until it succeeds.
4. If this device object was the first in the device queue, an IPL 8 software interrupt is requested to activate the device queue ISR, KER$DEVICE_SIGNAL.
5. The routine returns success.

When it runs, the IPL 8 ISR, KER$DEVICE_SIGNAL (also in module SIGNALDEV), walks the device queue and unblocks waiting processes. KER$GQ_DEVICE_QUEUE can contain a number of device objects, not just a single object from one device signal. The ISR processes all the devices in the queue at this time.

KER$DEVICE_SIGNAL, running at IPL 8 on the interrupt stack, executes as follows:

1. A device object is removed from the queue. If the queue is empty, an REI instruction is executed to dismiss the interrupt.
2. The SEIZE macro is executed to prevent other processors from accessing to the device queue.
3. The lower bit in the DEV$B_STATE field is set to signify that the device has been signaled.
4. The interlocked instruction BBCCI is executed to clear the low bit in DEV$B_LOCK, so that the device can again be signaled.
5. The KER$TEST_WAIT subroutine is called for each WCB in the device's wait queue until it finds an associated process whose wait is completely satisfied by this signal. Only one process can be unblocked for each signaled device. If no satisfied waits are found, the routine does not unblock any processes waiting for this device object; it moves on to the next signaled device object in the device queue.

6. If a process is found whose wait is completely satisfied, the KER$SATISFY_WAIT subroutine is called to clear the low bit in DEV$B_STATE, signifying that a process's wait for the device is satisfied. If this process was in a wait-all wait, the state of all the objects it was waiting for could be changed by KER$SATISFY_WAIT.
7. The KER$UNWAIT subroutine is called to unblock the process that was waiting for the device and remove its WCBs from their respective wait queues.
8. The routine loops back to service the next device object in the device queue.

11.3.3 Kernel Subroutines to Support Object Signaling

The kernel code that manages the blocking and unblocking of a process relies on a set of subroutines in module KERNELSUB. These three subroutines, KER$TEST_WAIT, KER$SATISFY_WAIT, and KER$UNWAIT, are described in the following sections.

11.3.3.1 KER$TEST_WAIT

When the kernel needs to determine whether a process's wait conditions have been completely satisfied, it calls the internal subroutine KER$TEST_WAIT, passing it the address of the WCB that represents the waiting process. That WCB provides the information that the subroutine requires to test the wait conditions of the process: the kind of wait (all or any) and the linkages to the other WCBs established for the wait. KER$TEST_WAIT returns the status of the wait and, if the wait is satisfied, the address of the WCB that satisfied the wait.

How KER$TEST_WAIT functions depends on whether the waiting process called KER$WAIT_ALL or KER$WAIT_ANY, as determined by the setting of the low bit in the WCB$B_WAIT field in the WCB passed in the subroutine call. If the WCB represents a wait-any state, then the test succeeds as soon as a WCB that reflects a satisfied state is found. If it is a wait-all wait, then the test succeeds only when all the WCBs in the wait reflect a satisfied state.

The heart of KER$TEST_WAIT is the local subroutine TEST_WAIT. When passed the address of a WCB, TEST_WAIT tests the wait status reflected by the WCB and returns success if the wait is satisfied. The test depends on the type of object the WCB is queued to, as determined

by the value of WCB$B_OBJECT_TYPE. Table 11–5 describes the test performed for each type of object.

Table 11–5: Wait Tests Performed by KER$TEST_WAIT

Object	Field Tested	Condition to Satisfy Wait
Area	ACB$L_COUNT	Count must be greater than 0.
Device	DEV$B_STATE	Low bit in field must be set.
Event	EVT$B_STATE	Low bit in field must be set.
Port	PRT$L_COUNT PRT$W_STATE	Message count must be greater than 0; or the state bit for port full or circuit connection/disconnection must be set.
Process	WCB$B_SATISFIED	Low bit in WCB field must be set. Object itself cannot be tested because it may have already been deleted.
Semaphore	SEM$L_COUNT	Count must be greater than 0.

Timer WCBs are ignored by the test. Instead, timeouts are handled handled independently by the software timer ISR, as described in Section 5.3.5.

When it is called, KER$TEST_WAIT checks the setting of the low bit in WCB$B_WAIT. If it is set, KER$TEST_WAIT performs its wait-all test as follows:

1. The address of the current WCB is passed to the TEST_WAIT subroutine. The subroutine tests the WCB as described in Table 11–5 and returns.
2. The status returned by the subroutine is tested. If the test of this object failed, then all wait conditions have not been satisfied. KER$TEST_WAIT therefore returns failure status to its caller. Otherwise, execution continues with the next WCB.
3. Because the last test succeeded, the address of the next WCB in the list for this wait is obtained from WCB$A_NEXT.

   If all the WCBs have been tested at this point, execution continues with the next step. Otherwise, the loop repeats from the beginning.
4. Since all the WCBs have now been tested successfully, the entire wait condition is satisfied. To show that the wait did not time out, the value of the WCB$B_ARGUMENT field of the first object WCB in the list is returned to the caller, along with success status.

If WCB$B_WAIT indicates a wait-any wait, KER$TEST_WAIT performs its test as follows:

1. The address of the current WCB is passed to the TEST_WAIT subroutine. The subroutine tests the WCB as described in Table 11–5 and returns.
2. The status returned by the subroutine is tested. If the test of this object succeeded, the wait conditions have been satisfied. KER$TEST_WAIT therefore returns success status and the address of the WCB that satisfied the wait. The caller of KER$TEST_WAIT will use the WCB$B_ARGUMENT field in this WCB to return the wait result argument to its caller.

   If the test of the object failed, execution continues with the next step.
3. Because the last test failed, the address of the next WCB in the list is obtained from WCB$A_NEXT.

   If all the WCBs have been tested at this point, the entire test has failed, and that status is returned to the caller. Otherwise, the loop repeats from the beginning.

11.3.3.2 KER$SATISFY_WAIT

When the kernel determines that an object signal (or process termination or message arrival) will cause a waiting process to be unblocked, it calls the internal subroutine KER$SATISFY_WAIT to make appropriate changes to the object or objects that process was waiting for.

The address of the WCB that apparently satisfied the process's wait is passed to the subroutine. That WCB provides the information the subroutine needs to satisfy the wait conditions of the process: the kind of wait (all or any) and the linkages to the other WCBs established for the wait. The subroutine alters only area, device, and semaphore objects.

How KER$SATISFY_WAIT functions depends on whether the process being unblocked called KER$WAIT_ALL or KER$WAIT_ANY to enter its waiting state, as determined by the setting of the low bit in the WCB$B_WAIT field in the WCB passed in the subroutine call. If the WCB represents a wait-any state, then only the state of the single satisfying object may have to be changed. If it is a wait-all wait, then the states of all the objects in the wait may have to be changed.

The heart of KER$SATISFY_WAIT is the local subroutine SATISFY_WAIT. When passed the address of a WCB, SATISFY_WAIT checks the value of WCB$B_OBJECT_TYPE and takes the action necessary to change the state of the object pointed to by the WCB. Table 11–6 describes the changes performed for each type of object.

Table 11–6: Changes to Objects Performed by KER$SATISFY_WAIT

Object	Field Changed	Change
Area	ACB$L_COUNT	Count increased by 1.
Device	DEV$B_STATE	Low bit in field cleared.
Semaphore	SEM$L_COUNT	Count increased by 1.

When it is called, KER$SATISFY_WAIT checks the setting of the low bit in WCB$B_WAIT. If WCB$B_WAIT indicates a wait-all wait, KER$SATISFY_WAIT calls SATISFY_WAIT repeatedly as a subroutine, passing each successive WCB pointed to by WCB$A_NEXT, until all have been processed. It then executes an RSB (Return from Subroutine) instruction to return to its caller.

If WCB$B_WAIT indicates a wait-any wait, KER$SATISFY_WAIT simply branches to SATISFY_WAIT rather than calling it as a subroutine. This means that SATISFY_WAIT makes the necessary alteration only to the object that satisfied the wait. The RSB within SATISFY_WAIT then returns control directly to the caller of KER$SATISFY_WAIT.

11.3.3.3 KER$UNWAIT

When the kernel discovers a process whose wait has been satisfied, it calls the internal subroutine KER$UNWAIT to remove the process from its waiting state and place it in the ready state. This subroutine also removes the process's WCBs from their object wait queues and returns the wait's completion status and wait result to the unblocked process.

As Figure 11–5 shows, the KER$WAIT kernel vectors expect R0 to contain the wait completion status and R1 to contain the wait result after the wait is satisfied. Since the waiting process is not executing, KER$WAIT must cause these values to be returned by inserting them into the save areas for R0 and R1 in the process's hardware context block (PTX). When the process once again executes, the LDPCTX instruction executed by the scheduler restores the correct values for R0 and R1 from the PTX. The restored PC for the process points to the instruction following the CHMK instruction in the KER$WAIT kernel

vector. That instruction is the vector's test of R0 to determine whether the wait was successfully concluded or interrupted by an asynchronous exception.

KER$UNWAIT expects two input values: the address of the WCB that satisfied the wait and the completion status for the wait. Normally, the completion status is KER$_SUCCESS, but when KER$UNWAIT is called by the KER$SIGNAL_AST routine to unblock a process with a pending asynchronous exception, a status value of 0 is specified. If the process handles the exception, normal execution continues in the KER$WAIT kernel vector, which encounters the 0 in R0. This causes the vector to reenter the wait code and possibly return the process to its interrupted waiting state, if its wait was not satisfied in the meantime.

Likewise, when an object for which a process is waiting is deleted, the code in KER$DELETE calls KER$UNWAIT for each process waiting for the deleted object. In the call, it specifies the return status of KER$_BAD_VALUE, which becomes the completion status for the process's call to KER$WAIT.

KER$UNWAIT executes as follows to unblock a process:

1. The address of the PCB is obtained from WCB$A_PCB, and the address of the PTX is obtained from PCB$A_PTX.
2. The completion status is copied to PTX$L_R0, the save area for register R0.
3. The value of WCB$B_ARGUMENT, the wait result, is copied to PTX$L_R1, the save area for register R1.
4. The current WCB is removed from its wait queue (object or timer queue) with the REMQUE instruction. If the low bit in WCB$B_SATISFIED is set (the wait was for a process), this step is not executed, because the WCB was removed from the process object's wait queue when the process being waited for was deleted.
5. The address of the next WCB in the wait is obtained from WCB$A_NEXT, and the previous step is executed until all the WCBs have been removed from their associated object queues.
6. The scheduler subroutine KER$READY_PROCESS is called to place the unblocked process in the ready state.

KER$READY_PROCESS in module SCHEDPRO sets the value of PCB$B_STATE to PCB$K_READY for the process and attempts to schedule the newly readied process. If the unblocked process has a higher priority than the currently executing process, the scheduler

causes the unblocked process to preempt the current process once it returns from its kernel procedure call.

Appendix A

Kernel Parameters and Data

This appendix summarizes the global parameters and data used by the VAXELN Kernel. Section A.1 describes kernel parameters, and Section A.2 describes kernel data.

A.1 Kernel Parameters

Kernel parameters are defined in module PARAMETER. They tranmit information about the system from the System Builder to the VAXELN Kernel. The contents of the parameter block are shown in Table A–1. The parameter block itself is described in Section 2.3.3.

Table A–1: Kernel Parameters

Symbol	Meaning
KER$GA_DECW_SCR	Address of a record describing the configuration of the DECwindows server
KER$GA_DEVICE_LIST	Offset from this location to the first system configuration record
KER$GA_FP_EMULATOR	Address of floating-point instruction emulator
KER$GA_KERNEL_DEBUG_CODE	Address of debugger code resident in the system image
KER$GA_KERNEL_DEBUG_DATA	Address of kernel debugger data
KER$GA_PROGRAM	Address of program list
KER$GA_SHARE_LIST	Address of shareable image table

Table A–1 (Cont.): Kernel Parameters

Symbol	Meaning
KER$GA_STRING_EMULATOR	Address of string instruction emulator
KER$GA_STRING_EMULATOR_EX	Address of string emulator exception fixup handler
KER$GB_CONSOLE_PRESENT	Boolean flag for requesting console support
KER$GB_DECW_CONSOLE	Boolean flag for requesting DECwindows console driver
KER$GB_INITIAL_AUTH_REQUIRED	Boolean flag for requesting authorization service
KER$GB_INITIAL_ERRLOG_ENABLE	Boolean flag for requesting error logging
KER$GB_INITIAL_NAME_SERVER	Boolean flag for requesting name service
KER$GB_INITIAL_TRIGGER_ ENABLED	Boolean flag for enabling trigger boot
KER$GB_JOB_SCHED_PREEMPT	Boolean flag for requesting job rotation on preemption
KER$GB_QBUS_RESPONSE_ OPTIMIZE	Boolean flag for requesting Q-bus interrupt optimization
KER$GB_REMOTE_DEBUG_PRESENT	Boolean flag for requesting the remote debugger
KER$GL_BIOS_OFFSET	Byte offset to code for the console I/O subsystem
KER$GL_DUMP_OFFSET	Byte offset to code for system dump routines
KER$GL_INITIAL_DATAGRAM_SIZE	Maximum size in bytes of a DECnet datagram
KER$GL_INITIAL_DEFAULT_UIC	System default UIC
KER$GL_MEMORY_LIMIT	Physical memory limit (in pages)
KER$GL_TIME_INTERVAL	System time update interval in 100-nanosecond intervals
KER$GQ_INITIAL_CONNECT_ TIMEOUT	Circuit connection timeout value in seconds
KER$GQ_INITIAL_NODE_ADDRESS	Local node's Ethernet address
KER$GT_ANNOUNCE_STR	System announcement string
KER$GT_INITIAL_NODE_NAME	Local node name string
KER$GW_EMB_COUNT	Number of error log buffers to preallocate

Table A–1 (Cont.): Kernel Parameters

Symbol	Meaning
KER$GW_IO_SIZE	Number of pages in system (communication) region
KER$GW_ISTACK_SIZE	Size in pages of the interrupt stack
KER$GW_NAME_SIZE	Number of entries in a name table
KER$GW_P0_INITIAL_SLOT_SIZE	Size in pages of a P0 page table slot
KER$GW_P0_SLOT_COUNT	Number of P0 page table slots
KER$GW_P1_INITIAL_SLOT_SIZE	Size in pages of a P1 page table slot
KER$GW_P1_SLOT_COUNT	Number of P1 page table slots
KER$GW_POOL_SIZE	Number of pages in the system pool
KER$GW_PORT_SIZE	Number of entries in the port object port
KER$GW_SYSTEM_SIZE	Size in pages of system image

A.2 Kernel Data

Kernel data items are defined in module SYSTEMDAT. They record dynamic information about the system to the VAXELN Kernel. The contents of the data block are shown in Table A–2. The data block itself is described in Section 2.3.2.

Table A–2: Kernel Data

Symbol	Meaning
KER$AA_CONIO_CONTEXT	Address of console I/O context block
KER$AA_CURRENT_JCB	Pointer to job control block of the currently executing job (one pointer for each processor)
KER$AA_NEXT_JCB	Pointer to the job control block of the job next in line for execution (one pointer for each processor)
KER$AB_REASON	Interprocessor interrupt reason bit mask (one for each processor)
KER$AQ_READY_HEAD	Head of scheduler's queue of ready jobs
KER$AQ_READY_TAIL	Tail of scheduler's queue of ready jobs

Table A–2 (Cont.): Kernel Data

Symbol	Meaning
KER$AW_CLASS_MASK	An array of bit masks, one for each job priority, containing a bit for each processor that is executing a job of the respective priority
KER$B_NBI	Boolean flag for an NBI nexus
KER$B_QBUS	Boolean flag for a MicroVAX II processor
KER$B_VAXBI	Boolean flag for a VAXBI processor
KER$GA_ACCESS_VIOLATION	Address of the access control violation service routine in module EXCEPTION (used by the VAX emulators)
KER$GA_ADAPTER_LIST	Adapter control block listhead
KER$GA_BIPORT_DATA	Pointer to a BI port local data block
KER$GA_BUGSTACK	Address of the bugcheck-in-progress stack (one for each processor)
KER$GA_CALL_HANDLER_PC	Virtual address within module EXCEPTION where condition handlers are called
KER$GA_CCA_ADDR	Virtual address of VAX 6200 processor header
KER$GA_CCA_BUFFER_BASE	Virtual address of array of VAX 6200 buffer addresses
KER$GA_CONIO_CODE	Address of the code for the console I/O subsystem
KER$GA_CPUREGSP	Virtual address of processor-specific register address space
KER$GA_CRASHLOG	Address of the crash-restart log (one for each processor)
KER$GA_DIGITAL_RESERVED	Virtual address of the reserved operand fault service routine in module EXCEPTION (used by the VAX emulators)
KER$GA_DISPATCH_EXCEPTION	Virtual address of the location in module EXCEPTION where the search for a condition handler begins (used by VAX emulators)
KER$GA_DUMP_CB	Virtual address of dump control block
KER$GA_DUMP_CODE	Virtual address of system dump image
KER$GA_ERRFMT_JCB	Virtual address of the job control block for the error formatting job

Table A–2 (Cont.): Kernel Data

Symbol	Meaning
KERGA_EXEREFLECT	Virtual address in module EXCEPTION of the uniform condition-dispatching logic (used by the VAX emulators)
KER$GA_IO_BASE	Virtual address of the I/O database
KER$GA_ISTACK_BASE	Virtual address of the base of the interrupt stack (one for each processor)
KER$GA_MACHINECHK_DATA	Virtual address of the data block for the machine-check handler (one for each processor)
KER$GA_NBIA_BASE	NBIA base virtual address (one for each NBIA)
KER$GA_NBIB_BASE	NBIB base virtual address (one for each NBIB)
KER$GA_NEXUS_BASE	Virtual address of table of VAX 6200 processor register addresses
KER$GA_P0_SLOT_BASE	Virtual address of the P0 page table slots
KER$GA_P1_SLOT_BASE	Virtual address of the P1 page table slots
KER$GA_POOL_BASE	Virtual address of the system dynamic pool
KER$GA_PORT_BASE	Virtual address of the port object table
KER$GA_PROGRAM_LIST	Program descriptor listhead
KER$GA_REGION_BASE	Virtual address of the base of the communication region
KER$GA_RESTART_ISTACK	Multiprocessor stack restart data (one for each processor)
KER$GA_RPB	Virtual address of the restart parameter block
KER$GA_SCB_BASE	Virtual address of the system control block
KER$GA_SHAREABLE_IMAGE_LIST	Shareable image table listhead
KER$GA_SPT_BASE	Virtual address of the system page table
KER$GA_SPT_PHYSICAL	Physical address of the system page table
KER$GA_STARTUP_PORT	Address of the start-up job's job port
KER$GA_UNWIND	Address in module RAISE of the KER$UNWIND_S procedure (used by the VAX emulators)
KER$GB_AMP	Boolean flag for closely coupled symmetric multiprocessing
KER$GB_AUTH_REQUIRED	Boolean flag for requesting authorization service

Table A–2 (Cont.): Kernel Data

Symbol	Meaning
KER$GB_BI_NUMBER	BI number of the current processor if it is a KA800
KER$GB_CPUHARD_INDEX	Processor hardware dispatching index
KER$GB_CPU_TO_WATCH	Array of CPU numbers for CPU watchdog timers
KER$GB_CPU_TYPE	Processor type code
KER$GB_CVAX	Boolean flag for MicroVAX 3000 processor
KER$GB_EPA_ENABLE	Boolean flag for enabling VAXELN Performance Utility
KER$GB_ERRLOG_ENABLE	Boolean flag for requesting error logging
KER$GB_NAME_SERVER	Boolean flag for requesting the name service
KER$GB_NODE_NUMBER	Node number of the NBIB on each BI (one for each BI)
KER$GB_POWERFAIL	Boolean flag to enable power-failure recovery
KER$GB_PTOOEY	Boolean flag for a KA800 processor
KER$GB_RTVAX	Boolean flag for a KA620 processor
KER$GB_SMP_FLAGS	Flags to indicate that the system is running under tightly coupled symmetric multiprocessing
KER$GB_SYS_TYPE	Copy of internal processor identification register (SIDEX)
KER$GB_TIME_SET	Flag to indicate that the system time has been set
KER$GB_TRIGGER_ENABLED	Boolean flag for enabling booting over the network
KER$GL_ACTIVE_SUMMARY	Summary bit mask representing which of the 32 job priorities has an active job on a processor in the system
KER$GL_BUGCODE	Processor fatal system bugcheck code
KER$GL_BUGCPUID	Physical identification of the processor suffering a fatal bugcheck
KER$GL_DATAGRAM_SIZE	Maximum size in bytes of a DECnet datagram
KER$GL_DEFAULT_UIC	Default user identification code
KER$GL_ERRFMT_WAKEUP	Object identifier for the event object on which the error format job waits
KER$GL_FIRST_WRT_PAGE	Virtual page number of the first writeable page in the system

Table A–2 (Cont.): Kernel Data

Symbol	Meaning
KER$GL_FREEZE_SYSTEM	Interprocess freeze lock
KER$GL_IPINT_MASK	Interprocessor interrupt mask
KER$GL_MULTIPROCESSOR_LOCK	Multiprocessor lock containing spinlock bits for interprocessor synchronization
KER$GL_NUMBER_AVAIL_CPU	Number of available CPUs
KER$GL_PAGE_BITMAP_START	Base physical address of the page frame allocation bitmap on a KA800 secondary processor
KER$GL_POOL_FREE	Negated number of free pool blocks in the system dynamic memory list
KER$GL_PRIMARY_CPUNUM	Number of the primary CPU
KER$GL_PRT_BIPORT	Value for the port ID field
KER$GL_READY_SUMMARY	Summary bit mask representing which of the 32 job priorities contains a ready job
KER$GL_SAVED_IPL	Saved IPL value for use by symmetric multiprocessing virtual console
KER$GL_SMP_INFO	Mask indicating available CPUs in a tightly coupled symmetric multiprocessing system
KER$GL_SPT_LENGTH	Length in longwords of the system page table
KER$GL_VCX_RXDB	Virtual console receive buffer
KER$GL_VCX_TXDB	Virtual console transmit buffer
KER$GL_WATCHDOG	Processor watchdog time counter, one for each CPU
KER$GL_WATCHDOG_VALUE	Timer value for detecting unresponsive processors in a multiprocessor configuration
KER$GL_XBIA_INIT	Table of initialized XBI adapters on the VAX 6200
KER$GQ_AREA_LIST	Listhead for queue of area control blocks
KER$GQ_CLOCK_OFFSET	Accumulator for all changes to the system time
KER$GQ_CONNECT_TIMEOUT	Interval time for circuit connection timeouts
KER$GQ_DEVICE_QUEUE	Listhead for queue of signaled device objects
KER$GQ_EMB_AVAIL	Listhead for queue of available error log buffers
KER$GQ_EMB_POSTED	Listhead for queue of posted error log buffers
KER$GQ_HOST_ADDRESS	Host node Ethernet address

Table A–2 (Cont.): Kernel Data

Symbol	Meaning
KER$GQ_IDB_LIST	Listhead for queue of interrupt dispatch blocks
KER$GQ_IDLE_TIME	Array of idle times for processors
KER$GQ_NODE_ADDRESS	Ethernet or DECnet address of the local node
KER$GQ_POOL_HEAD	Listhead fof queue of system pool blocks
KER$GQ_PREV_JOB_TIME	Total time used by previous jobs (jobs that have been deleted)
KER$GQ_START_TIME	System initialization time or time of last setting of system time
KER$GQ_SYSTEM_JOB	Listhead for queue jobs
KER$GQ_SYSTEM_TIME	Absolute system time
KER$GQ_TIME_QUEUE	Listhead for queue of timer wait control blocks
KER$GQ_TOD	Host node time-of-day value
KER$GR_EPA_DATA	Virtual address where VAXELN Performance Utility data is maintained
KER$GR_LOCAL_NAME	Port object identifier for the name server for local KA800 processors in a closely coupled symmetric multiprocessing system
KER$GR_LOCAL_TABLE	Local name table descriptor, containing the address of the first table listhead and the total number of listheads in the table
KER$GR_NETWORK_CONNECT	Port object identifier for the Network Service remote circuit connection port
KER$GR_NETWORK_DATAGRAM	Port object identifier for the Network Service remote datagram port
KER$GR_NETWORK_NAME	Port object identifier for the Name Service port
KER$GR_P0_SLOT_BITMAP	Descriptor for the P0 page table slot allocation bitmap
KER$GR_P1_SLOT_BITMAP	Descriptor for the P1 page table slot allocation bitmap
KER$GR_PAGE_BITMAP	Descriptor for the page frame allocation bitmap
KER$GR_REGION_BITMAP	Descriptor for the communication region allocation bitmap
KER$GT_HOST_NAME	String descriptor for host node name

Table A–2 (Cont.): Kernel Data

Symbol	Meaning
KER$GT_NODE_NAME	String descriptor for local node name
KER$GW_CNT_POSTED	Count of posted error log buffers
KER$GW_CPUAVAIL_MASK	Mask of all available processors
KER$GW_CPU_ACTIVE	Summary bit mask representing active processors in the system
KER$GW_CPU_IDLE	Summary bit mask representing which of the processors in a multiprocessing system is idle
KER$GW_EMB_SIZE	Size in bytes of an error log buffer
KER$GW_ERRSEQ	Error log entry sequence number
KER$GW_JOB_GENERATION	Job generation number
KER$GW_MAX_POSTED	Maximum number of error log buffers that can be posted before the error format job runs
KER$GW_P0_SLOT_LENGTH	Size in pages of a P0 page table slot, which includes the page table itself and the associated allocation bitmap
KER$GW_P0_SLOT_SIZE	Size in pages of a P0 page table slot
KER$GW_P1_SLOT_LENGTH	Size in pages of a P1 page table slot, which includes the page table itself and the associated allocation bitmap
KER$GW_P1_SLOT_SIZE	Size in pages of a P1 page table slot
KER$GW_PORT_FREE	Index of the next free entry in the port object table
KER$GW_VCX_RXCIE	Virtual console receive-enable register
KER$GW_VCX_RXCRDY	Virtual console receive-ready register
KER$GW_VCX_TXCIE	Virtual console transmit-enable register
KER$GW_VCX_TXCRDY	Virtual console transmit-ready register

Appendix B

Kernel Data Structures

This appendix summarizes the data structures used by the VAXELN Kernel. Each of the following sections describes the purpose of each structure and its location and source for allocation.

B.1 ACB — Area Control Block

Purpose:	Describes characteristics and state of a shared area of memory.
Location:	Linked into global area queue with listhead at KER$GQ_AREA_LIST.
Allocated from:	System pool.
References:	Figure B–1, Table B–1.

Figure B–1: Structure of an Area Control Block

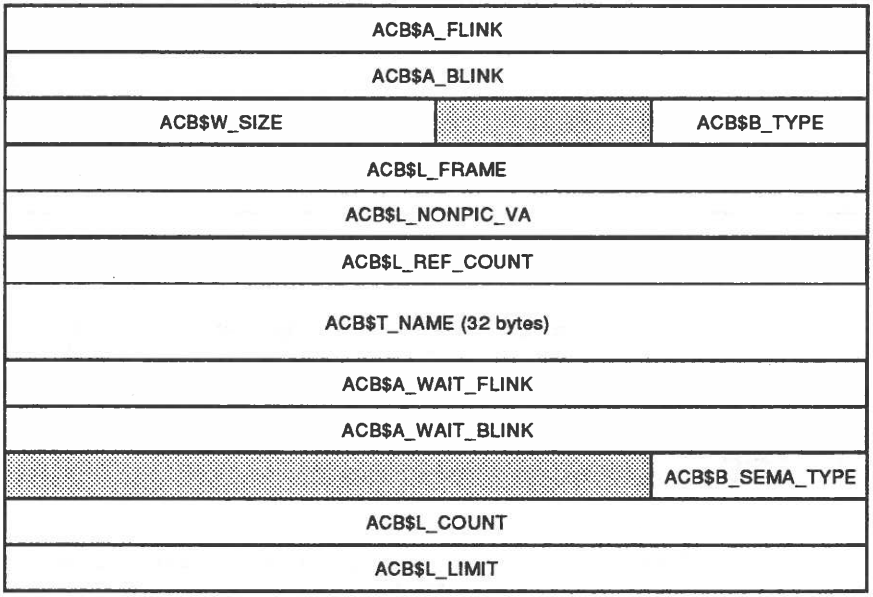


Table B–1: Area Control Block Fields

Field	Meaning
ACB$A_FLINK ACB$A_BLINK	The links for inserting the ACB into the kernel's queue of ACBs, located at KER$GQ_AREA_LIST.
ACB$B_TYPE	The object type: OBJ$K_AREA_CONTROL_BLOCK.
ACB$W_SIZE	Number of characters in the area's name string.
ACB$L_FRAME	The starting page frame number for the physical memory occupied by the area buffer. This value is used for mapping the area buffer into the P0 virtual address space of a job that shares the area.
ACB$L_NONPIC_VA	The P0 virtual address at which the area buffer is mapped into a sharing job's address space. This field is used only if the creator of the area specifies an explicit virtual address for the area buffer.
ACB$L_REF_COUNT	The number of jobs that have called KER$CREATE_AREA for this area. This count is set to 1 when the area is first created. As each additional job calls KER$CREATE_AREA to map this area, the reference count is increased by one. As each job deletes the area, the reference count is decreased by 1. When the count reaches 0, the ACB and the area buffer are deallocated.
ACB$T_NAME	The ASCII string containing the uppercase name of the area. The maximum size of the string is 31 characters.
ACB$A_WAIT_FLINK ACB$A_WAIT_BLINK	The listhead for the queue of processes waiting for this area. A WCB representing a waiting process is inserted into this queue by the KER$WAIT procedure. When the area is signaled, the first process whose wait is completely satisfied is removed from the queue and unblocked.

Table B–1 (Cont.): Area Control Block Fields

Field	Meaning
ACB$B_SEMA_TYPE	Unused.
ACB$L_COUNT	The count value for the area's internal semaphore, signifying the number of processes that can yet access the area. This field is initialized to 1, meaning that at most one process can access the area. A successful wait for an area decreases the count to 0. Signaling an area, in turn, restores its count to 1. A count of 0, then, means that a process will have to wait for the area to be signaled before gaining access to its data buffer.
ACB$L_LIMIT	The limit value for the area's internal semaphore, signifying the maximum number of processes that may have access to the area. This value is always 1. When the area count equals this limit, processes must wait for the area to be signaled.

B.2 ADP — Adapter Control Block

Purpose:	Describes characteristics and state of an I/O adapter.
Location:	Linked into global adapter queue with listhead at KER$GA_ADAPTER_LIST.
Allocated from:	System pool.
References:	Figure B–2, Table B–2.

Figure B–2: Structure of an Adapter Control Block

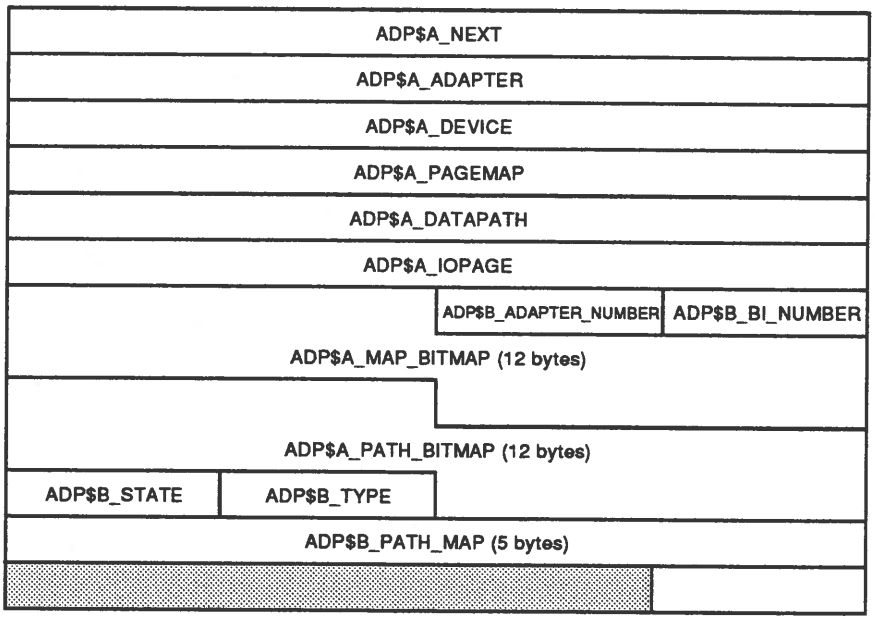


Table B–2: Adapter Control Block Fields

Field	Meaning
ADP$A_NEXT	The address of next the ADP in the system's list of adapters, located at KER$GA_ADAPTER_LIST. A new ADP is created with a call to the KER$CREATE_ADAPTER subroutine (in module INITIAL) from a processor-specific initialization routine. Each new ADP is placed at the head of the adapter list.
ADP$A_ADAPTER	The base system virtual address of adapter address space; 0 if there are no adapters.
ADP$A_DEVICE	The base physical address of the device's control/status register (CSR), loaded from the SCR$L_DEVICE field of the system configuration record (SCR) that was passed to KER$CREATE_ADAPTER.
ADP$A_PAGEMAP	The base system virtual address of adapter page map registers, used to map VAX memory to UNIBUS or Q-bus memory addresses.
ADP$A_DATAPATH	The base system virtual address of adapter datapath registers, used by UNIBUS DMA devices.
ADP$A_IOPAGE	The base system virtual address of Q-bus or UNIBUS I/O space in system memory. For a DWBUA bus adapter on a VAXBI, the field contains the UNIBUS I/O space address; the field is 0 for VAXBI device adapters.
ADP$B_BI_NUMBER	The VAXBI bus number, copied from the SCR$B_BI_NUMBER field of the SCR passed to KER$CREATE_ADAPTER. For a VAXBI-based VAX system, the field represents the number of the VAXBI bus on which the device or DWBUA adapter is a node. The field is 0 for a single-VAXBI system or a non-VAXBI system.
ADP$B_ADAPTER_NUMBER	The VAXBI node number or UNIBUS adapter number. For a VAXBI-based VAX system, the field represents the node number of the device or DWBUA adapter node on its VAXBI bus. For a VAX 11-750, the field represents a UNIBUS adapter number. For all other VAX systems, the field is 0.
ADP$A_MAP_BITMAP	The map register allocation bitmap descriptor for allocating map registers on an adapter.

Table B–2 (Cont.): Adapter Control Block Fields

Field	Meaning
ADP$A_PATH_BITMAP	The datapath register allocation bitmap descriptor for allocating datapath registers on an adapter.
ADP$B_TYPE	The adapter type, as supplied by the processor-specific initialization routine that called KER$CREATE_ADAPTER: ADP$K_UNKNOWN, ADPK_QBUS, ADPK_UNIBUS, or ADP$K_VAXBI.
ADP$B_STATE	The adapter state: a value of 1 indicates the adapter is on-line, 0 indicates it is off-line.
ADP$B_PATH_MAP	Datapath register allocation bitmap.

B.3 ARA — Area Object

Purpose:	Describes a job's access to a shared area of memory.
Location:	Address recorded in job's object table.
Allocated from:	System pool.
References:	Figure B–3, Table B–3.

Figure B–3: Structure of an Area Object

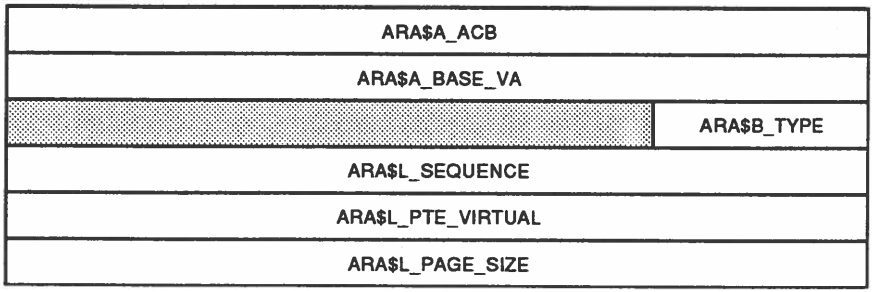


Table B–3: Area Fields

Field	Meaning
ARA$A_ACB	The system virtual address of the area control block that defines the area.
ARA$A_BASE_VA	The explicit P0 virtual address at which the area has been mapped in the job's address space. This field is used only if the creator of the area specifies an explicit virtual address for the area buffer. This value must match the value of ACB$L_NONPIC_VA in the associated ACB.
ARA$B_TYPE	The object type: OBJ$K_AREA.
ARA$L_SEQUENCE	The object sequence number. This value must match the sequence field in the object identifier during object translation.
ARA$L_PTE_VIRTUAL	The system virtual address of the first P0 page page table entry used to map the area buffer into the job's address space. This value is used to unmap the area buffer when the job deletes the area.
ARA$L_PAGE_SIZE	The number of pages of the area buffer mapped into the job's address space. This value is used to unmap the area buffer when the job deletes the area.

B.4 BMP — Allocation Bitmap Descriptor

Purpose:	Describes the size, location, and state of an allocation bitmap.
Location:	Resides in kernel data block (PFN bitmap descriptor) and system space (P0 and P1 page table slot bitmap desriptors; P0 and P1 virtual memory bitmap desriptors; and communication region bitmap descriptor).
Allocated from:	System pool.
References:	Figure 9–2, Table 9–1.

B.5 DEV — Device Object

Purpose:	Describes the characteristics and state of channel to an I/O device.
Location:	Address recorded in job's object table.
Allocated from:	System pool.
References:	Figure B–4, Table B–4.

Figure B–4: Structure of a Device Object

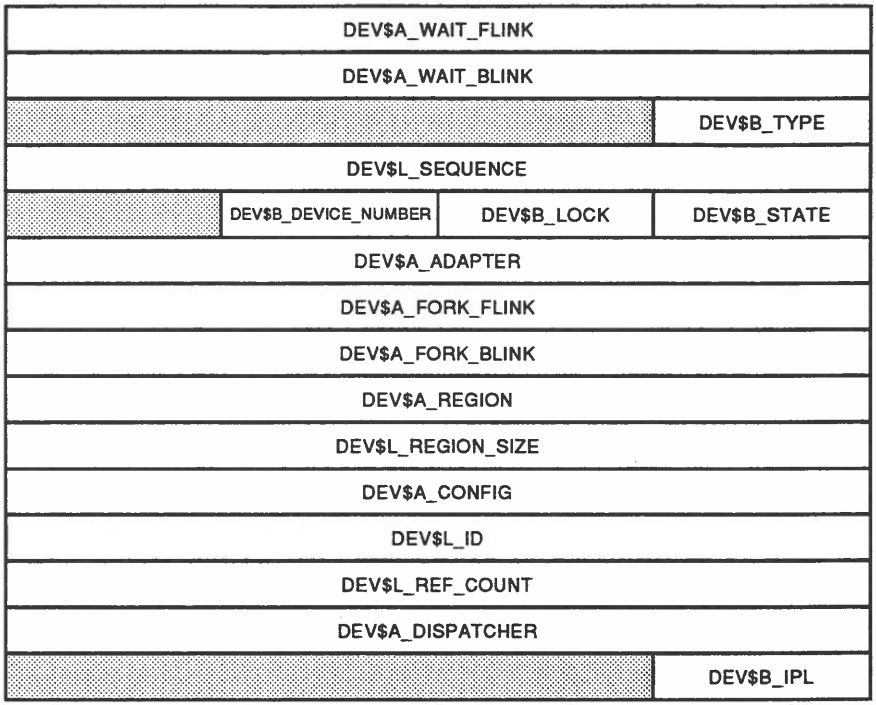


Table B–4: Device Object Fields

Field	Meaning
DEV$A_WAIT_FLINK DEV$A_WAIT_BLINK	The listhead for the device's wait queue — the queue of WCBs representing processes waiting for the device to be signaled by its ISR. When the device is signaled, the IPL 8 ISR that services the device queue scans the queue of WCBs queued to the device and unblocks the first waiting process whose wait conditions are completely satisfied by the device signal.
DEV$B_TYPE	The structure type: OBJ$K_DEVICE.
DEV$L_SEQUENCE	The object sequence number, generated by the internal routine KER$ALLOCATE_OBJECT that allocates the device object.
DEV$B_STATE	The current device state. If the low bit is set, this device object has been signaled by an ISR. The bit is set by the IPL 8 device-signal ISR when it services the device. The bit is examined by routines that test for satisfaction of wait conditions. If the wait of a process in this device's wait queue is completely satisfied by the device signal, the process is unblocked and the bit is cleared. (A device signal unblocks at most one process.)
DEV$B_LOCK	The device object interlock. If the low bit is set, a device interrupt is pending for the device. If a device has an interrupt pending, it means the device cannot be deleted and, if the device is being signaled again by the ISR, the kernel does not have to reenter the device object in the device signal queue or raise the device-signal software interrupt. The bit is set, interlocking the device object, when the ISR calls the KER$SIGNAL_DEVICE procedure to signal the device, and cleared when the IPL 8 ISR removes the device object from the system's device queue.
DEV$B_DEVICE_NUMBER	The relative device number. This is the number of the device relative to others created in the same KER$CREATE_DEVICE procedure call. Devices are numbered upward from 0. The device number is used as an index to access device information stored in arrays.
DEV$A_ADAPTER	The system virtual address of the device's adapter control block (ADP).

Table B–4 (Cont.): Device Object Fields

Field	Meaning
DEV$A_FORK_FLINK DEV$A_FORK_BLINK	The forward and backward links to the next and previous device objects in the system's device signal queue, located at KER$GQ_DEVICE_LIST. When an ISR calls the KER$SIGNAL_DEVICE procedure to signal a device, the kernel inserts the device object into the device queue to await servicing.
DEV$A_REGION	The system virtual address of the device communication region, created by the KER$CREATE_DEVICE procedure. Its address is returned to the caller of KER$CREATE_DEVICE. This address is also placed in an interrupt dispatch block (IDB) field, IDB$A_REGION, when the KER$CREATE_DEVICE procedure sets up vectoring of the device's interrupts. The device communication region address will be passed as an argument to the device's ISR when a device interrupt is dispatched.
DEV$L_REGION_SIZE	The allocated size, in pages, of the device communication region. The size in bytes specified by the KER$CREATE_DEVICE caller is converted to a page count, rounded up to the next page if necessary, and used to allocate and free the region. The value is also used to test shared-device creations for compatible memory requirements.
DEV$A_CONFIG	The system virtual address of the device's system configuration record (SCR), containing the System Builder parameters for the device.
DEV$L_ID	The device object identifier, generated by the internal routine KER$ALLOCATE_OBJECT (in the ALLOCATE module) and returned to the caller of KER$CREATE_DEVICE.

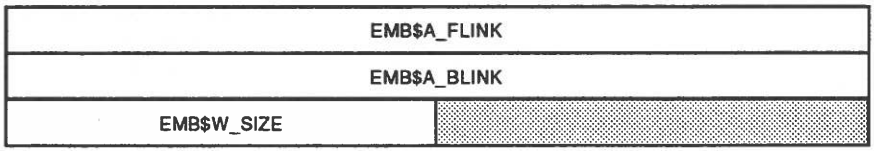
Table B–4 (Cont.): Device Object Fields

Field	Meaning
DEV$L_REF_COUNT	The device object's reference count. This is the count of the number of jobs that have created the device — greater than one only in the case of a shared device. The count is incremented for each job that successfully creates the device and decremented for each job that deletes the device. When the count reaches 0, the device is removed from the IDB's device list and the device object's pool block is returned to the system.
DEV$A_DISPATCHER	The system virtual address of the device's interrupt dispatcher block (IDB), used when removing a deleted device object from the IDB's device list.
DEV$B_IPL	The device's hardware IPL in the decimal range 20 (low) to 23 (high), copied from the device's SCR. The device hardware IPL is 16 greater than the device bus-request priority specified in the System Builder device description. If the KER$CREATE_DEVICE caller specified a location to return the value, the value is placed there.

B.6 EMB — Error-Logging Message Buffer Header

Purpose:	Links an error message buffer into EMB queues.
Location:	Resides at head of error message buffers linked into global EMB queue at KER$GQ_EMB_POSTED or KER$GQ_EMB_AVAIL
Allocated from:	Error-log buffer in queue located at KER$GQ_EMB_AVAIL.
References:	Figure B–5, Table 7–1.

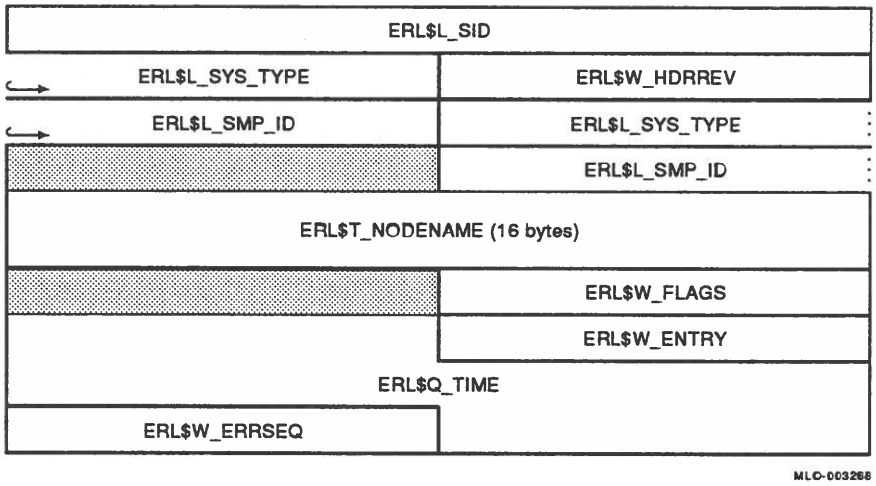
Figure B–5: Structure of an Error-Logging Message Buffer Header



B.7 ERL — EMB Record Header

Purpose:	Records information in the error-log buffer for use by the VMS Error Log Utility.
Location:	Resides in error-log buffer after EMB record header.
Allocated from:	Error-log buffer in queue located at KER$GQ_EMB_AVAIL.
References:	Figure B–6, Table 7–2.

Figure B–6: Structure of an EMB Record Header



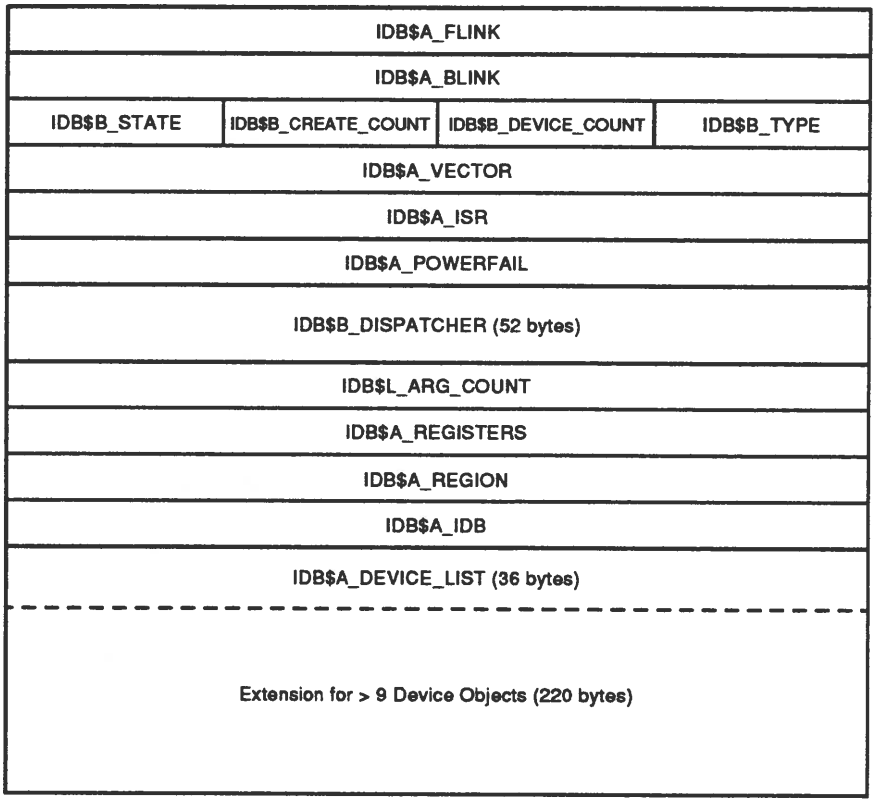
B.8 EVT — Event Object

Purpose:	Describes the characteristic and state of a user-defined event.
Location:	Address recorded in job's object table.
Allocated from:	System pool.
References:	Figure 11–3, Table 11–3.

B.9 IDB — Interrupt Dispatch Block

Purpose:	Describes the charactertics and state of an I/O channel and connects a device object, an interrupt service routine, and a device communication region.
Location:	Linked into global IDB queue located at KER$GQ_IDB_LIST.
Allocated from:	System pool (small IDB) or one page of the communication region (large IDB).
Notes:	There are two forms of IDB: • Small IDB, for 9 or fewer device objects • Large IDB, for more than 9 device objects
References:	Figure B–7, Table B–5.

Figure B–7: Structure of an Interrupt Dispatch Block



MLO-003259

Table B–5: Interrupt Dispatch Block Fields

Field	Meaning
IDB$A_FLINK IDB$A_BLINK	The forward and backward links to next and previous IDB in the system's list of IDBs; the listhead is located at KER$GQ_IDB_LIST. When a job attempts to create a sharable device, the list of IDBs is walked to locate a matching IDB with a matching transfer address.
IDB$B_TYPE	The structure type: OBJ$K_INTERRUPT.
IDB$B_DEVICE_COUNT	The device object count, representing the number of device objects remaining among those initially created by the KER$CREATE_DEVICE call. The count is initialized to the count of devices created and then decreased by 1 for each device deleted. If any device objects have been deleted, requests by jobs to share this device are disallowed. When the IDB's device count reaches zero, the ISR is disconnected from the interrupt dispatcher, the device communication region memory is deallocated, the interrupt dispatcher is removed from the list, and the IDB memory is deallocated.
IDB$B_CREATE_COUNT	The created device count, recording the number of device objects, up to 64, created for this device. If more than 9 device objects are requested, the IDB is allocated in a 512-byte system region page rather than a 128-byte pool block. The value of this field is checked to determine whether the IDB was allocated in its short or long form. Also, this value is used to check that any jobs attempting to create this device for shared access specify the identical number of device objects.
IDB$B_STATE	The IDB state, including the following fields: IDB$V_STATE_DEVSHARED — Bit <24> is set if the device is sharable by more than one job and clear if the device is not sharable. IDB$V_STATE_DEVLOCK — Bit <25>, the device interrupt lock, is set if the device is locked and clear if it is unlocked.
IDB$A_VECTOR	The system virtual address of the device's interrupt vector in the SCB.

Table B–5 (Cont.): Interrupt Dispatch Block Fields

Field	Meaning
IDB$A_ISR	The system virtual address of the unexpected-event dispatcher vector for this device vector. When the device is created, this field is used to store the displaced address of the unexpected-event dispatcher for this SCB vector. When the device is deleted, the unexpected-event dispatcher address is restored to the SCB vector from this field.
IDB$A_POWERFAIL	The address of the caller-specified powerfail recovery routine, if any was specified. In the event of a power failure, the KER$RESTART routine in the POWERFAIL module calls the specified routine.
IDB$B_DISPATCHER	The code that dispatches control to the device's ISR when an interrupt occurs.
IDB$L_ARG_COUNT	The ISR argument count (3). This field is the start of a standard VAX argument list passed to the ISR when it is called with the CALLG instruction. The next three fields in the IDB comprise the arguments in the list.
IDB$A_REGISTERS	The system virtual address of the device's CSRs. This is the first argument passed to the ISR.

Table B–5 (Cont.): Interrupt Dispatch Block Fields

Field	Meaning
IDB$A_REGION	The system virtual address of the device communication region, allocated by the KER$CREATE_DEVICE procedure. This is the second argument passed to the ISR.
IDB$A_IDB	The system virtual address of this IDB. This is the third argument passed to the ISR. This argument is not used by the ISR itself; rather, it is used implicitly by the VAXELN Pascal compiler to pass the address of the IDB when it generates the JSB call to the KER$SIGNAL_DEVICE_R2 subroutine.
IDB$A_DEVICE_LIST	An array of longwords containing the addresses of the device objects. The number of longwords in this field is determined by the number of device objects created by the call to KER$CREATE_DEVICE. If nine or fewer device objects are created, the IDB is allocated in a pool block, and the IDB$A_DEVICE_LIST field is 36 bytes long. If more than nine device objects are created, the IDB is allocated in a page from the communication region, and this field is 256 bytes (64 longwords) long. The address of a device object is obtained by using the relative device number, from field DEV$B_DEVICE_NUMBER in the device object, as a longword index into this array.

B.10 KSD — Kernel Section Descriptor

Purpose: Describes the location and virtual memory requirements of an program or shareable image section.

Location: Linked to program descriptor list at PRG$L_KSD or to shareable image descriptor at SHT$L_KSD.

Allocated from: System image, within the program list or shareable image table.

Notes: There are three forms of KSD:

- Private KSD, for describing executable image sections
- Shareable KSD, for describing shareable image sections
- Global KSD, for describing the location of a writeable shareable image's KSDs

References: Figure 2–5 and Table 2–5 (private KSD); Figure 2–6, Figure 2–7, and Table 2–8 (shareable and global KSDs).

B.11 MSG — Message Object

Purpose:	Describes the characteristics and state of a region of memory mapped between jobs on a local node.
Location:	Linked to a port object when transmitted with the KER$SEND procedure.
Allocated from:	System pool.
References:	Figure B–8, Table B–6.

Figure B–8: Structure of a Message Object

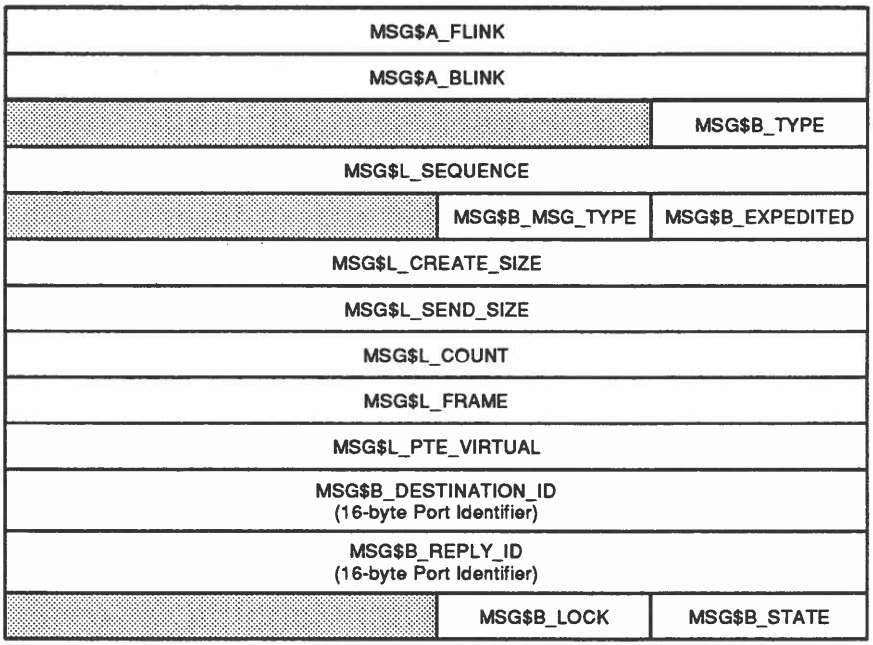


Table B–6: Message Fields

Field	Meaning
MSG$A_FLINK MGS$A_BLINK	The links for inserting the message into a port's message queue.
MSG$B_TYPE	The object type: OBJ$K_MESSAGE.
MSG$L_SEQUENCE	The object sequence number. This value must match the sequence field in the object identifier during object translation.
MSG$B_EXPEDITED	A Boolean flag that is set when the message is to delivered to a ports expedited message queue.
MSG$B_MSG_TYPE	Field used in message passing between KA800 processors in a closely coupled symmetric multiprocessing system.
MSG$L_CREATE_SIZE	The size in bytes of the messasge buffer as requested by the caller of KER$CREATE_MESSAGE.
MSG$L_SEND_SIZE	The number of bytes of the message buffer the caller of KER$SEND wishes to send. The field is initialized to the value of MSG$L_CREATE_SIZE.
MSG$L_COUNT	The number of physical and virtual page occupied by the message buffer. This value is used to map the message buffer into a receiving job's address space and to unmap the message buffer when the job sends or deletes the message.
MSG$L_FRAME	The starting page frame number for the physical memory occupied by the message buffer. This value is used for mapping the message buffer into the P0 virtual address space of a job that creates or receives the message.
MSG$L_PTE_VIRTUAL	The system virtual address of the first P0 page page table entry used to map the message buffer into a receiving job's address space. This value is also used to unmap the message buffer when the message is sent or delete.
MSG$B_DESTINATION_ID	The identifier of the port to receive the message. This field is set by the KER$SEND procedure.
MSG$B_REPLY_ID	The identifier of the port to which the receiver of the message can reply. This field is set if the caller of KER$SEND supplies a reply port argument.
MSG$B_STATE MSG$B_LOCK	Bit fields used to synchronized message passing between KA800 processors in a closely coupled symmetric multiprocessing system.

B.12 NAM — Name Object

Purpose:	Associates a character string with a port.
Location:	Linked to a local name table listhead; global listhead desriptor resides at KER$GR_LOCAL_NAME and points to the first of 128 table listheads. The listhead is selected by indexing the table with the value derived by hashing the name string.
Allocated from:	System pool.
References:	Figure B–9, Table B–7.

Figure B–9: Structure of a Name Object

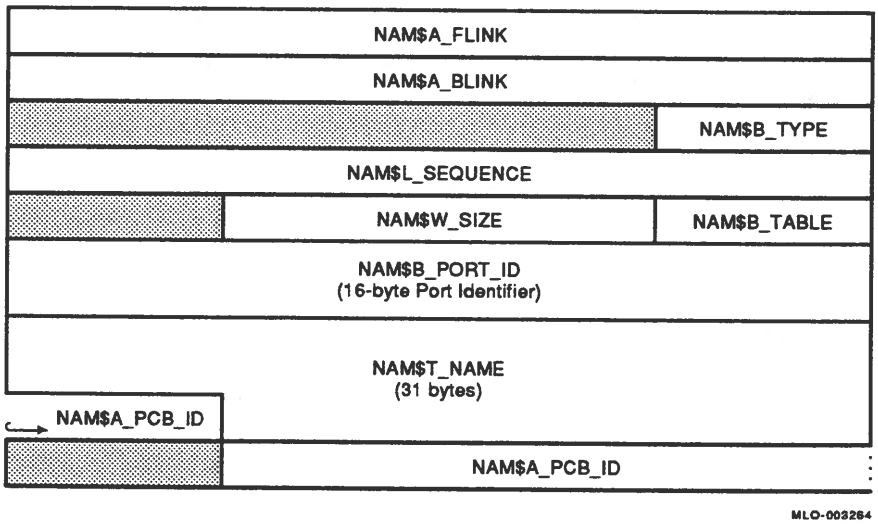


Table B–7: Name Fields

Field	Meaning
NAM$A_FLINK NAM$A_BLINK	The links for inserting this name into the kernel's local name table.
NAM$B_TYPE	The object type: OBJ$K_NAME.
NAM$L_SEQUENCE	The object sequence number. This value must match the sequence field in the object identifier during object translation.
NAM$B_TABLE	A flag to indicate in which name table the name is entered: 0 for the local table, 1 for the universal table.
NAM$W_SIZE	The size in bytes of the port's name.
NAM$B_PORT_ID	The port identifier for the named port.
NAM$T_NAME	The ASCII string containing the uppercase name of the port. The maximum size of the string is 31 characters.
NAM$A_PCB_ID	The identifier for a named process object. This field is not used for naming port objects.

B.13 NETCON — Network Connection Message

Purpose:	Transmits data required by the kernel and the Network Service to link two ports into a circuit.
Location:	A message buffer mapped into a sending or receiving job's P0 address space.
Allocated from:	A message object.
References:	Figure B–10, Table B–8.

Figure B–10: Structure of a Network Connection Messsage

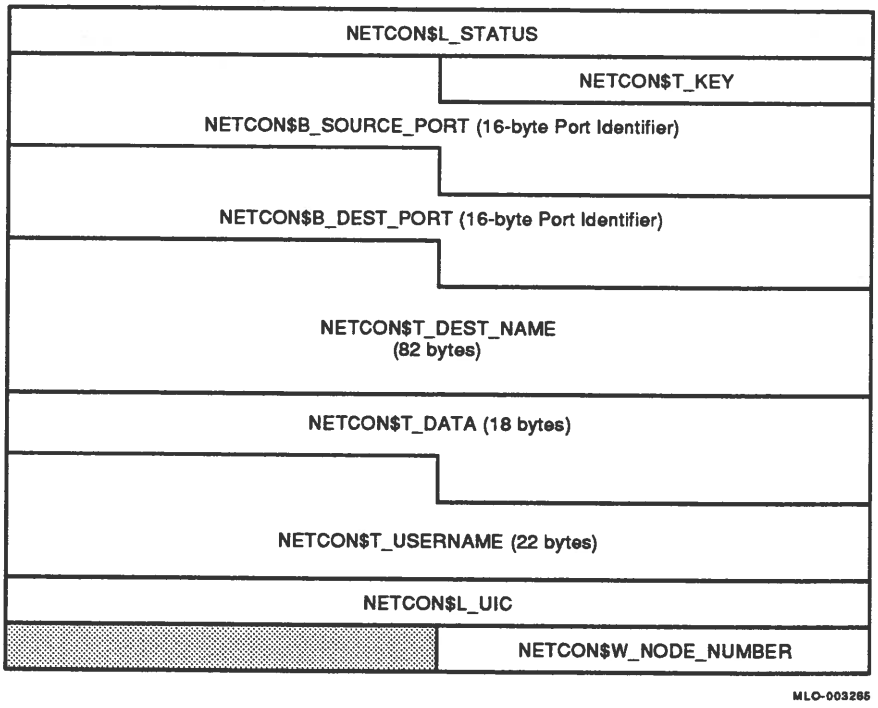


Table B–8: Network Connection Message Fields

Field	Meaning
NETCON$L_STATUS	The status of the network connection operation.
NETCON$T_KEY	The purpose of the request. This field can contain the ASCII string "CI" for a connection-initiation message or "CC" for connection confirmation message. The former is sent by the KER$CONNECT_CIRCUIT procedure; the latter is sent by the KER$ACCEPT_CIRCUIT procedure.
NETCON$B_SOURCE_PORT	The identifier for the port to which the circuit will be or has been connected.
NETCON$B_DEST_PORT	The identifier for the partner port in the circuit.
NETCON$T_DEST_NAME	The string descriptor for the partner's port name. The first word in the field contains the size of the name string; the remaining 80 bytes contain the string.
NETCON$T_DATA	The string descriptor for the accept or connect data specified in the call to KER$CONNECT_CIRCUIT or KER$ACCEPT_CIRCUIT. The first word in the field contains the size of the string; the remaining 16 bytes contain the string.
NETCON$T_USERNAME	The string descriptor for the username of the owner of the partner port in the circuit. The first word in the field contains the size of the string; the remaining 20 bytes contain the string.
NETCON$L_UIC	The user identification code of the owner of the partner port in the circuit.
NETCON$W_NODE_NUMBER	The DECnet node number of a remote partner port in the circuit.

B.14 NS — Name Service Request Message

Purpose:	Provides information required by the Name Service to create a universal name for a port.
Location:	A message buffer mapped into a sending or receiving job's P0 address space.
Allocated from:	A message object.
References:	Figure B–11, Table B–9.

Figure B–11: Structure of Name Service Request Messsage

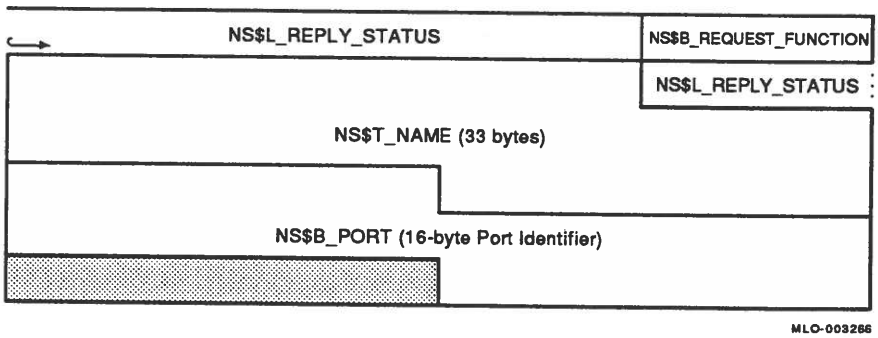


Table B–9: Name Service Request Message Fields

Field	Meaning
NS$B_REQUEST_FUNCTION	The name service function requested by the message: create a name (0), delete a name (1), translate a name (2).
NS$L_REPLY_STATUS	The status returned from the name service.

Table B–9 (Cont.): Name Service Request Message Fields

Field	Meaning
NS$T_NAME	The string descriptor for the name to created or translated. The first word in the field contains the size of the name string; the remaining 31 bytes contain the string.
NS$B_PORT	The identifier for the port to be named or translated.

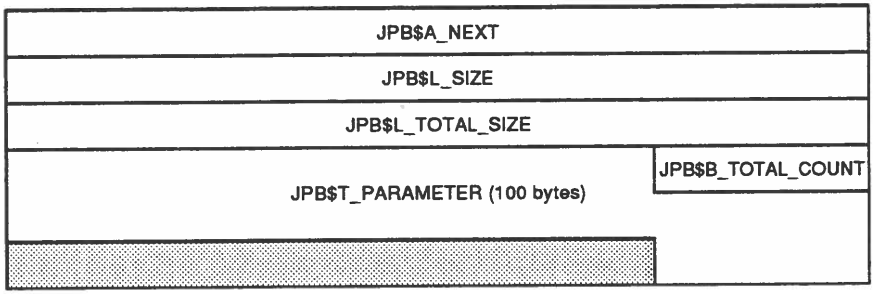
B.15 JCB — Job Control Block

Purpose:	Describes the characteristic and state of a VAXELN job.
Location:	Linked into global job queue at KER$GQ_SYSTEM_JOB.
Allocated from:	One page of the communication region.
References:	Figure 4–2, Table 4–1.

B.16 JPB — Job Parameter Block

Purpose:	Holds the job arguments passed through the Program Description Menu or a call to KER$CREATE_JOB.
Location:	Linked into list of parameter blocks in the program descriptor pointed to by PRG$W_PARAMETER; at job creation, linked into list of parameter blocks pointed to by JCB$A_PARAMETER_LIST.
Allocated from:	System pool.
Notes:	Parameter blocks reside temporarily in pool blocks during job creation. Before normal job execution begins, the parameter desriptors are copied to a table in the job's P0 address space and their pool blocks are freed. The argument table first contains a standard VAX argument list containing the argument count and the addresses of the argument descriptors; this argument list is passed to the job when it is called. Following the argument list are the descriptors themselves.
References:	Figure B–12, Table 2–4.

Figure B–12: Structure of a Job Parameter Block



B.17 PCB — Process Control Block

Purpose:	Describes the characteritics and state of a process's software context.
Location:	Linked to the job's process queue at listhead JBC$A_PROCESS_FLINK/JCB$A_PROCESS_BLINK.
Allocated from:	One page of the communication region, shared with the PTX.
References:	Figure 4–3, Table 4–2.

B.18 PRT — Port Object

Purpose:	Describes the charactertistics and state of a message queue and/or circuit connection.

Location:	Address recorded in the job's object table; linked into job's port queue with listhead at JCBA_PORT_FLINK/JCBA_PORT_BLINK.
Allocated from:	System pool.
References:	Figure B–13, Table B–10.

Figure B–13: Structure of a Port Object

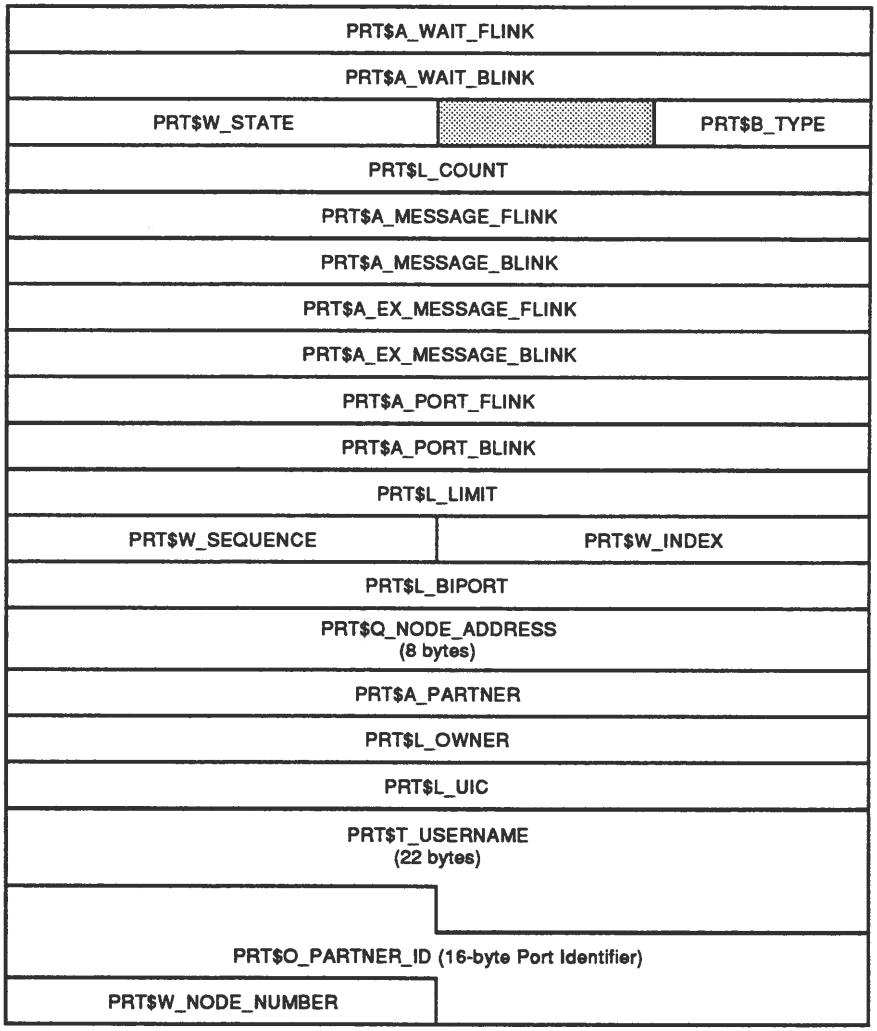


Table B–10: Port Fields

Field	Meaning
PRT$A_WAIT_FLINK PRT$A_WAIT_BLINK	The listhead for the queue of processes waiting for a message to arrive on this port. A WCB representing a waiting process is inserted into this queue by the KER$WAIT procedure. When a message arrives, the processes whose waits are completely satisfied are removed from the queue and unblocked.
PRT$B_TYPE	The object type: OBJ$K_PORT.
PRT$W_STATE	A bit field recording the state of the port in circuit-connection and message-sending operations. When set, the bits in the field have following meanings:

Bit	Meaning
PRT$V_CIRCUIT_CONNECTED	The port is connected in a circuit.
PRT$V_CONNECT_PENDING	A circuit connection is pending for the port.
PRT$V_PARTNER_DISCONNECT	The port's circuit partner has disconnected the circuit.
PRT$V_FULL_FAILURE	The partner port in a circuit has reached its message limit.
PRT$V_FULL_ERROR	Return an error status on a send if the partner port is at its message limit.
PRT$V_FULL_WAIT	A send from this port over a circuit has blocked until the partner's port is no longer full.

PRT$L_COUNT	The number of messages queue to the port.
PRT$A_MESSAGE_FLINK PRT$A_MESSAGE_BLINK	The listhead for the queue of messages sent to this port.
PRT$A_EX_MESSAGE_FLINK PRT$A_EX_MESSAGE_BLINK	The listhead for the queue of expedited messages sent to this port. At most one expedited message can be queued here. When a message is received from the port, an expedited message is dequeued before a normal message.

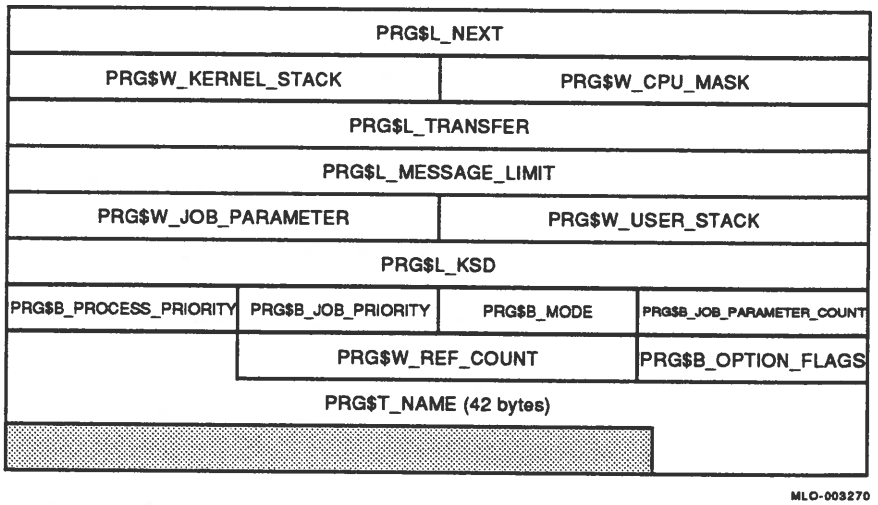
Table B–10 (Cont.): Port Fields

Field	Meaning
PRT$A_PORT_FLINK PRT$A_PORT_BLINK	The links for inserting the port into the job's queue of ports, located at JCB$A_PORT_FLINK.
PRT$L_LIMIT	The maximum number of messages that may be queued to the port.
PRT$W_INDEX	The longword offset into the port address table to the address of this port. This field corresponds to the index field in the port identifier for this port.
PRT$W_SEQUENCE	The generation number of the port in its port address table entry; used to detect mismatches between an identifier and a port object. This field corresponds to the sequence field in the port identifier for this port.
PRT$L_BIPORT	VAXBI node information for ports created by KA800 processors in a closely coupled symmetric multiprocessing system. This field corresponds to the second longword in the port identifier for this port.
PRT$Q_NODE_ADDRESS	The DECnet node address or the Ethernet hardware address of the node on which the port was created. This field corresponds to the node address field in the identifier for this port.
PRT$A_PARTNER	The address of port object to which this port is connected in a circuit. For a remote circuit, this is the address of port owned by the Network Service.
PRT$A_OWNER	The address of the job control block of the job that created the port.
PRT$L_UIC	The user identification code for the owner of the partner port in a circuit.
PRT$T_USERNAME	The string descriptor for the username of the owner of the partner port in the circuit. The first word in the field contains the size of the string; the remaining 20 bytes contain the string.
PRT$O_PARTNER_ID	The object identifier of the partner port in a circuit.
PRT$W_NODE_NUMBER	The DECnet node address of the node on which the parter port in a circuit resides.

B.19 PRG — Program Descriptor

Purpose:	Records information entered on the Program Characteristics Menu or supplied implicitly by the System Builder for system programs.
Location:	Linked into the program list with listhead at KER$GA_PROGRAM in the kernel parameter block; at system initialization, the listhead is transferred to KER$GA_PROGRAM_LIST in the kernel data block.
Allocated from:	System image, within the program list.
References:	Figure B–14, Table 2–3.

Figure B–14: Structure of a Program Descriptor



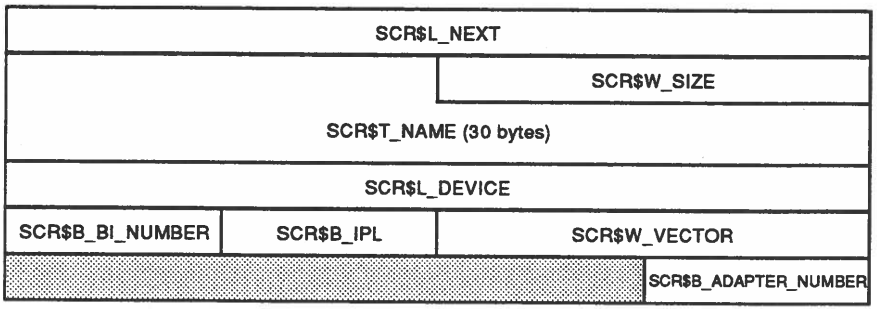
B.20 PTX — Process Hardware Context Block

Purpose:	Contains the hardware context of a process while it is not executing.
Location:	Address is recorded in PCB$A_PTX.
Allocated from:	One page of the communication region, shared with the PCB.
References:	Figure 4–4, Table 4–3

B.21 SCR — System Configuration Record

Purpose:	Records information entered on the Device Desription Menu or supplied implicitly by the System Builder for integrated device controllers.
Location:	Linked into the device list in system image at listhead pointed to by KER$GA_DEVICE_LIST.
Allocated from:	System image, within the device list.
References:	Figure B–15, Table 2–6.

Figure B–15: Structure of a System Configuration Record



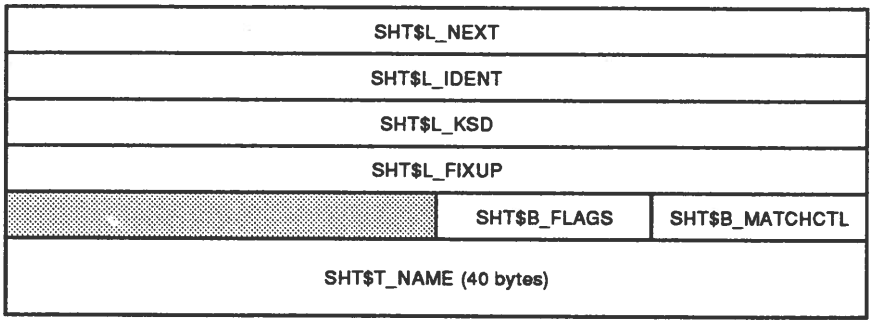
B.22 SEM — Semaphore Object

Purpose:	Records the characteristic and state of a gate to control access to a shared resource.
Location:	Address recorded in job's object table.
Allocated from:	System pool.
References:	Figure 11–4, Table 11–4.

B.23 SHT — Shareable Image Descriptor

Purpose:	Records information about the shareable images included in a system image; used by the Program Loader to resolve a dynamic program's references to shareable images.
Location:	Linked into the shareable image table with listhead at KER$GA_SHARE_LIST in the kernel parameter block; at system initialization, the listhead is transferred to KER$GA_SHAREABLE_IMAGE_LIST in the kernel data block.
Allocated from:	System image, within the shareable image table.
References:	Figure B–16, Table 2–7.

Figure B–16: Structure of a Shareable Image Table Entry



B.24 WCB — Wait Control Block

Purpose:	Records the characteristics and state of a process's wait for a kernel object or timeout.
Location:	Statically linked to PCB at PBC$B_WCB + WCB$A_LIST; dynamically linked to PCB at PCB$A_FIRST_WCB. Also linked dynamically to OBJA_WAIT_FLINK/OBJA_WAIT_BLINK fields of the kernel object being waited for. Timer WCBs are linked dynamically into the global timer queue with listhead at KER$GQ_TIMER_QUEUE.
Allocated from:	Timer WCB allocated within PCB; normal WCBs allocated four per system pool block.
Notes:	There are two forms of WCB: • Timer WCB, for waiting with a timeout • Normal WCB, for waiting for a kernel object
References:	Figure 11–1, Table 11–1.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

P

Q

R

S

T

U

V

W

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Alaska, Hawaii, or New Hampshire	603–884–6660	Digital Equipment Corporation P.O. Box CS2008 Nashua NH 03061
Rest of U.S.A. and Puerto Rico[1]	800–DIGITAL	
Canada	800–267–6219 (for software documentation) 613–592–5111 (for hardware documentation)	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: Direct Order Desk
Internal orders (for software documentation)	—	Software Supply Business (SSB) Digital Equipment Corporation Westminster MA 01473
Internal orders (for hardware documentation)	DTN: 234–4323 508–351–4323	Publishing & Circulation Services (P&CS) NRO3–1/W3 Digital Equipment Corporation Northboro MA 01532

[1]Prepaid orders from Puerto Rico, call Digital's local subsidiary (809–754–7575)

Reader's Comments

VAXELN Internals Manual
AA–NC72A–TE

Your comments and suggestions will help us improve the quality of our future documentation. Please note that this form is for comments on documentation only.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (product works as described)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What I like best about this manual: _____

_____

What I like least about this manual: _____

_____

My additional comments or suggestions for improving this manual:

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Please indicate the type of user/reader that you most nearly represent:

☐ Administrative Support
☐ Computer Operator
☐ Educator/Trainer
☐ Programmer/Analyst
☐ Sales
☐ Scientist/Engineer
☐ Software Support
☐ System Manager
☐ Other (please specify) _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

10/87

— 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
PKO3-1/30D
129 PARKER STREET
MAYNARD, MA 01754-2198

— — Do Not Tear — Fold Here — — — — — — — — — — — — — — — — — —

Cut Along Dotted Line