AEGIS Internals and Data Structures

Order No. N/A
Revision 00
Software Release 9.0

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

Printed in U.S.A.

First Printing: January 1, 1986

This document was produced using the SCRIBE document preparation system. (SCRIBE is a registered trademark of Unilogic, Ltd.)

Apollo Computer Inc. reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should in all cases consult Apollo Computer Inc. to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF APOLLO COMPUTER INC. HARDWARE PRODUCTS AND THE LICENSING OF APOLLO COMPUTER INC. SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN APOLLO COMPUTER INC. AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY APOLLO COMPUTER INC. FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY BY APOLLO COMPUTER INC. WHATSOEVER.

IN NO EVENT SHALL APOLLO COMPUTER INC. BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATING TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF APOLLO COMPUTER INC. HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGES.

THE SOFTWARE PROGRAMS DESCRIBED IN THIS DOCUMENT ARE CONFIDENTIAL INFORMATION AND PROPRIETARY PRODUCTS OF APOLLO COMPUTER INC. OR ITS LICENSORS.

Preface

The *AEGIS Internals and Data Structures* manual describes the algorithms and data structures that comprise the AEGIS operating system kernel, Software Release 9.0. It includes detailed descriptions of the following kernel services:

- Memory management
- Local and remote file systems
- Process management
- Network management
- Naming interface
- System initialization

This manual intends to describe only the services provided by the AEGIS kernel; subsequent internals documents will describe the user-mode AEGIS functions.

Audience

This manual is intended for new and existing employees of Apollo's Research and Development group who need a detailed explanation of AEGIS operating system architecture. The manual is also intended for selected OEM customers with source code licenses who sign or have signed a non-disclosure agreement with Apollo Computer.

Organization of this Manual

This manual contains 29 chapters that are organized into six sections.

- Section I (Chapters 1 and 2) contains two introductory chapters that describe the philosophy behind AEGIS system design and provide an overview of AEGIS system components, both kernel and user-mode.
- Section II (Chapters 3 through 8) describes the components of the object storage system, the naming interface, and the object locating service.
- Section III (Chapters 9 through 13) describes the virtual memory management system.
- Section IV (Chapters 14 through 19) describes the process environment. It contains chapters that describe level 1 and level 2 processes, eventcounts, fault handling, and SVC dispatching.
- Section V (Chapters 20 through 24) describes the hardware and software components of the network interface.
- Section VI (Chapters 25 through 29) describe the system initialization procedure.

The manual also contains two appendixes.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following conventions:

UPPERCASE — Uppercase words or characters in formats and command descriptions represent commands or keywords that you must use literally.

lowercase — Lowercase words or characters in formats and command descriptions represent values that you must supply.

[ ] — Square brackets enclose optional items in formats and command descriptions. In sample Pascal statements, square beackets assume their Pascal meanings.

{ } — Braces enclose a list from which you must choose an item in format and command descriptions. In simple Pascal statements, braces assume their Pascal meanings.

| — A vertical bar separates items in a list of choices.

< > — Angle brackets enclose the name of a key on the keyboard.

CTRL/Z — The notation CTRL/ followed by the name of a key indicates a control character sequence. You should hold down the <CTRL> key while typing the character.

... — Horizontal ellipsis points indicate that the preceding item can be repeated one or more times.

. — Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.

Suggested Reading Paths

Although this manual discusses some aspects of DOMAIN system hardware as they relate to DOMAIN system software, it does not provide a complete description of processor, network, or display hardware implementations. For such information, refer to the Apollo publications listed below:

- The *DOMAIN System MULTIBUS Reference* document, which describes Apollo Computer's implementation of the Intel MULTIBUS.

- The *Engineering Handbook*, (*for internal use only*), which gives brief descriptions of processor, memory, I/O and display hardware implementations and provides diagrams of control and status registers.

- The *Programming with DOMAIN Advanced System Calls* manual, which documents a set of unreleased system services, such as the file utility (FU) and the command line handler (CL).

- The *Extending Your DOMAIN Streams* manual, which describes the extensible streams interface (IOS).

The *Writing Device Drivers with GPIO Calls* manual, which describes how to use the general-purpose I/O system services to write device drivers for customer-supplied devices.

- The *DOMAIN Assembler Reference* manual, which describes the DOMAIN assembler and gives information about object module format.

For information on processor hardware, refer to the following Motorola publications:

- The Motorola MC68020 32-Bit Micropro essor User's Manual. Prentice-Hall, Inc. 1984.

- The Motorola 16-Bit Microprocessor User's Manual, Third Edition. Prentice-Hall, Inc. 1982.

Contents

Illustrations

Tables

Chapter 1
AEGIS System Design

It may be that the whims of chance are really the importunities of design.
But if there is a Design, it aims to look natural and fortuitous;
that is how it gets us into its web.

Mary McCarthy, On the Contrary

The DOMAIN system is an integrated local area network of personal workstations and server computers designed to meet the computing environment needs of technical professionals. The DOMAIN system intends to provide these technical professionals with a substrate upon which to execute complex scientific and engineering applications by providing:

- The ability to run large, mainframe class application programs tailored to their tasks
- A high user-to-computer bandwidth, where the processor resides close to the display
- A network for cooperation and sharing with others

In order to best achieve these goals, DOMAIN system architecture is based upon AEGIS, an integrated, distributed, object-oriented, local area network operating system that runs on a personal workstation. This chapter discusses the design principles that influence AEGIS operating system architecture.

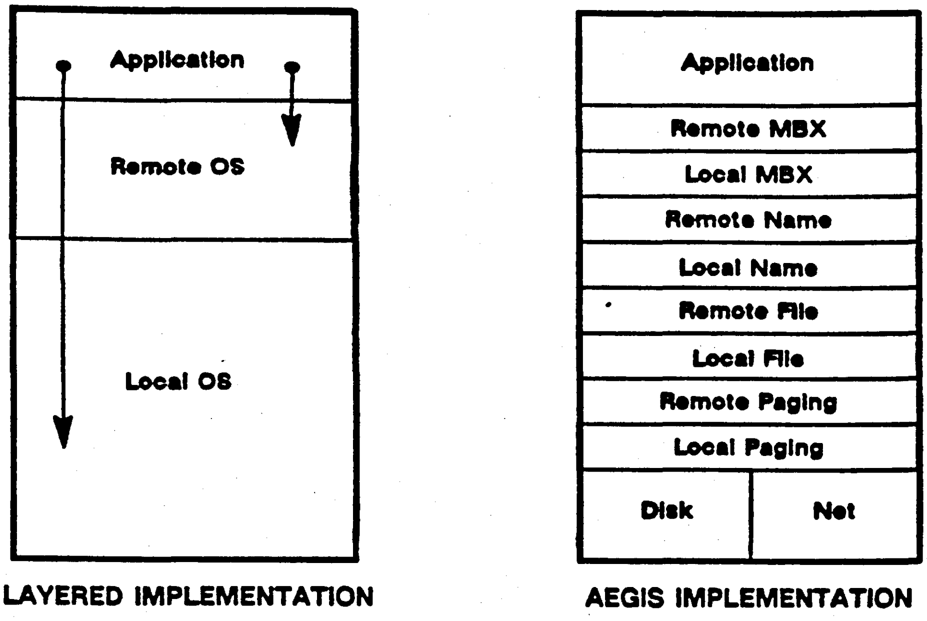
1.1. The Distributed System Design

Some operating systems are designed first with local facilities in mind, and are then made distributed by adding a network layer. In constrast, AEGIS was designed from its inception to provide the facilities that make it a distributed system; conseqently, remote and local operations are tied together in the same module rather than being layered on top of each other. Figure 1-1 illustrates the two styles of distributed system.

A distributed system has certain significant advantages over a centralized system. A distributed system can expand in increments as more workstations are added to the network. This feature produces a high degree of aggregate computing power. In addition, a distributed system has the potential for robustness; a single workstation can fail without hampering the performance of other workstations on the network. However, distributed systems often experience partial failure, whereas centralized systems are either running or completely down. For this reason, much of AEGIS code is devoted to signaling, handling, and recovering from errors.

1.2. The Integrated System Design

An important property of networks and distributed systems is that distinct components are often under different administrative control. As a result, cooperation, protection, and reliability become more complicated.



LAYERED IMPLEMENTATION

AEGIS IMPLEMENTATION

Figure 1-1. Styles of Local/Remote Implementations

There are several types of distributed systems in the industry that provide varying degrees of autonomy and cooperation. Two such examples are:

- The ARPA Internet (ARPAnet) model, a communications facility between autonomous hosts that are separately owned and administered.
- The VAXcluster model, a distributed multi-computer that appears to the user as a single machine.

The ARPAnet provides a high degree of autonomy, as each host has complete control of its users, and limits cooperation to electronic mail and some file transfer. In contrast, the VAXcluster provides a high degree of cooperation and sharing but perpetuates the problems of timesharing systems, such as protection and machine usage allocation.

The Apollo DOMAIN distributed system is integrated in that it balances cooperation and autonomy. It is designed to provide the information sharing of a VAXcluster and the autonomy of the ARPAnet network. To achieve this integrated state, the network architecture must permit users to cooperate if they choose, while simultaneously allowing them to declare autonomy. To allow cooperation, AEGIS system architectures makes access to files transparent and provides a network-wide registry that identifies all users without regard to the machine they use. To provide autonomy, AEGIS maintains a complete set of operating system facilities on each workstation that permits it to run independently of the network. Finally, the system supports access control mechanisms that allow users to decide with whom they wish to cooperate and from whom they desire autonomy.

1.3. Local Area Networking Design

Local area networks offer high bandwidths and low error rates. Consequently, an operating system on a local area network should minimize the processor time required to get messages on and off the network and use simple retransmission techniques instead of high-overhead handshaking protocols and error correction techniques.

To achieve these goals, the AEGIS network architecture is built upon an inexpensive datagram service, with problem-oriented protocols layered on top of the datagram service. The base-level system, called the **kernel**, has no separate virtual circuit, session, or presentation layers like those specified in other network architectures. Instead, each AEGIS system service defines its own "lightweight" protocol tailored to its own special needs; the protocol can take advantage of the service's special characteristics so that it is efficient and fast, and can use simple retransmission techniques for error recovery.

For example, some operations can be repeated back-to-back with identical results, such as a repeated read request. Since duplicate requests of this type pose no problem to network operation, AEGIS network architecture does not contain a mechanism to suppress them, as it would if it subscribed to the network layering model.

1.4. Typed File Design

In all systems, a file system **object** is a named collection of bits. Generally, the program that writes the collection of bits -- the *file* -- has a purpose for the file and a model for the file data's interpretation. For example, text editing programs produce text files; compilers produce object files; file system directories represent another use of files in a file system.

What differs from system to system is not the use of the file system object, but the method used to interpret the bits within any given file. In many systems, the naming convention dictates the interpretation: directories are named name.DIR, FORTRAN source code is named source.FOR and object files are named name.OBJ. Rather than depending on naming conventions, AEGIS file system objects are **typed**; that is, they are stamped with a file type identifier that declares the writer's intention for the file. It is the type identifier, not the object's name, that determines how the file is to be used.

File typing has two advantages. First, it separates the file's type from the human naming convention. More importantly, file typing is *extensible*: new file types can be added to the system at any point in its lifetime, and Apollo engineers need not be the only ones to add them.

For example, a *source history* file type has recently been added to the AEGIS system. A source history file stores compressed source text and the source's complete revision history. When one of the compilers reads the file, its type identifier indicates that it is a source history file; the standard I/O package interprets the file's contents (by expanding and hiding the revision history) so that the compiler recognizes it as a regular text file. The source control facility, however, manipulates the same source history file directly, so that the file's raw contents are available to the facility.

1.5. AEGIS as a Personal Workstation System

Because AEGIS is meant to run on a personal workstation, its design differs from the time-sharing system design. Since all computation on a node is carried out on behalf of a single person, the system requires no protection from intentional interference, but rather protects against accidents. Consequently, much of AEGIS software exists in user space, where it is easier to modify and debug. Resource allocation and accounting mechanisms are also simpler in the workstation environment.

Chapter 2
AEGIS System Overview

This chapter introduces AEGIS system components. They consist of:

- The protected operating system software, called the AEGIS nucleus or **kernel**
- The user program environment, which is composed of:
  - The process manager (PM), which handles local program invocation and execution
  - The software libraries, which provide the environment in which the programs run
  - The server process manager (SPM), which handles remote program invocation and execution
  - Network programs, such as NETMAIN and NETMAN
  - The serial I/O login facility
  - The alarm server
- The user environment -- the collection of programs (referred to as commands) that make up the DOMAIN command line interpreter, called the **shell**.
- The display manager (DM) -- the software process that manages the screen environment of a node's display terminal.

2.1. Interaction of AEGIS Kernel and User Components

The M680x0 processor architecture supports two modes in which software programs can run: unprivileged user mode and privileged supervisor mode. The AEGIS system is divided into services that operate in user mode and services that run in supervisor mode. The operating system services that the AEGIS kernel provides run in supervisor mode. The process manager, software libraries, display manager, and the rest of the user environment run in user mode.

User-mode programs, whether they are AEGIS user-mode system services or user-written application programs, gain access to AEGIS kernel services through the SVC catcher. This procedure takes the user program's call and arguments, changes from user mode to supervisor mode, and then dispatches the call to the appropriate AEGIS kernel service. When the target kernel service completes its operation, it passes control back to the user program. The AEGIS kernel modules that user-mode programs can run are collectively known as the **supervisor**.

2.2. AEGIS Kernel Services

The AEGIS system is designed as a structured set of subsystems called **managers**. Each manager is composed of one or more modules that define the manager's set of operations and its private database. Many of the modules within a particular manager are available to other managers in the system. Consequently, each manager can use its own internal database plus modules in other managers to build a complete set of system services. And, because the system is composed of small, independent modules, a change to a module does not require a change to the entire operating system.

The services provided by the managers of the AEGIS kernel can be separated into the following categories:

- File management
- Process management
- Virtual memory management
- Network management
- I/O management
- Time management
- Access control
- System initialization and shutdown

The next sections introduce the methods that the AEGIS system uses to carry out each of these functions.

2.2.1. File Management

AEGIS file management at the kernel level is, for the most part, object management. At this level, files are abstracted, just as are all other system resources, into objects. In general, file system objects are simply storage containers for bits. The AEGIS managers that handle objects at this level make no attempt to interpret the bits within the object as representations of an object type. It is the system's higher-level managers that interpret the bits.

The AEGIS file system carries out two functions -- **object management** and **object naming**.

2.2.1.1. Object Management

The AEGIS components that carry out object management are collectively known as the **object storage system** (OSS). The object storage system manages the storage of objects on disk and provides the ability to read and write 1024-byte portions, or **pages**, of an object from local or remote disk to main memory. At any given time, the permanent storage for an object resides entirely at only one node, called the **home node**. The object storage system returns the results of any remote modifications to an object back to the home node for permanent storage. In addition, the system does not arbitrarily shift an object's home node from one node to another.

The interpretation of the bits within an object is left to the object's **type manager**. Most of the object type managers exist at the user-space level; an example of a user-space type manager is the stream interface. There are, however, two kernel-space managers that interpret the bits of file system objects: the naming server, which recognizes directory objects, and the ACL manager, which interprets access control list objects.

2.2.1.2. Object Naming

Objects are identified by 64-bit unique identifier strings, or UIDs. When an object is created, the system manufactures it a UID by concatenating the unique node ID of the node generating the object with a time stamp from the node's timer. (The DOMAIN network does not use a global clock; instead, each node keeps its own time.) The UID is the mechanism the system uses to locate the object; that is, it is the system's *internal* name for the object.

The naming server is the AEGIS manager that allows users and programs to refer to objects in the network using text string names instead of UIDs. The naming server on each node manages a collection of directories organized as a network-wide, multilevel tree. The directories contain the associations between text string names and the UIDs of objects local to that node. (By convention, an object is located on the same node as the directory in which it is catalogued.) A user refers to an object by its text string name, or pathname; the naming server's function is to translate this text string name to the object's UID using the directory data structures. Chapter 8 describes naming server function and structure in detail.

2.2.2. Process Management

Process management concerns the allocation of processor resources. The AEGIS kernel manages processor resources by multiplexing the processor into many *virtual* processors, or processes. A **process** is an independent, asynchronously executing entity.

The AEGIS kernel supports two levels of processes:

- Level 1 processes, also called supervisor, kernel, or PROC1 processes

- Level 2 processes, also called user or PROC2 processes

2.2.2.1. Level 1 Processes

Level 1 processes are processes that only run the protected operating system software and thus run *exclusively* in supervisor mode. Level 1 processes are completely internal to the AEGIS kernel: their **context** – processor state and stack – exists in a protected portion of virtual memory. In addition, level 1 process context is permanently **wired**; that is, the process's context is permanently resident in physical memory and can never be paged out by the virtual memory management subsystem. Note, however, that although a level 1 process's context is wired, it can still run pageable AEGIS kernel procedures.

There are 32 level 1 processes; the system names them with small integers (1-32) called process IDs, or PIDs. Process IDs are not unique; when the system deletes a level 1 process, it reissues its PID to the next new level 1 process it creates. (A level 1 process is deleted when the level 2 process on top of it is deleted.) Because PIDs, unlike UIDs, are not unique identifiers, the system can only refer to level 1 processes on a single machine, rather than on a network-wide basis.

2.2.2.2. Level 2 Processes

At system initialization, eight level 1 processes are reserved to the AEGIS kernel. The remaining 24 processes can be used as additional level 1 processes, or they can be augmented, or **bound** to level 2 processes (also called user or PROC2 processes).

Level 2 processes are level 1 processes with additional, user-mode context: their own process virtual address space. For the most part, level 2 proc sses run user-mode software. To run the supervisor-mode AEGIS kernel services, level 2 processes enter supervisor mode via the SVC catcher.

The user-mode context of a level 2 process is **pageable**: the virtual memory management subsystem can move pages of the process's virtual memory in and out of physical memory (unless the process specifically wires its pages.) However, while the level 2 process context is pageable, the level 1 process context underneath it is not. The level 2 process's pageable user-mode context provides the environment for its user-mode activity. The level 1 process context that is bound to every level 2 process supports the level 2 process's supervisor-mode activity.

The system gives a unique name to a level 2 process by assigning it a UID. Consequently, a level 2 process on one machine can explicitly refer to a level 2 process on another machine. Both the supervisor-mode level 2 process (PROC2) manager and the user-mode process manager (PM) handle level 2 process operations. The PROC2 manager handles the binding of level 2 process context to level 1 processes, while the process manager handles user-mode process operations such as program invocation, fault handling, and resource cleanup. Figure 2-1 illustrates the relationship between level 1 and level 2 processes.

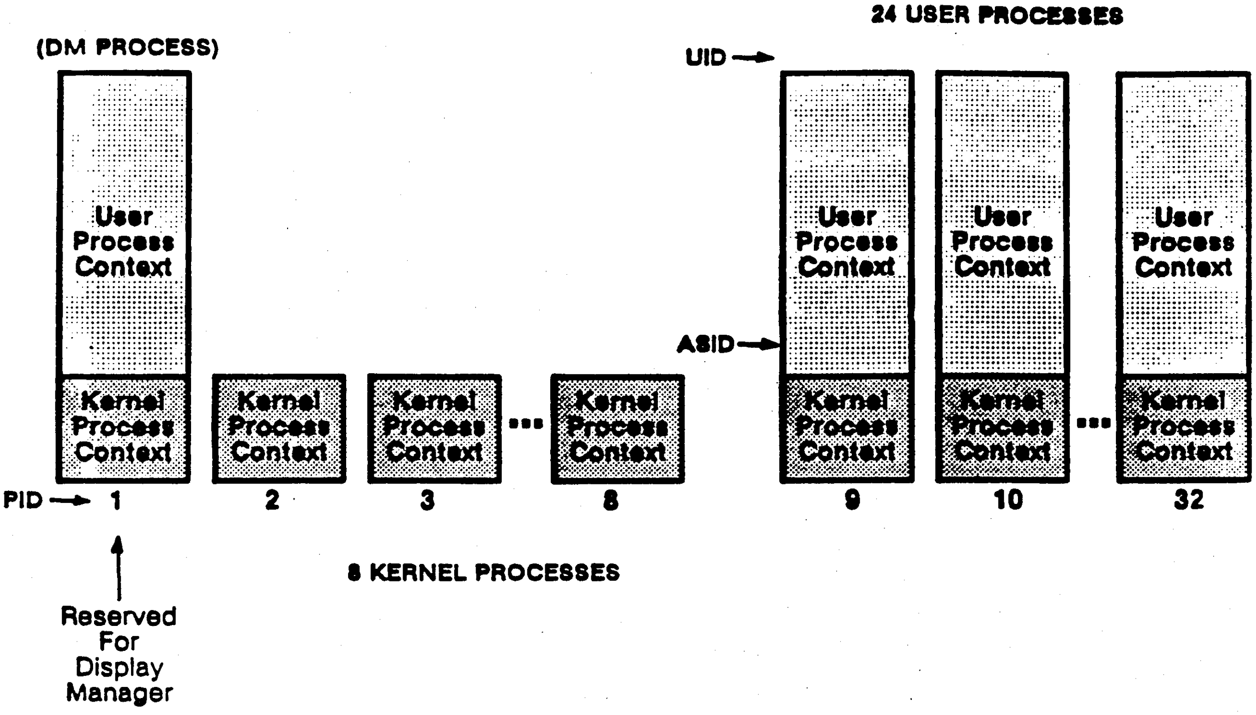


Figure 2-1. Process Levels

2.2.2.3. Process Synchronization

In the AEGIS system, process synchronization is based on eventcounts. An **eventcount** is an object that keeps a count of the number of events within a particular class that have occurred so far in the execution of the system. A process signals the occurrence of an event by advancing the eventcount associated with it. Each time the eventcount is advanced, the counter value is incremented. Consequently, waiting processes can synchronize their operations around an eventcount by:

- Waiting on the very next event by waiting for the eventcount to be advanced to a new value

- Waiting on a future event by reading the eventcount's current value, then waiting for it to reach the future trigger value (the *current* value plus the *nth event* value)

As with processes, there are two levels of eventcounts: level 1 (EC1) and level 2 (EC2). Level 1 processes use level 1 eventcounts to synchronize operations in the kernel, while level 2 processes use level 2 eventcounts to synchronize operations with other level 2 processes.

Because the AEGIS eventcount operates as a shared object, only processes running on the same machine can use it. See the section on memory management for an explanation of shared objects.

2.2.2.4. Process Scheduling

The AEGIS system schedules processes for execution based on their priority, running the highest priority process first. The system calculates process priority inversely against the amount of CPU time the process requires. Consequently, a process that requires a large amount of CPU time is assigned a low priority. Process scheduling is dynamic. The system's scheduling procedure, called the **scheduler**, periodically checks a process's CPU needs; as its CPU need changes, so does its priority. The scheduler then performs a **process exchange** (also known as a **context switch** or **dispatch**): it switches from the lower priority process to the higher. Dynamic scheduling intends to give interactive processes priority, on the theory that an interactive process is usually waiting for the user to type.

2.2.2.5. Trap, Interrupt, and Fault Handling

The AEGIS system distinguishes between traps, interrupts, and faults. A **trap** is an instruction, like any other M680x0 processor instruction. Traps include SVC traps and traps to the PROM. For example, typing CTRL/RETURN executes a trap instruction to the $F trap. A trap generates a hardware exception that changes the normal flow of program execution. When an exception occurs, the processor hardware indexes to the appropriate trap vector address in the **trap page** and uses this address as the next instruction to execute. The trap page contains the entry points to routines that the processor hardware uses to handle exceptions and interrupts. Once the trap is handled, code execution resumes at the next user code instruction. Hardware exceptions include bus errors, zero divide, and privilege violations.

An **interrupt** is a hardware-generated event that takes the processor away from the currently running process. Interrupts vector through interrupt entry points in the trap page directly to driver interrupt service routines (ISRs). Interrupts may or may not restart or wait for completion. Although an interrupt changes the flow of execution, it is generated by system activity that occurs independently of instruction execution, while an exception always occurs as the result of instruction execution. Chapter 15 describes interrupt handling in more detail.

In addition to hardware exception vectors and interrupt vectors, the trap page contains five vectors that the AEGIS system uses to handle **SVC traps**, which are traps from user to supervisor mode. The trap handlers to which these vectors point field user-mode calls to AEGIS supervisor-mode services; these handlers are collectively called the **SVC catcher.** Chapter 19 describes these trap handlers in detail.

Faults are generated by either the hardware or software. The fault interceptor manager (FIM) handles hardware-generated faults; user code fault handlers deal with software-generated faults. Hardware-generated faults restart the instruction that caused the fault; resumption of execution after a software fault depends on how the user fault handler is designed. Chapter 18 describes the fault handling carried out by the AEGIS kernel. See the manual *Programming with General System Calls* for more information on user fault handlers.

2.2.3. Layout of Virtual Address Space

The AEGIS system allocates virtual memory into private and shared areas called per-process and global address spaces. In addition, it separates both per-process and global spaces into protected and unprotected areas. Figure 2-2 illustrates this allocation of virtual address space.

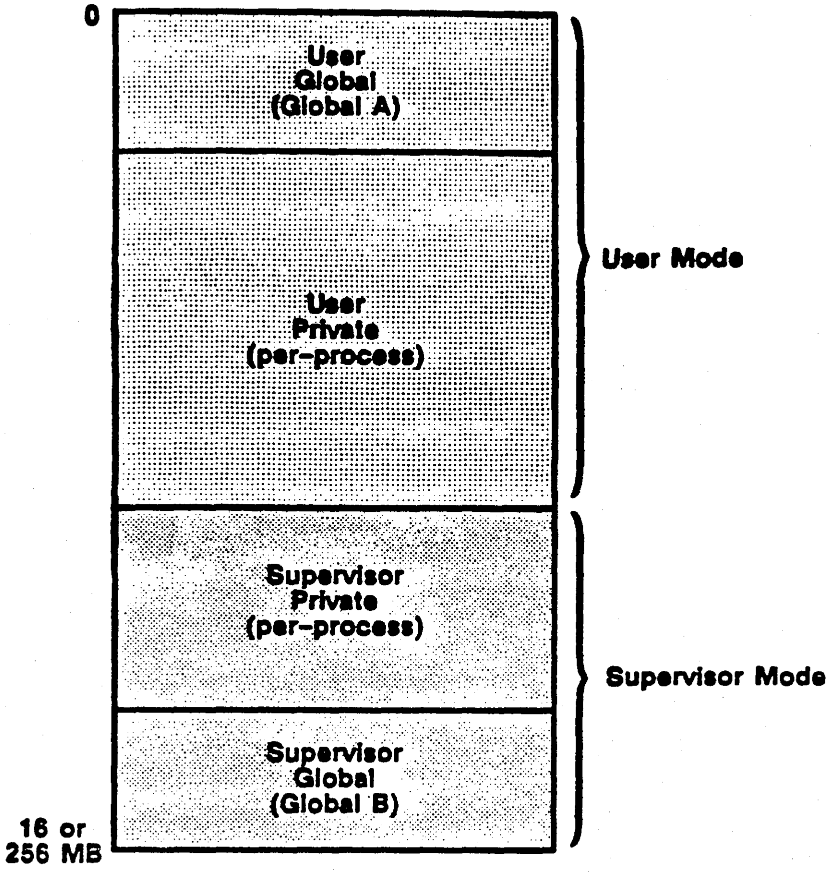


Figure 2-2. Layout of Virtual Address Space

Per-process address space is the virtual address space that the system gives to each level 2 process it creates. The unprotected portion of per-process address space is called **user private** address space and contains the process's private programs and data.

Supervisor private address space is the protected portion of per-process address space as the process can only access supervisor private address space while it is running in supervisor mode. Consequently, user-mode processes must call the supervisor via the SVC catcher to get the information mapped in supervisor private space on their behalf. For example, the system maps a process's working and naming directories in into its supervisor private space. When the user-mode process wants to access its working directory, it makes an SVC call to the naming server to fetch the directory from its supervisor private address space.

Because the contents of per-process address space varies with each process, different processes view different objects at the same virtual address. In contrast, global address space is shared among all processes in the system, so that each process views the same object at the same virtual address. Global address space is also separated into protected and unprotected regions. **User global** address space (also known as **global A**) is unprotected shared virtual memory that all the user-mode programs in the system can access. User global address space contains the global libraries and other unprotected global data.

Supervisor global (or **global B**) address space is the protected virtual memory shared among all supervisor-mode processes. Supervisor global space contains supervisor-mode programs and data such as the AEGIS kernel source code and system data structures.

The size of virtual address space differs depending on the DOMAIN node model. Chapter 9 provides more details about the contents of virtual address space for each node model.

2.2.4. Virtual Memory Management

The AEGIS virtual memory management subsystem provides network-wide access to objects. Virtual memory management includes two related operations:

- Mapping, (or **single level storage**) where the system sets up an association between a local or remote object and a process's virtual address space so that the process can refer to the object directly by referencing addresses in its virtual memory

- Demand paging, where the system dynamically transfers 1024-byte pages of an object residing on local disk or remote node to the requestor, be it on the local node or on another node in the network

2.2.4.1. Mapping

Some systems separate storage into levels; main memory is the primary storage level while the disk is the secondary storage. These multilevel storage systems allow programs direct access only to the primary storage level. A program must explicitly copy an object from secondary to primary storage before it can access the data. In contrast, the AEGIS system uses a single level storage mechanism, called SLS. Under SLS, a process gains access to an object by requesting that it be mapped directly into its address space, associating network-wide object pages with pages of process virtual address space. The direct mapping feature of SLS allows processes to access objects using programming language variables, arrays, strings, and other constructs.

In addition, once the object is mapped, the system does not demand page any data until the processor actually references it; consequently, processes can map objects to regions of process address space without incurring excessive system overhead.

The mapping between object space and process address space is the fundamental I/O primitive of the DOMAIN architecture. It provides one level of storage for all the objects in the network, whether the objects exist on local disk or on another disk in the network. It also allows users to share single copies of programs and data files. Because mapping proceeds independently of whether the object is local or remote, it provides a uniform, network-transparent way to access objects. As a result, the user can execute a program without being concerned about its location or the location of the files it uses. For example, it is possible to execute on node A a program that resides on node B, reads input from node C, and creates output on node D.

2.2.4.2. Demand Paging

AEGIS manages virtual memory over physical memory by **paging** 1024-byte pieces of virtual memory in and out of physical memory both locally and over the network. Each node has a remote paging server process that handles remote requests to read and/or write 1024-byte pages of objects on that node. When a page belonging to an object is referenced by another node on the network, the remote paging server dynamically transfers, or demand pages, it to the requesting node.

The paging system saves, or caches, copies of the pages it has fetched; consequently, subsequent references proceed at main memory speeds. The object storage system ensures that these copies are always up-to-date by purging obsolete, or stale, pages as necessary; it also automatically reflects an object's page modifications back to the node on which the object is stored. Because the AEGIS virtual memory management subsystem uses a node's main memory as a cache over objects from the entire network, the subsystem only needs to read mapped objects from local disk or from the network when they are actually demanded and are not already cached locally.

2.2.5. Network Management

The proprietary DOMAIN network consists of a token-passing ring. It is called a ring because the communications cable connects the nodes in a circle. The ring uses a **token-passing** architecture, in which a special bit pattern circulates around the ring, passing through each node. In order to transmit a message, a node must gain control of this token. The node's ring transmitter generates the message, which is received at each successive node and re-transmitted to keep the message going around the ring; this process is called **transceiving**. The ring hardware carries out the transceiving sequence without intervention from the central processor.

Although messages sent by a transmitting node pass through each node on the ring, only the target node actually processes the message. If the ring hardware on a node decides to receive a message passing through it, it awakens the processor by signaling an interrupt as well as transceiving the message. Because each node transceives the message, it eventually returns to the node that generated it. This node checks the message for evidence of the target node's receipt and removes the message from the ring.

Messages on the ring are called **packets**. While the ring hardware is responsible for getting a packet to the target node, low-level network software exists to get the packet to its intended recipient *within* a node. The AEGIS system's low-level IPC software is based on the socket abstraction. A **socket** is a queue of incoming messages; the system identifies each socket by a small number. The message sender addresses the packet to a node and to a socket number within that node. When the packet arrives at the target node, the ring receive interrupt handler places it into the socket specified by the socket number in the packet.

Because sockets provide the only means for packet delivery, a process that intends to receive messages from the network must have a socket into which the messages can be queued. The AEGIS system runs special server processes that handle requests for service from remote nodes. Since these servers expect to receive messages from the network, they are assigned sockets at system initialization. These sockets are called **well-known** sockets because the socket number assigned to a given AEGIS process is the same on all nodes; for example, the paging server is always assigned socket number 1, regardless of the node on which it is running. Well-known socket owners within the AEGIS kernel include the paging server, the remote file server, and the information server. Well-known sockets are also assigned to AEGIS user-space system services, such as the mailbox helper and the network (NETMAN) manager.

Processes that request a service from a server at a well-known socket must also allocate a **reply socket** to receive the eventual response to the request. The reply socket provides a "return address" to send along with the request packet. DOMAIN nodes support other types of communications lines in addition to the proprietary ring; for example, a node can support the communications hardware necessary to connect two or more ring networks, and has the facilities to support customer-supplied network hardware that connects to a MULTIBUS controller. An **internet** is a group of two or more connected networks. In an internet, a packet can be destined for a local network or a remote network. Packets destined for remote networks must pass through one or more interim networks before they reach the target network; this process is called **routing**. The AEGIS system contains routing software that provides the ability to send a packet to a node on a remote network and provides routing information to the nodes on multiple networks.

2.2.6. I/O Management

Each DOMAIN node model supports a different variety of I/O devices. In addition, the device controllers attached to a node differ greatly in the operations they permit and the way these operations can be performed, and usually only one of each controller type is connected to the node. Because of these differences, the AEGIS system does not currently provide a device-independent I/O interface at the nucleus level. In general, each type of device is handled by its own manager (or device driver); for example, device managers exist for the display, floating point board, magtape drive, serial I/O interface, line printer, and network ring controller. The disk controllers are the only devices that support a generalized interface above the device manager level.

A device-independent I/O facility does exist above the kernel: this is the stream interface. User processes call the stream manager to gain access to the various device managers; the stream manager exists in its own library within user space. In addition, the general-purpose I/O software library (GPIO) allows user processes to create user space device drivers for I/O devices that AEGIS does not support. The GPIO library uses the memory management subsystem to allow these processes to map a controller's CSR pages to process virtual address space to manipulate the device.

2.2.7. Time Management

The system uses four timer clocks:

- A battery-operated calendar chip clock on each node that give the time of day, month and year. The system uses this clock only during system initialization, when the system bootstrap program (SYSBOOT) uses it to get the time. These clocks can be updated using the CALENDAR utility (which can run as a stand-alone utility (SAU)). Note that each node does its own time-keeping; there is no system-wide clock to synchronize all nodes on the network.

- A real-time clock that keeps track of real time since system bootstrapping. The real time clock is a counter that is incremented in 4 microsecond intervals; the system uses this clock to calculate real time during system operation; for example, when a user issues the DATE command.

- An interval timer that the system uses to generate real-time events. A list of waiting (or **blocked**) processes is associated with this clock; the system sets the clock to a specific interval (an interval can be as small as 32 microseconds) and awakens the processes in the list when that interval has passed.

- A CPU timer that computes processor time. The system uses this clock in process scheduling; it keeps track of the CPU time accumulated by the currently running process. The system resets the clock at each process exchange.

2.3. Access Control Mechanisms

The AEGIS system supports three levels of access control that can be applied to an object:

- Controlling how an object page is accessed
- Controlling who is allowed to access the object
- Controlling how many processes can access the object at the same time

2.3.1. Processor Access Modes

Processor and memory management hardware keep track of the kind of access permitted to an object on a per-page basis: whether the page can be read, written, or executed, and in what processor operating mode (supervisor or user) these types of access are permitted. An object page's access mode is specified when the object segment is mapped, and is stored in memory management tables that differ depending on the node model. The processor checks the access mode during virtual-to-physical address translation and issues an access violation if the kind of access does not match the processor access mode.

2.3.2. Access Control Lists

The access control list (ACL) controls access on a per-object level. The ACL is a type of object that defines who is permitted access to one or more objects and also how those objects can be accessed. The system component that is using the ACL interprets the rights it specifies. For example, the virtual memory mapping routine interprets the ACL as allowing mapping with the hardware access mode read/write/execute privileges. The naming server, on the other hand, will interpret the same ACL as as allowing directory listing/name changes/adding names and links privileges.

2.3.3. Object Locking

The AEGIS object locking facility controls how many processes can simultaneously reference an object page. This locking facility consists of a lock manager that synchronizes process access to objects in the network. The lock manager enforces user-selected object concurrency rules. It also supports the distributed system by notifying the virtual memory management subsystem on each node when to flush obsolete object pages from its cache, and when to send modified pages of remote objects back to the nodes on which they are stored.

2.3.4. Resource Control

The AEGIS system also controls access to system resources through **resource locks** and **mutex locks**.

The system uses resource locks mainly for kernel process deadlock detection. Resource locks are hierarchically ordered (from 1 to 32); processes with higher priority resource locks (low numbers) run before processes with lower priority locks. Disk and network I/O locks are the highest priority to ensure that processes have access to all the other resources they need (by obtaining the associated resource locks) before they proceed with disk or network I/O.

Kernel-level mutex locks provide mutual exclusion to system resources; the system uses mutex locks to synchronize process access to resource locks. For example, if five processes want the resource lock to lock one disk, the mutex lock controls which process will get the lock first. (Currently, the first process that requested the lock gets it.)

AEGIS also supports a user level mutex library, which uses the kernel level 2 eventcount mechanism to guarantee process synchronization.

2.4. The User Program Environment

The user program environment is composed of the process manager (PM) and the various standard and optional software libraries.

2.4.1. The Process Manager

The process manager is the user-mode AEGIS software component that handles the invocation and execution of programs. A program appears to AEGIS simply as an object that consists of a procedure or set of procedures bound together by the system's binder program. (The binder program is one of a collection of programs that make up the user environment.)

Programs do not exist within process address space for the life of the process. Instead, the process manager maps programs in and out of process address space as they are invoked. When a program is invoked, the process manager *loads* the program by mapping it to process address space; it also attempts to resolve references to the entry points of external procedures by checking them against the known global table (KGT), which stores the entry points of all the libraries installed in process address space. This process is called **dynamic linking**, because external references are resolved when a program is invoked, rather than when it is bound.

The process manager is also responsible for tracking the resources a program uses -- the objects it maps, the streams it opens, the scratch storage it uses, and the data bases it manipulates -- and for releasing these resources when the program exits.

The process manager actually consists of several separate programs:

- The program manager (PGM), which invokes programs
- The loader, which loads programs into process address space
- The read/write storage manager (RWS), which allows the program to allocate scratch storage
- The mapped segment manager (MS), which tracks a program's access to objects
- The process fault manager (PFM), which handles faults sent from AEGIS kernel modules to the user program environment

Process manager internals will be described in a separate document.

2.4.2. Libraries

The software libraries provide the environment within which programs run. They consist of all the entry points that a program can call that reference procedures existing outside of the program. Most of the system services that the DOMAIN system provides are made available through libraries; in fact, the entry points to the AEGIS interface are contained in one of the software libraries.

There are two types of libraries: private and global. Private libraries exist only in the address space of processes in which they have been installed. Users install private libraries with the INLIB program (part of the user environment discussed in Section 2.5). The program loads the libraries and also places their externally callable entry points into the known global table.

Global libraries exist in the address space of ALL processes. The process manager automatically installs these libraries into user global address space as part of its initialization process when a node is bootstrapped (when the DM or SPM is first invoked).

Private libraries are shareable among processes because of the system's shared object mechanism; see Section 2.2.4.1.

2.5. The User Environment

The DOMAIN user environment consists of a native environment and a UNIX environment, called DOMAIN/IX. The native environment provides a command line interpreter, called the **shell**; the shell accepts and parses lines of input and also invokes various programs, or **shell commands**. The shell commands provide users and application programs with a comprehensive set of standard computing operations, such as compiling, binding, and copying files.

The native-mode shell supports I/O redirection facilities such as pipes and filters as well as allowing users to create their own command procedures, or **shell programs**. Shell programs are written in a programming language whose constructs are in many ways similar to a conventional language. However, an executable statement in a shell program frequently involves the complete execution of one or more additional programs. Thus, the shell program is like a sophisticated command procedure that coordinates the execution of multiple program steps.

The DOMAIN/IX environment supports two UNIX-style user environments: The user environent provided with the Bell Laboratories System V UNIX operating system, and the Berkeley 4.2 UNIX operating system. Both DOMAIN/IX variants support all user and program environment features offered by Bell and Berkeley UNIX. See the DOMAIN/IX documentation for more information about these environments.

2.6. The Display Manager

Within each node, the user environment is managed by a process known as the display manager (DM). The display manager allows the user to view and control separate activities both concurrently and independently by dividing the screen into windows whose size, shape, and placement are under user control.

Windows are viewports into objects called **pads**. Because it is implemented as an object, the pad behind ea^h window is practically unlimited in size. There are several types of pads. An edit pad displays an object that the user can modify, given the proper access rights. Input and transcript pads provide a process with a *virtual* terminal; the process writes output and reads keyboard input from these pads.

Users can overlay windows on top of each other, either partially or entirely. Consequently, using the display is analogous to arranging various pieces of paper on a desktop, except that the pieces of paper actively perform some function or display some graphic output.

Display manager internal operations will be described in a separate document.

2.7. System Initialization

System initialization consists of the procedures that bring a node from power-on or reset to the display manager login prompt. System initialization begins with the central processor's bootstrap PROM and extends through a variety of stand-alone software programs and system routines. In general, system initialization consists of:

1. Bootstrap PROM initialization operations

2. AEGIS bootstrapping from disk, network, or cartridge tape

3. AEGIS kernel initialization

4. AEGIS user environment initialization

Chapter 25 describes the system initialization procedure.

Chapter 3
Object Storage System Overview

The AEGIS object storage system (OSS) consists of the following components:

- The local object storage system, which manages the storage of objects on disk volumes attached to the local node

- The remote object storage system, also called the remote file system, which provides access to objects on remote nodes

- The cached object storage system, which manages a per-node cache of recently used objects, whether local or remote

- The object locating service, which aids in determining an object's location in the network

- The higher-level object management service, which uses the facilities of the cached, remote, and local storage systems to provide user space programs with a location-independent way to manage objects

Together these components create a file system that provides objects with permanent storage and allows AEGIS to use UIDs to identify and manipulate objects independent of their location in the network. Figure 3-1 shows the relationship between the object storage system components.

This chapter introduces information about objects and UIDs, in particular:

- How objects are structured into pages and segments

- How objects are described to the OSS and the rest of the system

- How UIDs are generated and structured and the role they play in object naming and location

Subsequent chapters discuss each component of the object storage system in detail.

3.1. Object Page and Segment

An object's smallest divisible unit is the 1024-byte **page**. The system stores an object's pages on **disk blocks** within the disk volume; each page occupies one 1056-byte disk block (see Section 4.1 for more information about disk block format.) The object storage system views objects on disk as a collection of pages occupying disk blocks on the volume. However, the virtual memory management subsystem defines another object division; this subsystem divides an object into units of 32 consecutive pages called **segments**. The memory management subsystem uses object segmentation in its mapping and activation procedures; see Chapter 10 for more details.

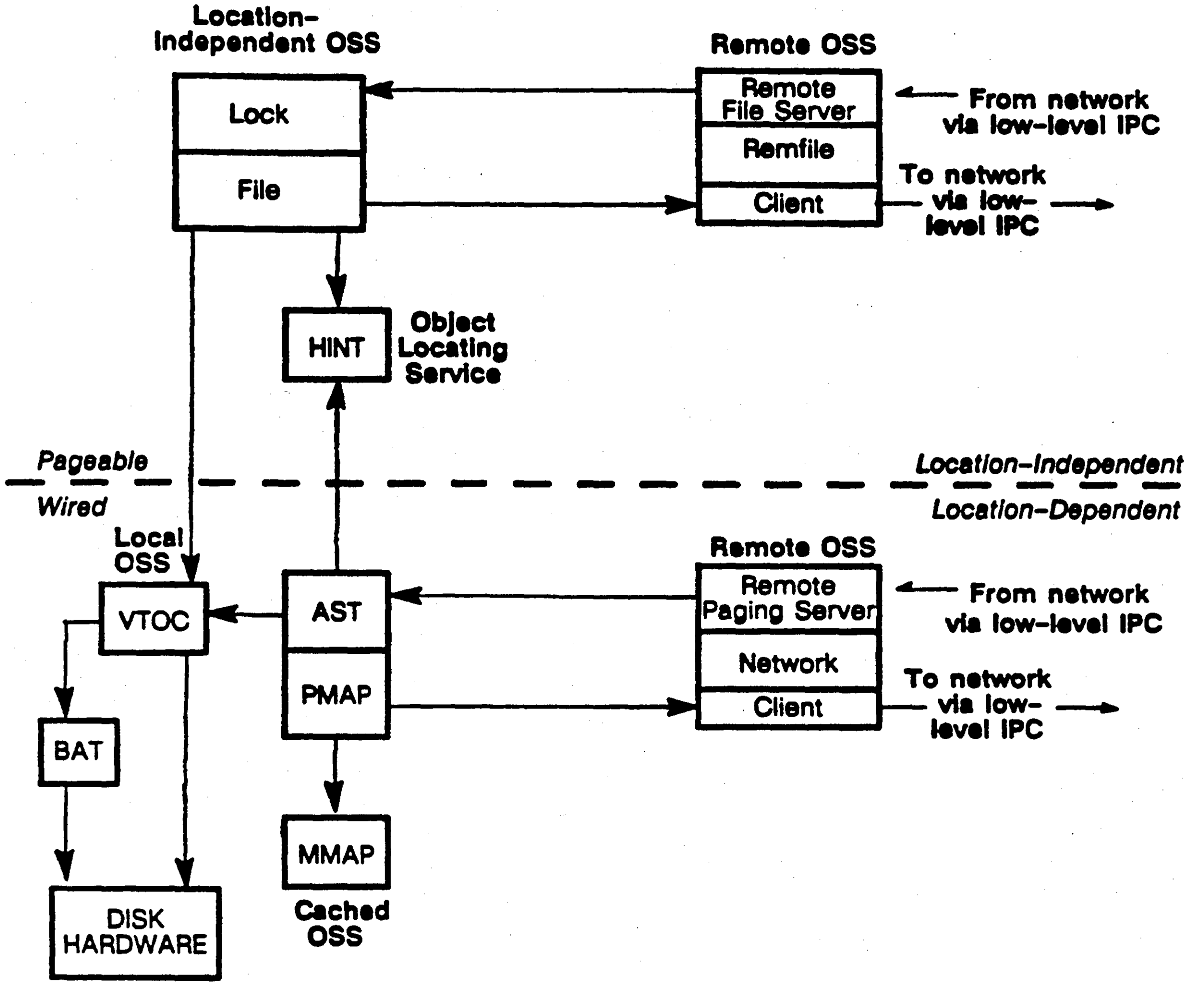


Figure 3-1. Object Storage System

3.2. Object Attributes

Each object has a set of attributes that describe the object to the object storage system, to managers in the AEGIS kernel, and to user space managers and programs. The object management service (the file manager) handles the reading and writing of an object's attributes; it assigns default attributes when it creates an object, and can also modify object attributes on a user program's or system component's behalf. The local object storage system provides the permanent storage for an object's attributes. Object attributes are described in the next sections.

3.2.1. System Type

The system type field (sys_type) is an 8-bit field used to distinguish, for the same object type, whether or not it requires special handling by the system. Currently, these system types exist:

- File objects (0)
- Directory objects (1)
- System directory objects (2)
- ACL object (3)
- Any object (-1)

The system type attribute was designed to provide the system with a way to identify objects important to its internal operation; for example, it allows the naming server to determine whether a directory object is a user directory (1) or a system directory (2). To date, however, very few AEGIS system components make use of this attribute.

3.2.2. Concurrency Control

The concurrency control attribute controls simultaneous access to the object among several processes. Concurrency control can be:

- None, where no concurrency control is applied. The system applies no concurrency control to temporary, uncatalogued objects.
- Shared reading or exclusive writing, (file_$nr_xor_1w), where any number of processes can read the object at the same time, but only one process at a time is allowed to write the object.
- Shared writing, (file_$cowriters), where any number of processes can write the object at the same time. The system limits shared writing to processes running on one node, although these processes do not have to be on the same node as the object they are referencing.

The lock manager enforces the object's concurrency control attribute.

3.2.3. Permanent and Temporary Attributes

The file system creates objects with the temporary attribute. Temporary objects can be deleted by the system when it shuts down or fails. Some objects in the system remain temporary; for example, display manager transcript pads. An object is given the permanent attribute to prevent the system from deleting it during a shutdown or crash and so that it can be catalogued in the directory subsystem.

3.2.4. Immutable Attribute

An object is marked immutable if it will never be modified. Consequently, programs can place copies of immutable objects anywhere in the network with the guarantee that they all will be identical. Currently, access control lists are defined as immutable objects.

3.2.5. Salvaged Flag

The salvage flag (trouble bit) indicates whether or not the system salvager (SALVOL) has repaired the object. SALVOL sets this bit if it detects that an object is corrupted and clears this bit after it has salvaged the object. Consequently, this bit, when set, acts as a warning flag to the system that the object may be corrupted.

3.2.6. ACL UID

Every object has an associated access control list (ACL) object that defines the type of access permitted to the object; Certain system components check the object's access control list to determine if the requested operation on the object is in fact permissible. (This manual does not currently describe access control list internals.)

3.2.7. Object Type UID

Every object also has a type manager that interprets the data within the object; the object type UID attribute identifies its type to the manager; the stream manager, for example, checks the type UID attribute to determine whether the object is is a UASC file or a record-structured file. User space type managers define classes of object types and store the class identifier -- the object type UID -- as an object attribute; AEGIS kernel managers generally do not recognize type UIDs.

Type UIDs (which are canned UIDs; see section 3.3.4) are assigned during user-mode file creation, primarily by the stream manager (stream_$create) and the mapped segment manager (ms_$crmapl). These managers call the object storage system to allocate storage for the object and its attributes, then write the type UID attribute field provided by the OSS.

3.2.8. Miscellaneous Object Attributes

Miscellaneous object attributes include the UID of the directory in which the object is catalogued, the object's current length, the date/time that the object was last used (DTU), and the date/time that the object was last modified (DTM). The object's *home node* always determines the DTM value. The memory management subsystem, in conjunction with the lock manager, uses the DTM to maintain distributed cache concurrency. See Chapter 6 for details.

3.2.9. Reference Count

The reference count attribute keeps track of how many other objects in the system are using the object. For example, several objects can share a single ACL object; the system increments and decrements the object's reference count as objects reference and dereference the ACL object. If the object storage system is called to delete an object, it first checks the reference count to determine if there are any other users; if the reference count is zero, the system deletes the object.

When the file system creates an access control list, it sets the reference count to 0. For all other object types, it sets the reference count to 1.

3.2.10. Lock Key Attribute

The lock key attribute enforces the higher-level concurrency control that the lock manager has assigned to the object. An object's lock key reflects how a user has locked the object, and can contain the following values:

- If there is no concurrency control on the object, the lock key field is 1; this key allows any node to read or write the object (any access lock key).
- If the object is locked for reading only (there are no writers) the lock key field is 0 (all readers lock key); this key allows any node to read.
- If the object is locked for write, the lock key field contains the node ID of the process that has locked the object for write; this key prevents other nodes from reading or writing the object.

The lock key attribute controls concurrent access to an object by other nodes in the network. It is the system's low level object concurrency control mechanism. Before paging in an object's pages, the paging server checks its lock key attribute (passed to it by the file system) to ensure that the requesting node is allowed access to the object. Object concurrency control and lock management are described in detail in Chapter 6.

3.3. Unique Identifiers

UIDs are 64-bit unique identifiers for objects. They are composed of:

- A 36 bit creation record
- 8 bits reserved for future use
- A 20 bit node ID

Figure 3-2 shows the structure of a UID.

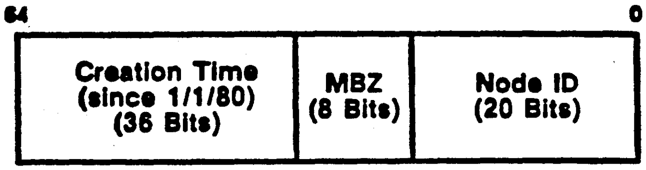


Figure 3-2. Anatomy of a UID

Because UIDs are absolute, fixed-length, and relatively short, they provide the following benefits to AEGIS system design:

- Their use as names for objects involves a minimum space penalty.
- They permit pathname translation to UID to occur in a layer above the AEGIS kernel.
- They can be used to denote the type of an object.
- They can name temporary objects which can be given pathnames at a later time.
- They can be easily hashed and stored in system tables and sent in IPC messages.
- They can be used as transaction IDs in network message-passing because they are guaranteed to be unique.
- They can be used to protect objects because they are hard to guess. Possession of an object's UID can be the key that unlocks the ability to use the object.
- They permit the possibility of composite objects by embedding UIDs within objects.

3.3.1. UIDs as Object Locators

An object's UID uniquely identifies the object no matter where it resides, but does *not* give the location of the object; this property of UIDs is called location independence. **Location independence** means that an object's UID has no relation to the object's actual location in the network. The node ID portion of the UID identifies the node on which the object was created, but this node ID does not necessarily identify the node on which the object currently resides. By convention, the only restriction placed on object location is that most objects must reside on the same volume as the directory in which they are catalogued.

Because UIDs are location-independent, the AEGIS system can move objects around in the network without having to find and alter all references to them. As a result, however, the system requires an object locating service in order to find an object given its UID. See chapter 7 for details on the object locating service.

3.3.2. Generating UIDs

The system generates a UID by concatenating the node ID of the node generating the object with a reading from the node's real time clock; the creation time has a 16 millisecond resolution. Because UID generation can occur in bursts, the system saves unused UIDs generated within the previous minute and uses these UIDs before generating new ones.

3.3.3. Guaranteeing UID Uniqueness

UIDs are guaranteed to be unique as long as the clock advances. However, to guard against clock failure, the system stores the last shutdown time on the disk. During system initialization, it checks this time against the calendar clock time. If a forward or backward time discrepancy exists, the initialization routine prompts the user for verification and/or correction. The system checks for forward time discrepancies because a forward time jump is likely to be an error that requires later correction. If the system has generated any UIDs from the erroneously advanced clock, these UIDs may be duplicated when real time catches up to the forward jump point.

3.3.4. Canned UIDs

Some system entities need to have the same UID across different systems. To avoid the assignment of a unique UID to these special objects, the AEGIS system supports **canned UIDs**, which are special UIDs guaranteed to be the same on each system. The system contains a list of canned UIDs organized into groups of related canned UIDs. System components assign the appropriate canned UID to the object. For example, the volume initialization utility INVOL assigns a canned UID of 200.0 to the physical volume label that describes the disk. See Appendix C for a list of canned UIDs.

3.4. Local Object Storage Components

The local object storage system provides access to objects stored on disk volumes attached to the node that is accessing them. The local OSS provides operations to create and delete local objects and to access the attributes and pages of existing objects. This subsystem contains two managers:

- The Volume Table of Contents (VTOC) manager
- The Block Availability Table (BAT) manager

The VTOC manager maintains the volume table of contents for the disk volume; this table contains an entry for each object that resides on the volume. The object's VTOC entry (VTOCE) stores the object's attributes and provides a *road map*, called a **file map**, to the disk blocks that contain the object's pages. The system uses the VTOC to locate an object's VTOC entry given its UID.

The block availability table keeps track of the disk blocks available for allocation. The BAT manager allocates and frees disk blocks. Chapter 4 describes the VTOC, BAT, and other disk data structures in detail and explains how the system uses these structures to manipulate local object storage.

3.5. Remote Object Storage Components

The remote object storage system (also called the remote file system) is composed of:

- The NETWORK manager, which provides remote access to the attributes and pages of existing objects
- The remote file (REMFILE) manager, which provides facilities to remotely create and delete objects

The remote OSS is one of the AEGIS network support services; it is layered on top of the system's socket datagram service and uses this low-level socket IPC to send and receive messages over the network. See Chapter 20 for a dicussion of the interaction between network services and the socket IPC service.

Both the NETWORK and REMFILE managers are object location-dependent; a remote object's location must be known before calling these managers to access it.

3.5.1. The NETWORK Manager

The NETWORK manager portion of the remote file system is responsible for the reading and writing of data on remote nodes; that is, reading and writing an object's attributes and pages. The NETWORK manager is divided into a **client side** and a **server** side. The cached OSS uses the client side to access the attributes and pages of existing remote objects not in the local cache. When the client side of NETWORK receives a request to access a remote object on a specified node, it packages that request into a message that it sends (via the low-level socket datagram service) to the NETWORK server side running on the target node; it then waits for a response.

The server side is composed of a **remote paging server** process that handles the requests from remote nodes to read or write pages and attributes of objects on the local node. The remote paging server calls upon the local cached OSS to carry out the read or write, then sends the reply back to the waiting client. Chapter 13 details how NETWORK client and server sides handle remote paging requests.

3.5.2. The Remote File Manager

The remote file manager (REMFILE) is concerned with object maintenance; it handles any remote requests that do not involve the reading and writing of data stored in an object. The REMFILE manager is also divided into client and server sides. The client side operates in the same way as NETWORK's client side, except that it packages requests from the location-independent OSS to create and/or delete remote objects. The server side uses a **remote file server** process to handle remote create/delete requests. This file server calls the local location-independent file manager (FILE) to service these requests. The REMFILE manager also handles remote lock requests; see Chapter 6. The REMFILE manager's role in file creation and deletion is described further in Chapter 5.

3.6. Cached OSS Components

To reduce the frequency of expensive disk and remote operations, AEGIS provides a mechanism to cache the results of recently performed operations so that it can use them again at a later time, providing they are still valid. The cached object storage system consists of:

- The active segment table (AST), which caches the locations and attributes of recently used, or **active** objects, whether local or remote

- The page map (PMAP), which stores the file map for one segment of an object and contains the locations in main memory of any segment pages that have been referenced

- The memory map (MMAP), which tracks the allocation and contents of main memory pages

Operations performed by the AST and PMAP managers can be classified as *file system* operations and as virtual memory management operations. These operations include:

- Fetching object pages from local or remote locations

- Reading and writing object attributes stored in the cache

- Purifying the cache by sending all modified pages back the objects' home nodes, be they local or remote

- Flushing the cache by removing objects' obsolete, or *stale* pages

Note that the AST manager does not contain a write operation for object pages. Programs modify an object's pages while they exist in the cache using MST manager mapping routines, while the purification process eventually writes the modified pages back out to disk.

The AST manager also manages its cache's consistency with the AST caches in other nodes and makes these operations available to the higher-level lock manager. Consequently, the lock manager can guarantee cache consistency for its clients, provided that they obey its locking rules. Chapter 6 provides more details. Chapter 10 describes the AST, PMAP, and MMAP managers in detail.

3.7. The Object Locating Service

AEGIS is a distributed system and thus keeps no global state information about the location of objects, as is commonly done in centralized systems. Instead, user programs and AEGIS system software obtain information about an object's location dynamically through these AEGIS components:

- The single-level store manager (the MST manager), which keeps track of an object's location when it maps the object into process address space. The system retains knowledge of the object's location for as long as the object is mapped.

- The cached OSS, which caches the location of recently used objects

- The hint manager, which carries out a heuristic search for an object, given its UID

Software components that cannot find mapped or cached location information about an object use the hint manager as a source of location information. The hint manager keeps a file of hints about object locations; there is one hint file per node that exists in the /sys/node_data directory.

A **hint** consists of the node ID and internet address at which the object corresponding to this UID was last located. Software components consult the hint manager to attempt to locate an object by locating the ID of the node on which the object currently resides.

AEGIS system components and user programs can dynamically add hints about object location. In general, any software component that can *guess* the whereabouts of an object can store that assumption in a hint file. Because most objects reside on the same volume as the directory in which they are catalogued, the naming server becomes an important source of hints for the system. In fact, AEGIS object location relies on the supposition that objects almost never move from the node on which they are created, but that if they do move, the naming server hints will almost always be accurate. Chapter 7 discusses the hint manager in detail.

3.8. The Object Management Service

The object management service exports the object management facilities of the AEGIS kernel to user space programs. It allows user space programs to create and delete objects and to read and write the attributes and pages of existing objects in a location-independent way.

The system divides object management operations between the mapped segment table (MST) manager and the location-independent object manager (FILE). The MST manager provides location-independent access to the contents of existing objects; it provides the functions to read and write an object's pages. The location-independent object manager, called the FILE manager, provides location-independent access to object attributes and provides operations to create and delete objects as well.

Both the MST and FILE managers call the kernel-level local and remote object storage systems to carry out user-mode requests to read and write object pages and attributes given the object's UID.

The FILE manager:

- Exports the cached OSS object attribute access to user programs
- Permits object creation and deletion
- Provides a location operation to return the address of an object's home node
- Exports (through the lock manager portion of FILE) the object storage system's concurrency control mechanism to user programs

To achieve location-independence, the FILE manager uses the hint manager to determine the location of an object. Once its location is established, the FILE manager performs the operation locally using the local system, or uses the client side of REMFILE if it must go remote. Chapter 5 describes how the FILE manager carries out its object management operations.

3.9. Lock Management

The lock manager is a portion of the file manager that gives clients of the object management system a method to obtain control of an object and block others from using it. A lock on an object specifies how the lock holder intends to access the object, and restricts the way in which others can access the object.

Each node has its own lock manager that keeps a lock database of all *local objects* that are locked by local or remote processes and all *local process* that have locked local or remote objects.

The lock manager exists to solve two problems:

- To control concurrent access to an object; that is, to provide orderly access to the data so that, when two processes modify the same object, their modifications do not simultaneously overwrite each other.

- To maintain consistent data between the caches in all nodes; that is, to ensure that all processes in the distributed system have the same view of an object.

Chapter 6 describes the lock manager in detail.

Chapter 4
Local Object Storage System

Disk structure defines how objects are stored on disk volumes attached to a node. The AEGIS system separates disk structure into two levels:

- Logical disk structure – Structures that th local object storage system uses to access objects on the disk; this is the logical volume level

- Physical disk structure – Structures that the disk driver uses to access objects on the disk; this is the physical volume layer and is the low-level disk structure

Figure 4-1 shows the relationship between the local object storage system managers and the lower level disk managers.

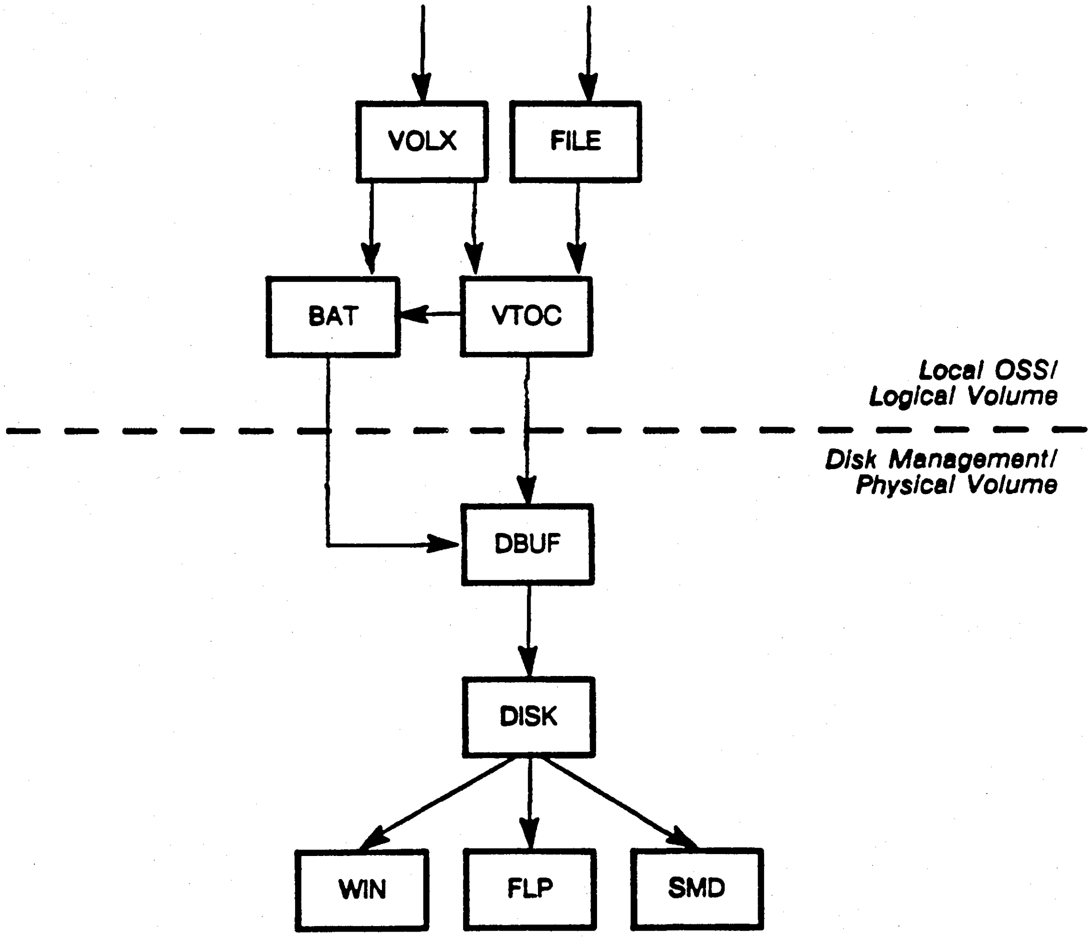


Figure 4-1. Disk Management Hierarchy

The components pictured in Figure 4-1 include:

- The location-independent FILE manager
- The local OSS managers VTOC and BAT
- The volume mount/dismount manager (VOLX), which handles private storage volumes
- The disk buffering mechanism (DBUF) used by VTOC and BAT to cache object file maps (their disk addresses on the logical volume)
- The device-independent disk manager, which fields I/O requests to the device drivers of particular disk drives
- The device drivers for the Winchester disk, floppy disk and storage module devices

This chapter discusses the local object storage system and its *logical volume* view of object storage. It covers the following topics:

- Disk block format, which is the structure of one 1056-byte disk block on a disk
- Physical volume structure as it relates to logical volume structure
- Logical volume format and the managers that manipulate it
- The role of the local OSS in file management

Another internals document will describe physical volume structure in detail and explain how AEGIS disk management components use it to read and write objects on disk.

4.1. Disk Block Format

A **physical volume** consists of some number of disk blocks. The AEGIS system defines a disk block as 1024 bytes of data plus a 32-byte disk block header. (Floppy disk blocks do not have disk block headers.) The information contained in the disk block header uniquely identifies every disk block. This information consists of:

- The UID of the object that owns the block.
- The block's page number within the object (the first block is page 0, the second is page 1, and so on).
- The time the block was last written to disk.

The type of block; that is, whether it is data (0) or a level 1, 2, or 3 file map (see Section 4.3.4 for information on file maps).

- The object type, which is file (0), directory (1), or system directory (2).
- A software-calculated checksum for the data in the block. This field is used only if read-after-write checksumming is turned on.
- The block's physical disk address (DADDR); that is, its sequence number relative to the start of the physical volume.

Figure 4-2 illustrates the disk block header.

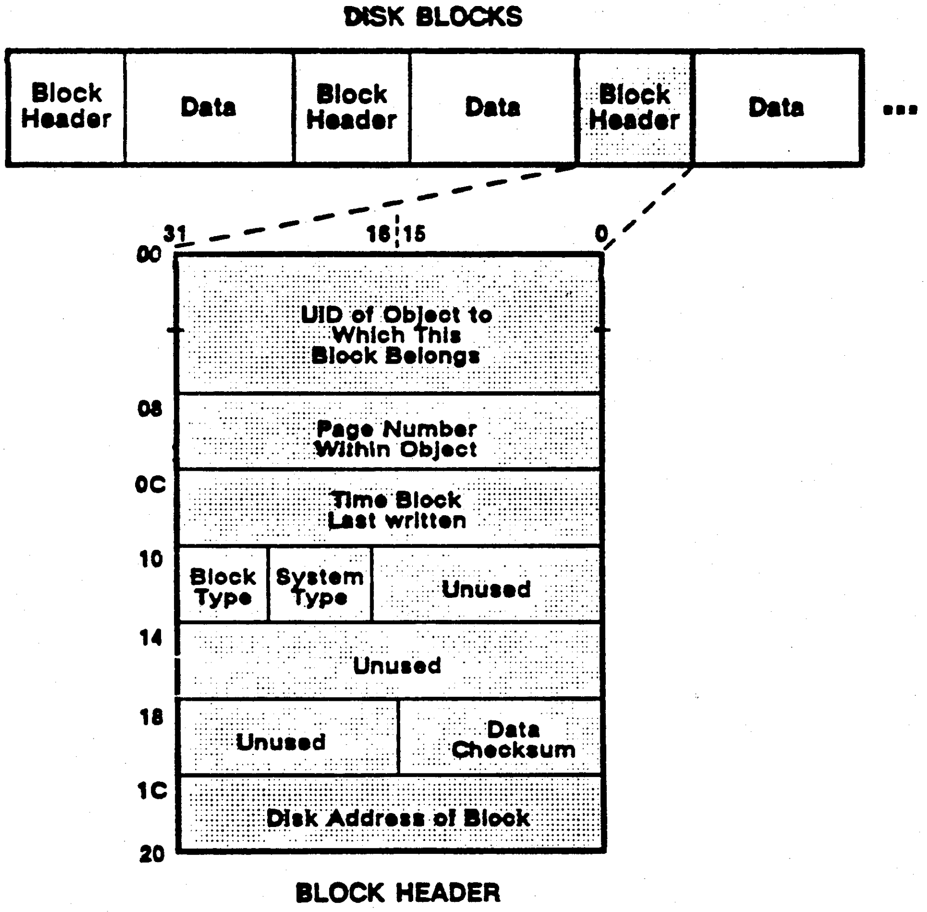


Figure 4-2. Disk Block Header Format

The disk block header exists to aid in volume recovery. If the headers for all the blocks on the volume exist, the system's volume salvage utility (SALVOL) can reconstruct the disk even if the volume table of contents has been destroyed. In addition, the disk block header protects against unreliable disk hardware.

4.2. Physical Volume Structure

The system's volume initialization utility (INVOL) builds the physical and logical disk structures onto a disk volume. Physical volume structure consists of:

- A physical volume label (PV label) that describes the physical disk and locates the logical volumes on the disk.
- One or more logical volumes.
- A badspot cylinder that records the physical badspots (unusable blocks) that exist on the volume. The badspot cylinder is usually one of the last two cylinders on a disk.
- A diagnostics cylinder that is reserved for diagnostics operations.

Figure 4-3 shows the layout of the physical volume and the structure of the physical volume label.

4.2.1. Physical Volume Label

The **physical volume label** (PV label) is a single disk block that describes the physical disk and stores the physical location of the logical volumes on the disk. INVOL always creates the PV label on the first block (physical DADDR 0) of a physical volume. The PV label is assigned a canned UID of 200.0.

The PV label provides the first step into the disk volume and provides the system with a way to distinguish between mounted volumes. The information in the PV label is as follows:

- Fields that identify the volume.
- A set of disk parameters that describe the size and shape of the physical volume. AEGIS uses these parameters to determine the size of a disk so that it does not need to depend on identification hardware or the disk drive itself.
- The physical disk address of each logical volume on the disk.
- The physical disk address of each **alternate logical volume label**. An alternate logical volume label is a copy of the logical volume label that INVOL creates when it initializes a logical volume. The alternate logical volume label makes it possible for SALVOL to reconstruct the logical volume label should the original be destroyed (usually by an erroneous write to logical disk address 0).

4.2.2. Badspot Cylinder

A **badspot** is a media defect on a disk that renders one or more blocks unusable for data storage. Most disks come from the manufacturer with a list of badspots. However, some storage devices are guaranteed free of defects. Floppy disks, for example, have no badspot lists.

When INVOL initializes a disk, it translates the hard-copy badspot list and copies it to the badspot cylinder for permanent storage. The badspot cylinder is usually one of the last two cylinders on a disk.

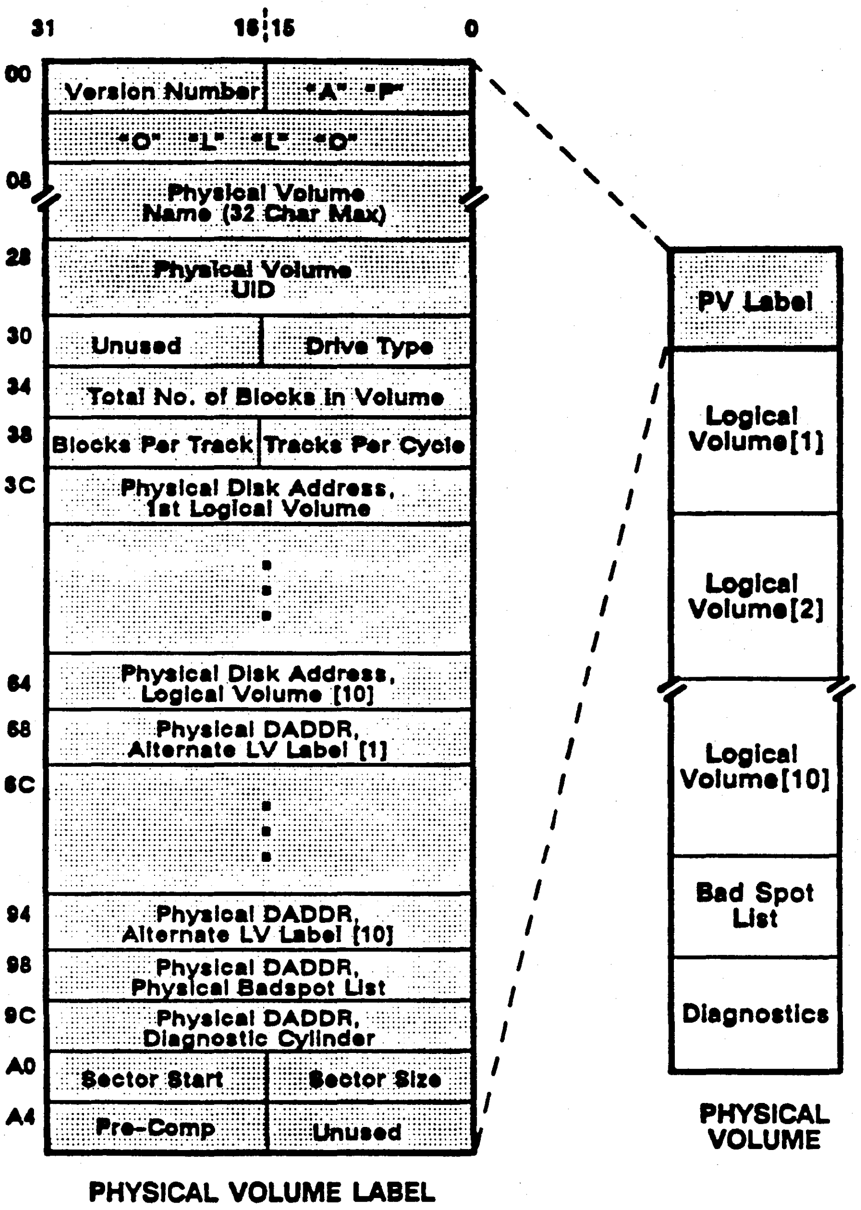


Figure 4-3. Physical Volume Structure and PV Label

4.2.3. Diagnostics Cylinder

The diagnostics cylinder is typically the last or next-to-last cylinder on the disk. It is reserved for disk diagnostic or controller diagnostic operations, and for use by the on-line TESTVOL program.

4.3. Logical Volume Structure

The **logical volume** is a section of the physical volume that is completely self-describing and self-contained. It is the structure that the local object storage system uses to record the storage of objects on disk.

One physical volume can support a maximum of ten logical volumes. The current practice, however, is to create one logical volume per physical volume because:

- The object locating service does not keep track of logical volumes; thus, finding an object that is local to the node but on one of several volumes becomes more difficult.

- The personal workstation philosophy suggests that each user on a node controls the local disk, instead of partitioning it into multiple logical volumes controlled by many different users.

- The existence of several VTOCs on one disk volume may impact system performance; one VTOC, although large, permits the system to search only once for an object.

The logical volume contains:

- A logical volume label that describes the logical volume.

- Ten contiguous blocks reserved for the operating system bootstrap program SYSBOOT.

- The block availability table (BAT), which lists, for every block allocated to the logical volume, whether or not it is available for use.

- The volume table of contents (VTOC), which contains an entry for every object stored on the logical volume. The system uses the VTOC to find objects on the disk and to record placement on the disk of new objects.

- The alternate logical volume label allocated by INVOL at volume initialization.

A block within a logical volume is identified by its **logical disk address**. A logical DADDR is relative to the start of the logical volume in which the block resides. All disk block addresses appearing within a logical volume, except those in the disk block headers, are logical DADDRs. The logical addresses of blocks on the first logical volume are one less than their physical disk addresses. For example, the logical volume label begins at logical DADDR 0, physical DADDR 1.

Figure 4-4 displays the logical volume structure and shows the fields in the logical volume label.

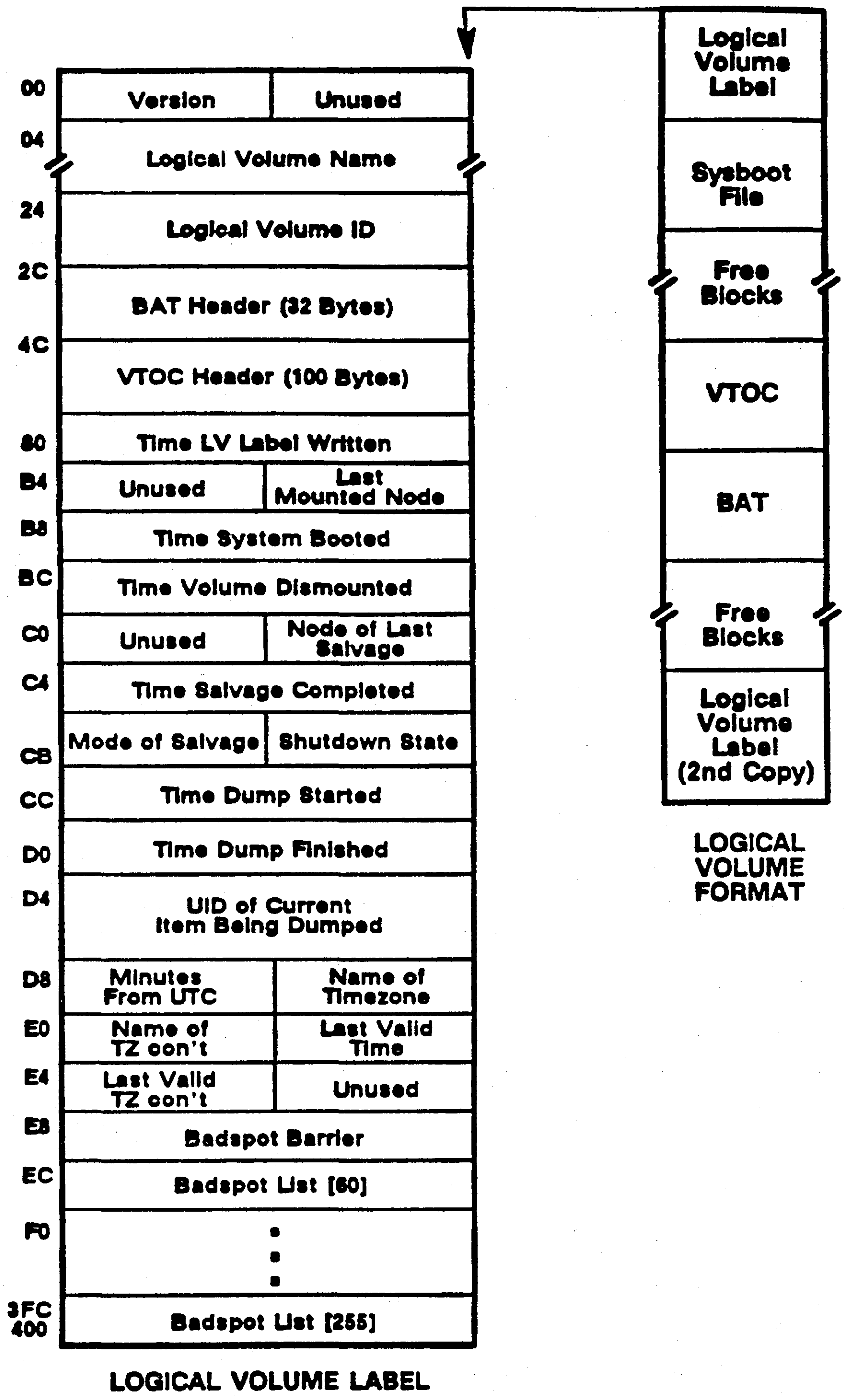


Figure 4-4. Logical Volume Label Format

4.3.1. Logical Volume Label

The first block of the logical volume (physical block 1 for the first logical volume, logical block 0) contains the **logical volume label**. The logical volume label consists of:

- Information about the logical volume's size and state.

- Headers for other data structures in the logical volume; in particular, the BAT and VTOC headers.

- Dates and times of last mount, dismount, and salvage.

- The disk addresses of the badspots that exist within the logical volume. INVOL constructs this list from the physical badspot list when it creates the logical volume. The list can contain 256 entries; if there are more than 256 badspots, INVOL allocates a *continue* block that can contain up to 60 badspot entries. The bad_spot_barrier field then contains the disk address of this block.

4.3.2. Block Allocation to SYSBOOT

The SYSBOOT program must reside in physical disk blocks 02 through 0B on any physical volume to be used as an AEGIS boot device so that the bootstrap PROM can gain access to it. These physical blocks are also the first 10 blocks of the FIRST logical volume on the disk. When it initializes the logical volume, INVOL marks these blocks as in use in the block availability table. However, INVOL does NOT copy SYSBOOT onto the logical volume. For more information about SYSBOOT, see the chapter on system initialization.

4.3.3. Block Availability Table (BAT)

The block availability table (BAT) is a bitmap that describes how the disk blocks on the logical volume are currently allocated. INVOL allocates the BAT (and the VTOC) in the middle of the disk to minimize seeks; at most, a seek will travel half the disk. The BAT resides in contiguous blocks on the disk.

Each bit in the BAT describes the state of one disk block:

- If the block is free, the value is 0.

- If the block is in use (or is a badspot), the value is 1.

The BAT header, which exists in the logical volume label, describes the location and size of the BAT. Figure 4-5 shows the relationship between the BAT header and the BAT.

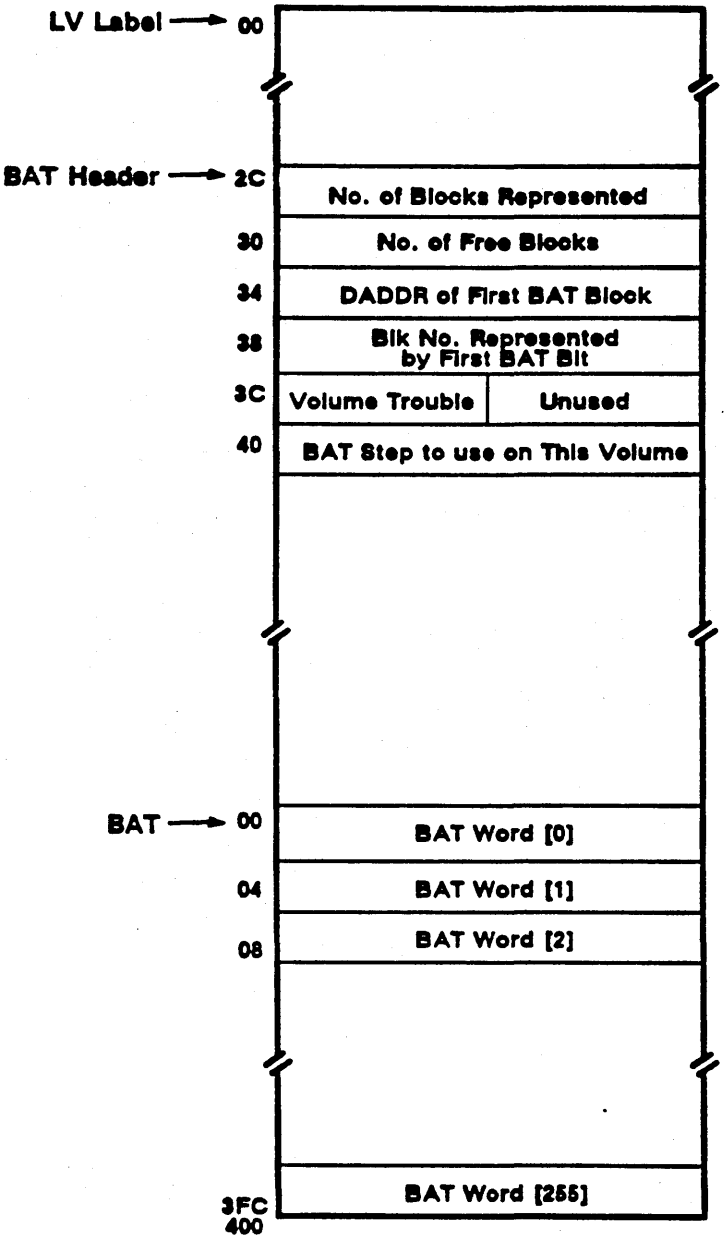


Figure 4-5. Relationship of BAT header and BAT

The BAT header contains the following information:

- The total number of blocks in the logical volume.

- The number of free blocks in the logical volume. Both SALVOL and the shell command LVOLFS (list volume free space) need the information in this field to run.

- The logical disk address of the BAT's first block.

- The disk address of the block represented by the first bit in the BAT; the first bit in the BAT corresponds to the first block in the logical volume.

- A volume trouble flag (bat_hdr.vol_trouble). The system sets this flag after a system crash to indicate that the BAT may need salvaging. SALVOL clears the flag when it has reconstructed the BAT.

- The **BAT step** to use on this volume. The BAT step controls how the BAT manager allocates blocks when it lays out a file. For example, a BAT step of 2 tells the BAT manager to allocate the next block at block n+2. Users set the BAT step with the INVOL utility to optimize disk transfer rates via sector interleaving. The correct BAT step for a given volume is the value that allows the disk to capture as many consecutive blocks in a single revolution as possible.

4.3.4. Volume Table of Contents Data Structures

The volume table of contents (VTOC) provides all the information the system needs to locate the disk blocks of an object given its UID. One VTOC exists for every logical volume; INVOL allocates the VTOC when it initializes the volume. VTOC data structures are:

- The VTOC header, which describes the VTOC and resides in the LV label

- The VTOC entry (VTOCE), which contains all the information that describes one object

- The VTOC block, a *hash bucket* of five VTOCEs

- The VTOC map, a part of the VTOC header that provides the information needed to locate all the VTOC blocks

- The VTOC index (VTOCX), which identifies a specific VTOCE

The next sections describe each of these data structures in detail.

4.3.4.1. VTOC Header

The VTOC header resides in the logical volume label. Figure 4-6 shows the fields within a VTOC header. (The offsets given are based from the start of the logical volume label.)

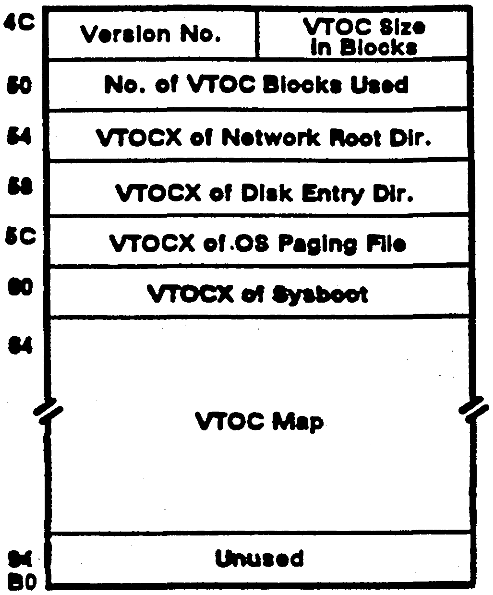


Figure 4-6. VTOC Header

The VTOC header contains the following information:

- The hash value that the system uses to locate an object's VTOCE in the VTOC given its UID. (A hash value is always the same as the size of the corresponding hash table.)

- The number of blocks in the VTOC. In general, this value is the same as the number of VTOC blocks, unless the system has allocated additional extension blocks.

- Pointers (VTOC indexes) to special objects that the system requires to boot successfully. These pointers are VTOCXs, not UIDs; a VTOCX points directly to the objects' VTOCEs. VTOC indexes are described further in Section 4.4. The special objects are:

  1. The AEGIS (OS) paging file – an uncatalogued, permanent object that must exist on any logical volume to be used as the boot device for AEGIS. The paging file is the backing storage for the pageable parts of the AEGIS operating system. The chapter on system initialization describes this file in detail.

  2. The network root directory (//) – the network-wide root directory replicated on every node. The network root directory exists in the logical volume label; no other data structure either catalogues it or points to it. Consequently, this VTOCX represents the only pointer to the directory.

3. The disk entry directory -- "/" of the node's boot volume. The system requires this information during system initialization to locate the node_data directory, the bootshell program, and the startup directories such as /sys and /dm.

   In addition, SYSBOOT needs this directory in order to locate the stand-alone utilities (see the chapter on system initialization).

4. The VTOCX to the 10 consecutive disk blocks allocated to SYSBOOT.

- The VTOC map, which describes the VTOC extents; each extent is represented by a disk address plus the number of consecutive blocks in the extent.

4.3.4.2. The VTOC Map

The size of the VTOC is a function of the size of the logical volume and the average file size specified to INVOL. Because the VTOC is usually large, INVOL cannot allocate it in contiguous blocks without running into badspots on the volume. Consequently, INVOL splits the VTOC into pieces, called **extents**. Each extent is a contiguous set of VTOC blocks. INVOL allocates the VTOC in one to eight extents. It attempts to circumvent any badspots in the volume when it creates the VTOC to minimize the number of VTOC extents it must create.

The VTOC map describes the extents that INVOL creates. Each VTOC map entry describes an extent by giving the logical disk address its first block and the number of consecutive blocks within it. Thus, the VTOC map provides the information to locate all the VTOC blocks in the VTOC. The local OSS consults the VTOC map to find the the disk address of particular VTOC blocks; see section 4.4.

4.3.4.3. The VTOC Block

A VTOC block is a hash bucket with space for five VTOCEs. Each VTOC block represents one value of hash; that is, an object's UID hashes to a particular VTOC block.

When it initializes the volume, INVOL calculates the number of VTOC blocks required for object storage based on the average file size in blocks that the user specifies (or from the default file size of five blocks) and allocates that number of blocks. However, the number of VTOC blocks initially allocated may not be sufficient to store all the objects that are eventually created on the disk.

If an object is created and its UID hashes to a VTOC block whose VTOCEs are already full, the VTOC manager handles the overflow by creating a VTOC extension block. The VTOC manager first consults the BAT to obtain a free block, then chains it to the original VTOC block by inserting its address into the *next block* field. This VTOC extension block is simply another VTOC block; however the system cannot locate it through the VTOC map because it has been dynamically allocated.

Figure 4-7 shows the relationship between the VTOC map, VTOC extents, VTOC blocks and VTOCEs.

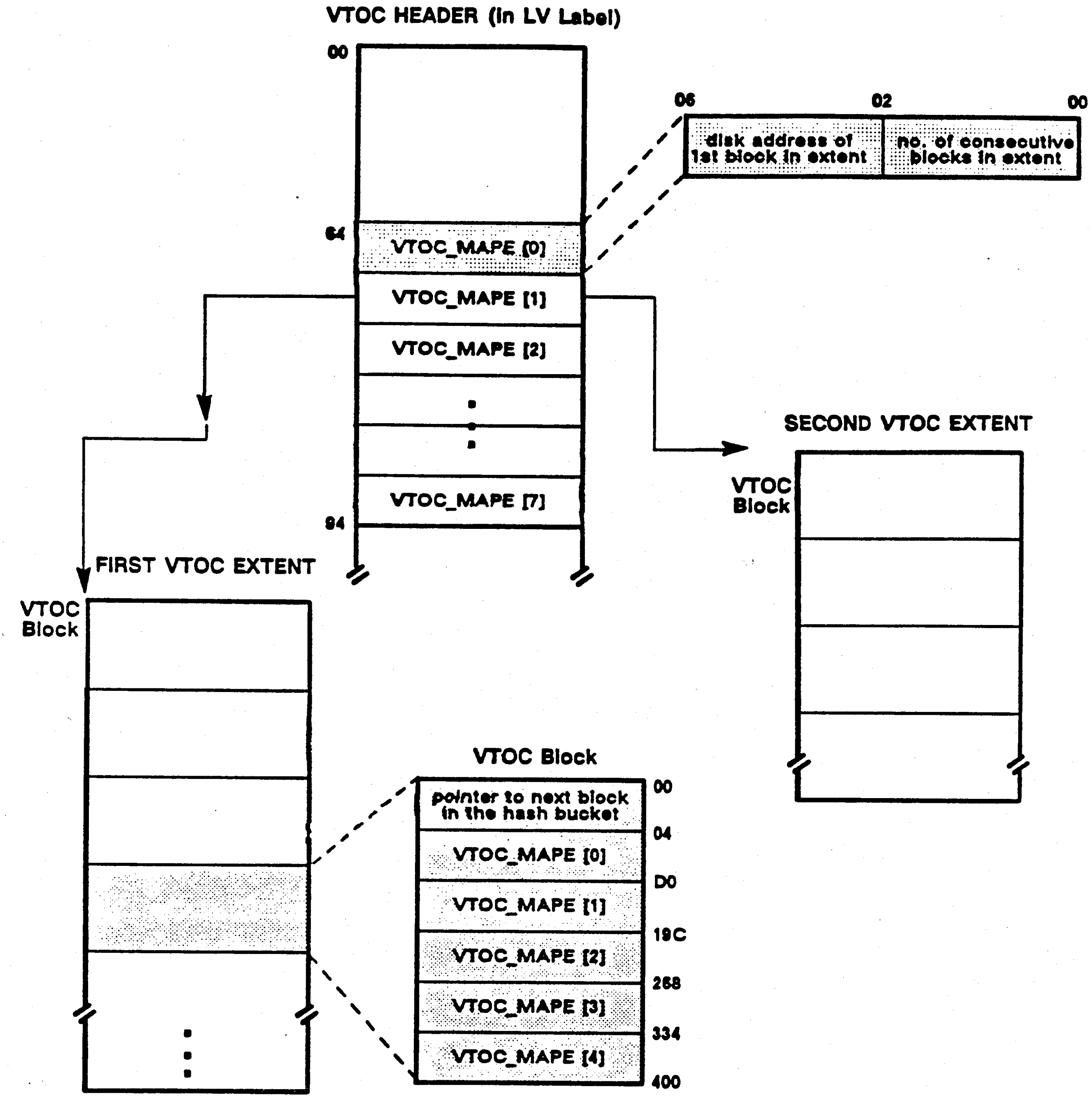


Figure 4-7. VTOC Map and VTOC Blocks

4.3.4.4. VTOC Entries

A VTOC entry, or VTOCE, completely describes a single object on the logical volume.

A VTOCE is composed of:

- The VTOCE header, which is the permanent storage for the object's attributes (which are described in Chapter 3).

- The disk addresses for the first 32 pages of the object (the object's first segment). The VTOCE provides an efficient way to represent small objects. If an object is 32 pages or less, all the information about object's data blocks resides in the VTOCE.

- Pointers to the object's other segments using file maps, which the local OSS allocates as needed. The system supports three levels of file map:

    1. The level 1 file map (L1) contains 256 disk addresses for the disk blocks allocated to the 32nd through 287th pages of the object (segments 1-8).

    2. The level 2 file map (L2) contains 256 disk addresses; each disk address points to a level one file map. Thus, the level 2 file map can map 2048 segments.

    3. The level 3 file map (L3) contains 256 disk addresses that point to level 2 file maps; it supports 524288 segments. The system only uses the level 3 file map if an object is larger than 64MB + 32KB.

The level 1, 2, and 3 file maps minimize the amount of data storage used to keep track of files. The object storage system supports pointers for every page of data to handle sparse objects (a sparse object is an object that contains gaps of zero-filled pages between actual data pages).

Figure 4-8 illustrates the structure of a VTOCE. Figures 4-9, 4-10 and 4-11 illustrate the relationship between file map levels.

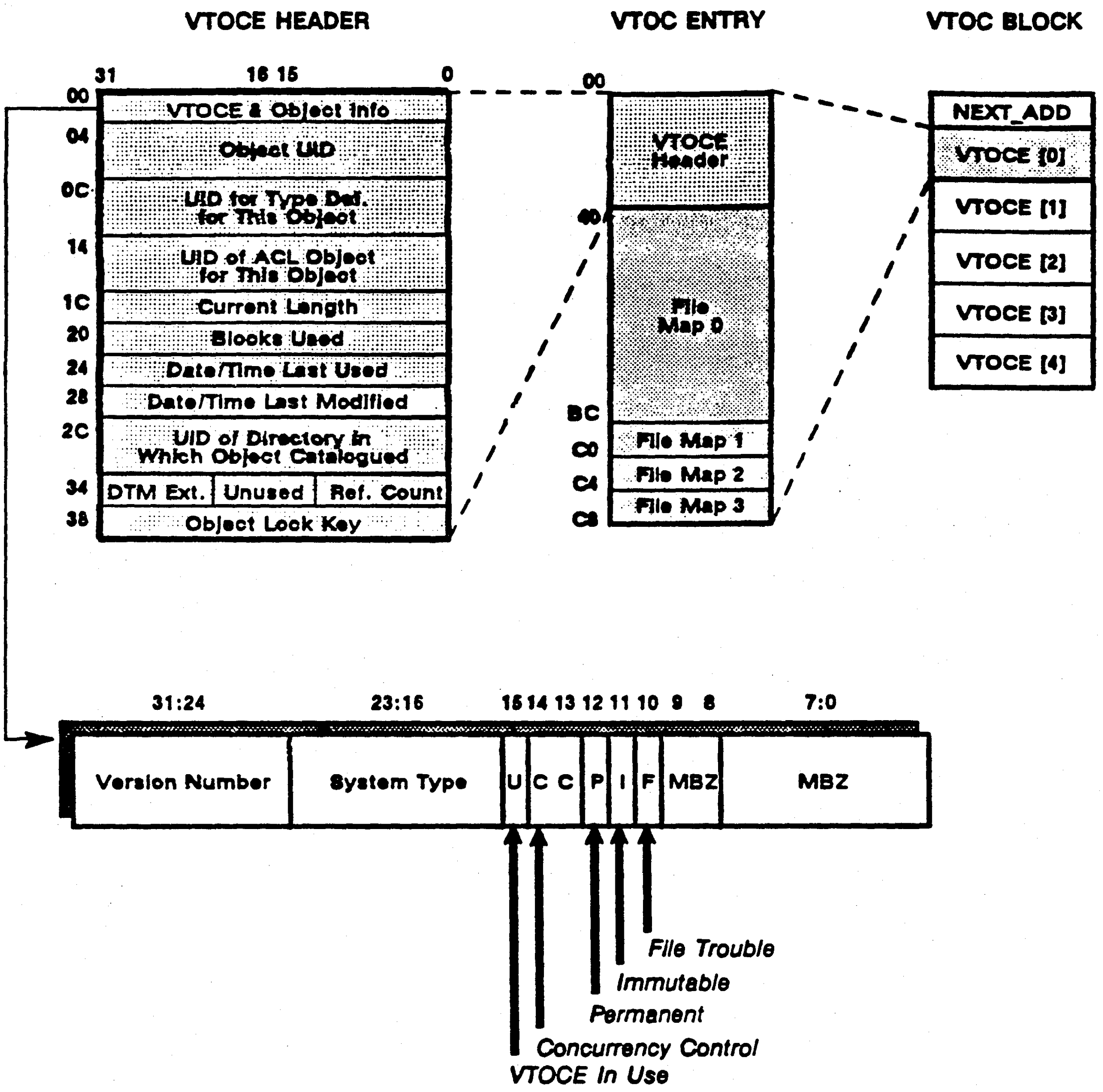


Figure 4-8. VTOC Entry

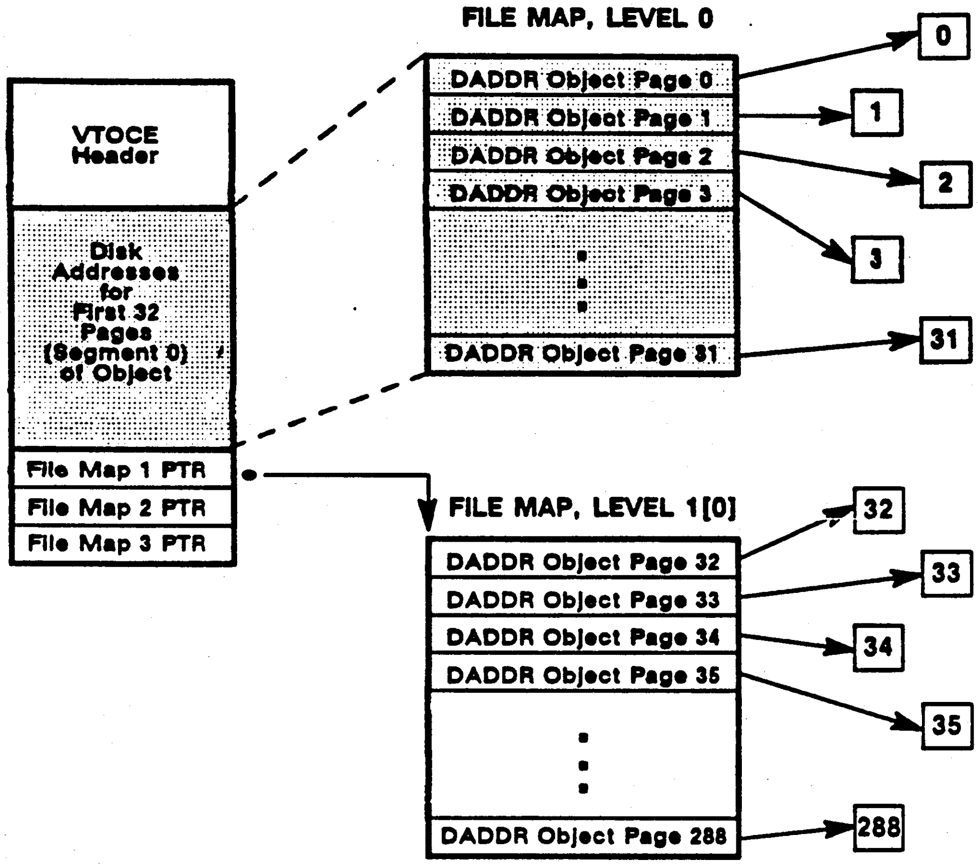


Figure 4-9. Level 1 File Map

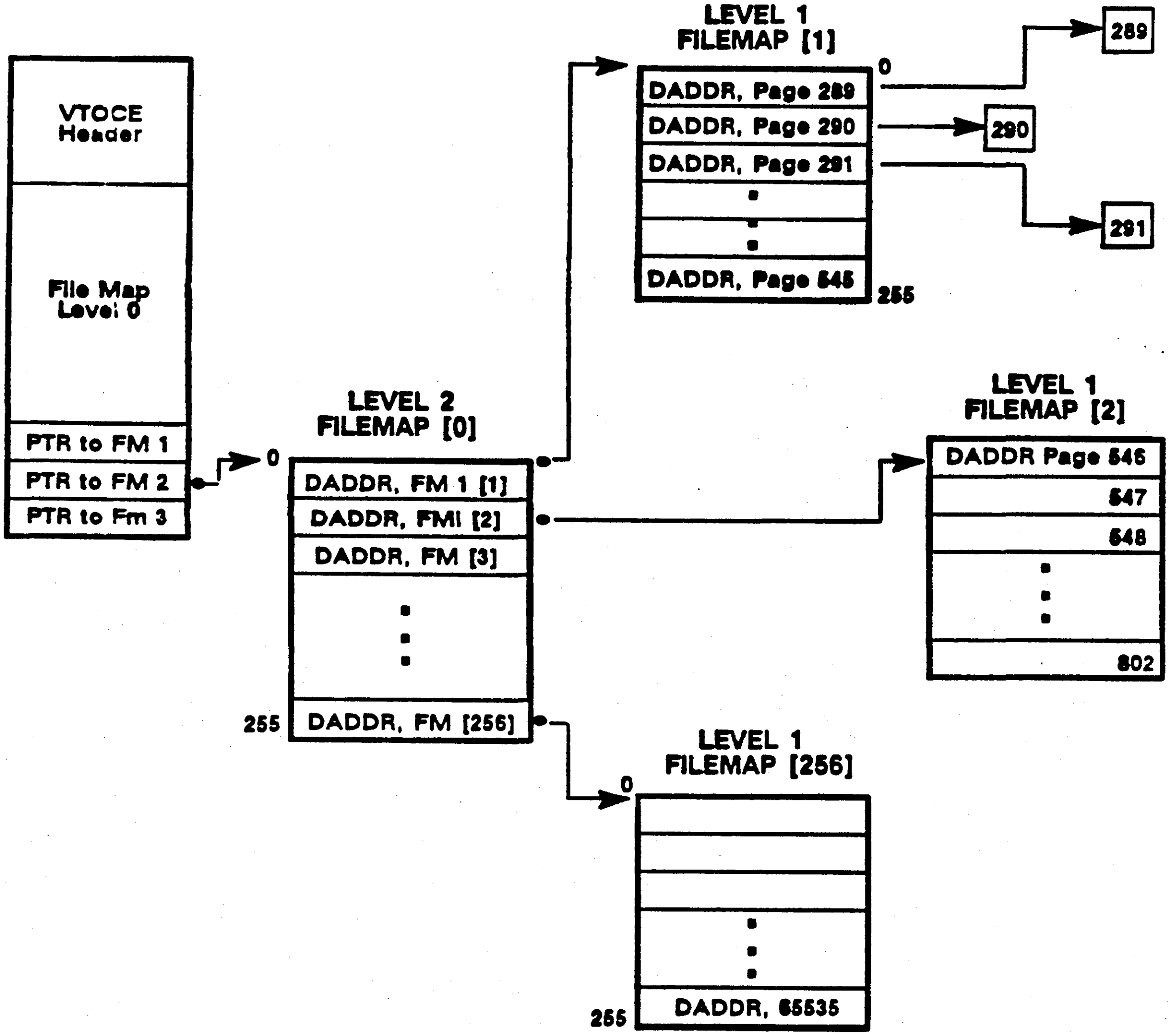


Figure 4-10. Level 2 File Map

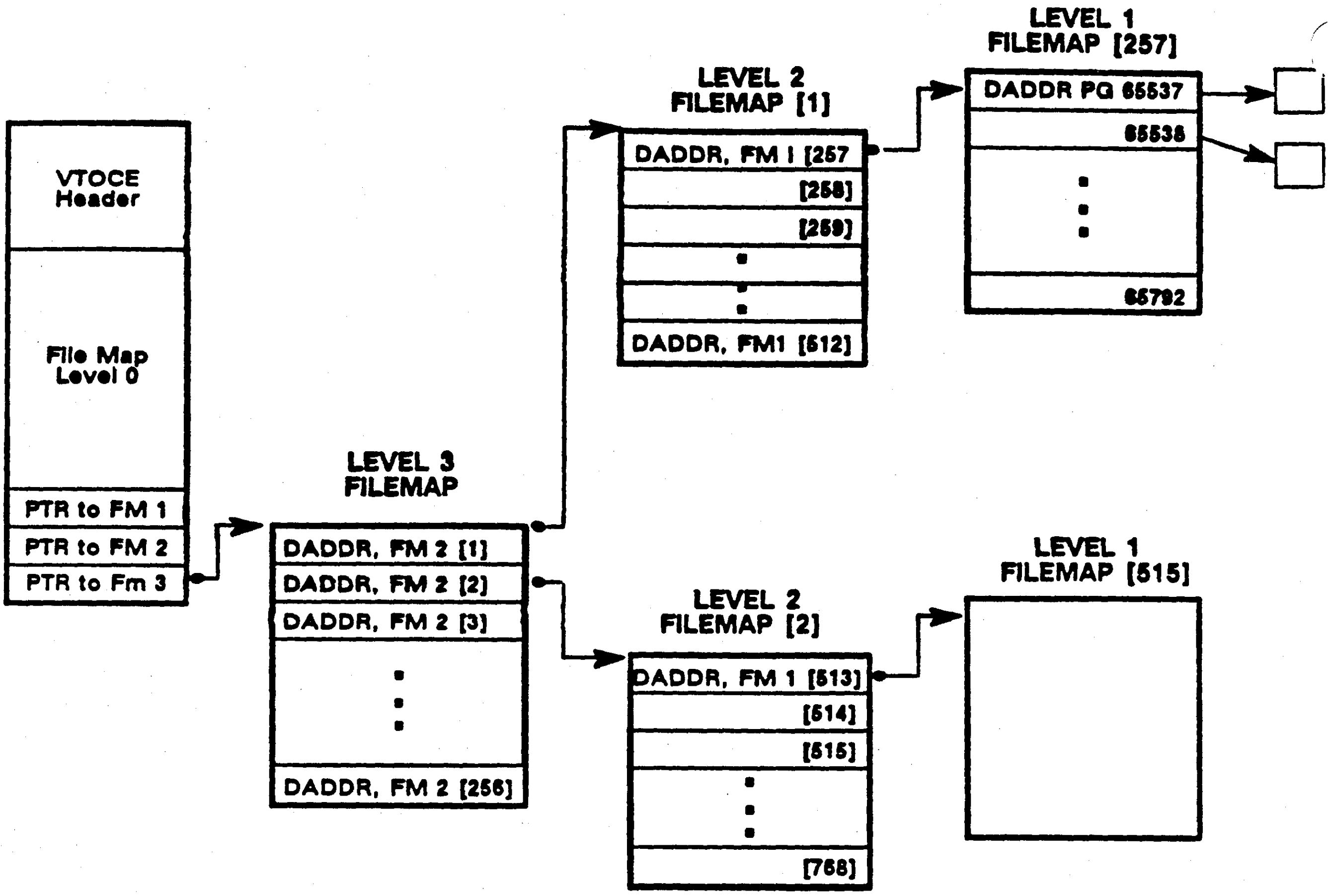


Figure 4-11. Level 3 File Map

4.4. VTOC and BAT Managers

The VTOC manager performs the following functions:

- Mounts and dismounts logical volumes
- Allocates and locates a VTOCE given a UID
- Returns the UID stored in a given VTOCE
- Gets and/or sets the UIDs of the naming server directories on a given volume
- Returns the VTOCE header for a given VTOCE
- Updates a VTOCE header with specified information and writes it to disk
- Gets a specified file map and extends it
- Deletes a file map level (truncates a file) starting at a specified page
- Reads and/or writes a file map

BAT manager operations are:

- Mounts and dismounts logical volumes
- Allocates available disk blocks
- Frees specified disk blocks
- Returns the number of free blocks on the volume
- Returns the BAT step specified in the BAT header

The next sections describe some of the more important functions performed by the VTOC and BAT managers.

4.4.1. Locating an Object in the VTOC

The memory management subsystem is the primary client of the VTOC manager lookup function; it calls the VTOC manager to read the attributes and pages of an object from the VTOCE into its object caching data structure, the active segment table. The VTOC manager lookup function locates an object given its UID. In addition, it passes the object's VTOCX back to the memory management subsystem; specifically, to the AST manager. The VTOCX points directly to the VTOCE that describes the object; it contains the logical disk address of the VTOC block that contains the object's VTOCE and the index within the block at which the VTOCE exists. The layout of a VTOCX is shown in Figure 4-12.

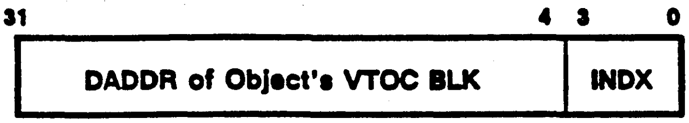


Figure 4-12. VTO C Index

For example, the pointer to the disk entry directory in the VTOC header is a VTOCX. If its value is 734D0, then the VTOCE for "/" is the first entry in VTOC block 734D, physical disk block 734E (assuming that the logical volume begins at DADDR 1). Figure 4-13 illustrates this example.

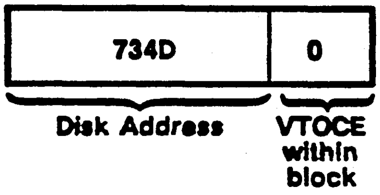


Figure 4-13. VTOC Index for Disk Entry Directory

The VTOC manager locates a VTOCE given a UID as follows:

1. Hashes the UID to get the index into the VTOC, which is the VTOC block number.

2. Consults the VTOC map to determine the disk address of that VTOC block. Given the block number, the VTOC map returns the logical disk address of that VTOC block.

3. Reads in VTOC block and compares the object UID field within each of the five VTOCEs to the given UID. If it does not find a match, the VTOC manager reads the next_add pointer to locate any VTOC block extensions. If next_add is zero, then the UID is not located on this volume.

4. If it matches the UID to a VTOCE, it creates the VTOCX and returns it to the AST manager, who places the VTOCX into the correct entry in its active segment table. The AST manager now has a pointer to the VTOCE, and marks it in use so that no one else can allocate it.

If the AST manager is forwarding the VTOCE information to a remote node, the remote AST manager's active segment table entry will not identify the VTOCE directly; instead, its VTOCX will contain the home node's ID and the number of the network to which it is connected. For more information on the AST manager, see the section on memory management; for information about network numbers, see the chapter on the internet manager in the network section.

4.4.2. Creating an Object

The managers in the object storage system interact as follows to create an object. A higher level system component, for example, the naming server (name_$create_file), calls the FILE manager to create a file, passing to it the directory UID where the object is to be catalogued.

The file manager (file_$create) uses the directory UID to determine on which volume to create the file, then creates the object in the following steps:

1. Calls the AST manager (ast_$get_info) to locate the volume that holds the directory UID.

2. Calls the UID generation routine (uid_$gen) to create a UID for the object.

3. Fills the VTOCE header for the new file with default object attributes and calls the VTOC manager to allocate a VTOCE.

4. The VTOC manager (vtoc_$allocate) takes the hashed UID, locates the appropriate VTOC block, and checks for a free VTOCE. If it locates one, it sets the "in use" field in the VTOCE header and appends the header to the VTOCE.

5. If the VTOCEs are full, it calls the BAT manager (bat_$allocate) to allocate an extension block and chains it to the VTOC block. Once it builds the VTOCE, the routine creates a VTOCX for the object and returns it to file_$create, who writes it to the active segment table entry for the object being created. In turn, file_$create returns the generated UID to its caller.

Once the VTOC space is allocated, the object actually exists.

4.4.3. Allocating Blocks on Disk

When allocating blocks, the BAT manager attempts to allocate the first available block that is nearest to the last block it allocated. It allocates free blocks as follows:

- Reads the BAT into main memory

- Locates the free blocks

- Changes the in-memory copy of the BAT

- Writes the BAT out to disk

Note that the on-disk copy of the BAT is nearly always out-of-date. Thus, one of SALVOL's most important functions is to reconstruct the BAT after a system crash.

Chapter 5
Object Management

Object management concerns three tasks:

- Object creation and deletion
- Reading and writing object pages
- Reading and writing object attributes

The AEGIS system splits the export of these tasks to user space between the single-level store (SLS) manager (the MST manager) and the location-independent object manager (the FILE manager). Together, these managers export the AEGIS kernel's object management functions to user-mode programs. The MST manager (SLS) provides the functions to read and write object pages, while the FILE manager makes it possible to read and write the object's attributes and to create and delete objects. Figure 5-1 displays these managers and illustrates their interaction with AEGIS kernel managers.

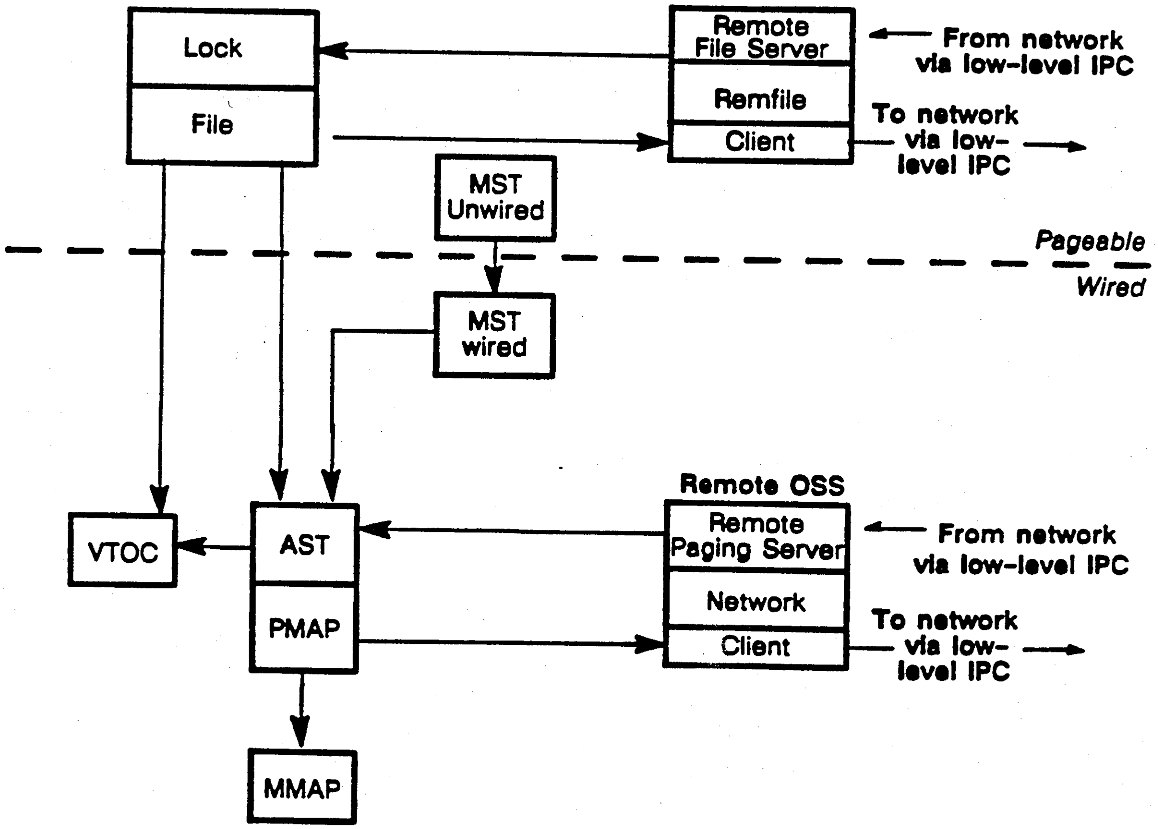


Figure 5-1. Object Management Components

5.1. MST Manager Object Management

The MST manager provides user-mode programs with location-independent access to an object's pages. The user programs specify the object's UID; the MST manager then calls the cached and remote object storage systems to locate the object and read or write the object's pages. The MST manager relies on the cached OSS managers AST and PMAP to read and write object pages. These managers obtain local object pages from disk storage, or call upon the remote file system's NETWORK manager to read in and write out remote object pages on their behalf. Chapter 10 and 13 describe how the MST, AST, PMAP and NETWORK managers carry out object page reading and writing.

5.2. FILE Manager Object Management

The FILE manager provides the following:

- Exports to user programs the ability to get and set an object's attributes
- Exports to user programs the kernel-level purification (ast_$purify)
- Permits object creation and deletion
- Provides a location operation to return the address of an object's home node

To carry out these operations, the FILE manager relies on following managers:

- The active segment table (AST) manager
- The remote file manager (REMFILE)

The file manager exports most of the functions of the AST manager. For example, the FILE manager modules to read and write attributes simply call the corresponding AST manager modules to carry out the operation. The AST manager carries out any operations on local objects or it invokes the REMFILE manager to read and write remote object attributes and delete remote objects.

The main difference between the The FILE manager and the AST manager is that the FILE manager can be called from user mode; that is, FILE modules are pageable, whereas the AST manager modules are wired, and FILE routines pass by reference instead of by value.

The REMFILE manager provides a client side that packages up requests for remote FILE level operations. The remote file client (be it FILE or AST manager) passes the arguments to the target remote file server, which executes the call and passes back the response to the client through the socket mechanism. The client side provides other services besides remote FILE operations; see Chapter 23 for more information on REMFILE operation.

The next sections describe the object management functions shared between the FILE, AST, and REMFILE managers.

5.2.1. Object Creation

The FILE manager is responsible for object creation. When called to create an object, it first calls the AST manager (ast_$get_info) to locate the volume on which the object is to be created.

If the object is to be created on a local volume, the FILE manager calls the VTOC manager (vtoce_$read/write) to create space for the object and build it into the logical volume structure.

If the object is to be created on a volume attached to another node in the network, the FILE manager calls the REMFILE manager to create the object (rem_file_$create_type). The REMFILE manager client side passes the creation request to the remote file server on the target volume, where the local VTOC manager creates the object.

5.2.2. Object Deletion

Programs can delete objects in a variety of ways. Delete operations at the FILE manager level include:

- Deleting the object on demand (file_$delete).

- Deleting the object when it is unlocked (file_$delete_when_unlocked).

- Deleting an object despite the protection assigned to it; (file_$delete_force, file_$delete_force_when_unlocked); this routine deletes the object whether or not the caller has delete rights if it ascertains that the caller has the power to change the ACL to delete rights (ACL change rights).

- Truncating and optionally zeroing the object (file_$truncate, file_$invalidate, or file_$set_len, where length equals 0).

When a program calls the FILE manager to delete an object, the FILE manager checks the access rights. If the check passes, it calls the AST manager's truncate routine (ast_$truncate).

When it is called to delete an object, the AST manager first checks to see if the object is local (by examining the VTOCX in the ASTE for the object). If it resides on a remote node, the manager determines the home node's internet address and calls the REMFILE manager to send the delete request to the object's home node (rem_file_$truncate).

If the object is local, the AST manager next checks the object's reference count to see if there are anyone else is using the object. If there are no other users, the manager deletes it. Otherwise, it decrements the object's reference count.

5.3. Reading and Writing Object Attributes

The FILE manager relies on the AST manager to read and write an object's attributes. When called to read or write an object's attributes, the FILE manager simply checks the caller's access rights to the object and then calls the AST manager to read or write the object's attributes on its behalf.

If the request is to read an object's attributes, the AST manager (ast_$get_info) reads the object's attributes:

- From the ASTE if the object has any active segments

- From the VTOCE if none of its segments are active

The ast_$get_info routine calls the REMFILE manager to send out a remote request to read the object's attributes:

- If the object is not local and the creator node portion of its UID (uid.node) specifies a remote node

- If the hint manager (see Chapter 7) returns a remote node to check

If the request is to write a specific object attribute, the AST manager (ast_$set_attribute) modifies the attribute cached in the ASTE. If the object is local, it writes this modification through to the VTOCE. If the object is remote, the AST manager calls the corresponding REMFILE manager set attribute routine, which asks the remote file server to change the attributes using the target node's AST manager.

5.4. Locating Objects

The FILE manager provides a routine that locates an object given its UID (file_$locatei) and return its internet address to the caller. Clients such as the lock manager and the naming server helper (see Chapter 8) call this routine to try to find the object's home node and network given its UID. For example, if objects are created on a floppy disk on one node, and then the floppy is moved to another node, the node ID portion of objects created on that floppy will contain the creator node ID, not their present home node ID. So, programs that want access to these objects call the file_$locatei routine to find the location in the distributed system at which they presently reside.

The file_$locatei routine first calls the AST manager to obtain the cached location information (the object's VTOCX). If the object is remote, the routine next calls the NETWORK manager to look up the network number of the object's home in the network ID table; otherwise, the object is stored locally and the routine passes back the local network and node IDs.

If there is no VTOCE associated with the object, the object behind the specified UID could be a diskess node. Because diskless nodes are named, the naming server helper could be attempting to locate a diskless node as part of its operations.

In this case, the FILE manager compares the given UID against the list of canned UIDs to see if it matches the canned UID type given to diskless nodes. If a match occurs, the routine can then consult the naming server's directory manager to find the network on which the diskless node resides (dir_$find_net).

5.5. Force-Writes and Force-Purification

The AEGIS system supports two kinds of write requests from users: force-writes and force-purifies. A force-write implies a write to disk. A force-purify writes remotely modified pages to the home node, where they are locally purified at a later time.

The FILE manager provides user programs with routines to force-write and to force-purify. The file_$purify routine force-purifies an object. The FILE calls that force-write objects to disk are:

- The file_$fw_file routine, which force-writes the entire object to disk. This routine force-writes all the modified pages of a specified object to disk, whether it is local or remote.

- The file_$fw_partial routine, which force-writes all the modified pages of one segment to disk.

- The file_$fw_on_unlock routine, which force-writes the object as soon as it is unlocked.

The force-write routines call the AST manager's purify routine (ast_$purify) with the force-write option. If the object is local, the AST manager writes the object's modified pages to disk and updates the DTM in the active ASTE with the current local time. If the object is remote, the AST manager writes the modified pages to the home node as if it was a force-purify. Sending the modified pages back to home node, however, simply sends the pages back to the home node's main memory, where they will fall into the local purification process; it does not guarantee that they will be written to disk as was requested. Consequently, the AST manager also sends a message to the remote file server on the remote node to make sure the modified pages get written to disk.

Chapter 6
Object Lock Management

The AEGIS system on every node runs a lock manager that provides file system clients with the ability to control simultaneous access to objects. Using the lock manager, a process can gain control of an object, then block other processes from using the object in an incompatible way. Both user-mode and supervisor-mode AEGIS components rely on the lock manager when accessing objects.

The AEGIS system locking mechanism is designed to solve two problems:

- To control concurrent access to an object; that is, to provide orderly access to the data so that, when two processes modify the same object, their modifications do not simultaneously overwrite each other.

- To maintain consistent data between the caches in all nodes; that is, to ensure that all processes in the distributed system have the same view of an object.

The next sections explain how the lock manager handles concurrent access and maintains cache consistency.

6.1. Controlling Concurrent Access

The kinds of access that the lock manager permits depends on the object's concurrency mode, its access mode, and its lock key.

6.1.1. Concurrency Mode

The AEGIS system defines three concurrency modes that a process can apply to an object:

- No concurrency control

- Protected concurrency control, which allows shared reading or exclusive writing (file_$nr_xor_1w)

- Shared concurrency control, which allows shared writing (file_$cowriters)

6.1.1.1. No Concurrency Control

An object that the FILE manager creates has the following characteristics:

- It is a temporary object.

- It is uncatalogued; that is, it has no associated pathname entry in the naming server's directory structure.

- It has no concurrency control assigned; that is, anyone in the distributed system can potentially read and write it.

The object's creator is responsible for making the object permanent, cataloguing it, and assigning it another concurrency mode. Some objects, however, remain temporary and uncatalogued because their creators do not intend to share them with other processes. Because they are not shared, these objects do not need any concurrency control; for example, the PROC2 manager does not apply concurrency control to the stack object, because only one process uses it exclusively.

6.1.1.2. Protected Concurrency Control

Protected concurrency mode (nr_xor_1w) synchronizes access to an object at the process level. Protected concurrency control means that either any number of processes can read the object, and writing is blocked, or only one process can write the object, and all readers are blocked.

6.1.1.3. Shared Concurrency Control

Shared concurrency control (cowriters) allows any number of processes to simultaneously write an object, but all the writing processes must be running on the same node; the object, however, can reside on a different node than the processes that are referencing it. Consequently, the cowriters type of concurrency provides protected concurrency mode (nr_xor_1w) access to objects at the *node* level: any number of nodes can read the object, and writing is blocked, or one node can write the object, and all readers are blocked.

6.1.2. Access Mode

When a process locks an object, it specifies, in addition to the object's concurrency mode, the way it wants to access the object; this is the lock's **access mode**. The lock manager supports three types of access for protected and shared locks: read, write, and read with intent to write at a later time (RIW). (Note that it does not support write with intent to read at a later time.)

In addition, processes can direct the lock manager to change from one access mode to another: change read to write, change read to RIW, and so on. The hierarchy of access modes and change access mode requests (from weakest to strongest) is as follows:

1. All (used on unlock only)
2. Read
3. Read with intent to write (RIW)
4. Change read to write
5. Write
6. Change write to read
7. Change read to RIW
8. Change write to RIW
9. Mark for deletion
10. Unmark for deletion

Mark for delete is not really an access mode. Rather, when set, it directs the lock manager on the home node to delete the object as soon as it has released the last lock held on it. The system "locks" temporary objects with the mark-for-delete access mode to enforce automatic resource cleanup during process deletion. The PROC2 manager's delete operation releases all locks that the target process holds; consequently, the lock release will trigger the object deletion (which the AST manager ast_$truncate routine carries out).

6.1.3. Lock Compatibility

Before the lock manager can grant a lock to a process, the requested lock must be *compatible* with the other locks held on the object. The compatibility rules are:

- Shared reads and protected reads are compatible and can originate from different nodes.

- Shared reads, RIWs, and writes are compatible if the locking processes are on the same node.

- Shared RIWs are compatible with protected reads, and can originate from different nodes.

- Shared reads are compatible with protected RIWs, and can originate from different nodes.

Table 6-1 summarizes the concurrency control applied given a concurrency mode and an access mode.

Table 6-1. Lock Compatibility

	Protected R	Protected RIW	Exclusive W	Shared R	Shared RIW	Shared W
	YES	YES	NO	YES	YES	NO
Protected RIW	YES	NO	NO	YES	YES	NO
Exclusive W	NO	NO	NO	NO	NO	NO
Shared R	YES	YES	NO	YES	YES	YES*
Shared RIW	YES	YES	NO	YES	YES	YES*
Shared W	NO	NO	NO	YES*	YES*	YES*

* must be on same node

Thus, under the compatibility rules, if an object is locked for protected reading, other processes, both local and remote, can also obtain read locks on the object, but no processes can obtain a write lock. One protected write lock, however, blocks any other writers or readers.

On the other hand, if a process holds a shared read lock, any process *on the same node* can simultaneously hold a shared write lock. As a result, holding a shared read lock does not bar other processes from getting a shared write lock and changing the object. It is up to the application using the shared concurrency mode to control concurrent access between the processes on that node.

In addition, the lock manager permits protected reads to occur simultaneously with shared reads, as long as there are no unregulated write locks on the object.

6.1.4. The Lock Table

The lock manager on each node maintains a **lock table** that records:

- All locks on local objects, whether they are held by local or remote processes
- All locks held by local processes, whether they lock remote objects or local objects

Consequently, when a process requests and obtains a lock on a remote object, two lock tables are updated: the table on the object's home node reflects the fact that the local object is locked, whereas the table on the calling process's node indicates that the process holds a lock. Lock tables never store information about remote processes holding remote locks; either the process or the object must be local.

The information stored in the lock table entry includes:

- The locked object's network number and UID.
- The locking process's network number and UID.
- The lock's concurrency mode (protected or shared) and access mode (read, write, or R/W).
- The transaction ID, which identifies individual lock requests in the event that a single process locks the same object more than once. The lock manager also uses the transaction ID to identify duplicate lock operations. On lock requests, the duplicate lock transaction ID will match the existing lock, so the lock manager can discard the duplicate. On unlock requests, the lock will no longer be recorded in the lock table.
- Whether or not it is marked for delete; only the lock table on the object's home node will indicate mark-for-delete status.

The object UID, process UID, and transaction ID together define a lock's uniqueness.

6.1.5. Lock Key

In addition to the lock entries in the lock table, the cached and local object storage systems (AST and VTOC) provide a **lock key** field that reflects how the object is locked. The field can contain any one of the following values:

- If the lock key field is 1, there is no concurrency control on the object. This key allows any node to read or write the object (any access lock key).

- If the lock key field is 0, the object is locked for reading only (there are no writers). This key allows any node to read the object (all readers lock key).

- If the lock key field contains a node ID, the object is locked for protected or shared write. The node ID in the lock key corresponds to the process that has locked the object for write. This key prevents processes on other nodes from reading or writing the object.

The lock key provides cached concurrency control. The system uses the key to enforce the locks that processes have obtained and stored in the lock table. Specifically, the AST manager on the object's home node checks this field when it is called to process remote read or write requests for the object's pages (ast_$touch). On a remote request to read a page, the home node checks the lock key and permits the read if:

- The lock key is all readers

- The lock key is any access

- The node ID of the reader matches the node ID in the lock key (shared reads and writes can occur simultaneously, but they must originate from the same node)

If the lock key and request are not compatible, the cached OSS reports a read concurrency violation.

If the remote request is to write a page, the write is permitted if one of the following conditions is met:

- The lock key says any access

- The node ID of the writer matches the node ID in the lock key

Otherwise, the system issues write concurrency violation status.

The lock key check enforces the lock in two ways. First, it ensures that the locking process cannot carry out an an illegal operation under the lock, like writing the object when it has locked it for read access. Second, it ensures that other processes will not break the locking process's lock as it is recorded in the lock table.

Storing the object's lock key in the ASTE also optimizes page I/O, because it allows the system to determine quickly whether or not the requested paging operation is legal.

6.1.6. Obtaining a Lock

A process that wants to lock an object makes a lock request to the lock manager, specifying the object's UID, the concurrency mode to assign, and the way it wants to access the object (R, W, RIW). When it receives the request, the lock manager checks its lock table to determine whether the process is allowed to lock the object; that is, whether the lock request is compatible with the other locks stored in the table. Because locks with different concurrency modes are compatible, the lock manager must scan the entire chain of locks for an object to make sure that the requested lock is compatible with every other lock held.

If the requested lock is compatible, the lock manager grants the lock by creating a new lock entry for the object in the lock table. In addition, if the lock key required by the new lock is different from the lock key cached in the object's active segment table entry (that is, if an ASTE for the object exists), the lock manager writes the new lock key into the ASTE. If the object is local, the AST manager will write it through to the VTOCE.

6.1.7. Changing a Lock's Access Mode

A process can change the access mode of an existing lock. The lock manager follows the exclusive or rule when handling requests to change the access mode of an existing lock:

- If the existing lock is a protected read, the caller can change it to a write lock as long as there are no other readers.

- If the existing lock is a protected write, the caller can always change to a read lock.

A process can only change the access mode of an existing lock; it cannot change its concurrency mode.

6.2. Maintaining Consistent Data

The lock manager uses the object's DTM attribute to maintain consistent data between nodes in the distributed system. To carry out this function, it observes the following rules:

- The object's DTM can only change while it is locked.

- The home node must always have a complete copy of the latest data when the object is not locked.

- The locking node's cache must be validated while the object is locked; that is, the file system client must verify that the object pages it has cached are the most recent before it reads or writes the object.

To obey these rules, the lock manager and a client process interact as follows:

1. The client makes a request to lock the object for reading. The lock manager sets up the object's lock key and passes back the object's DTM to the client.

2. If the client has previously cached segments of the locked object, it compares the DTM that the lock manager has returned with the DTM stored with its cached copy of the object. The client makes this check to determine whether the copy of the object in the cache is the most recent version. If the client finds that the cached version is obsolete, it flushes the old version from the cache and reads in the new version from the home node.

3. The client requests some read operations. On each read, the AST manager checks the lock key to make sure that the client accesses the object properly.

4. The client changes to a write lock (the lock key changes to the process's node ID), and makes some write requests on the object. Each time the client writes a page to the object's home node, it receives a new DTM from the home node.

5. If the client changes the lock from write to read (or RIW), the lock manager calls ast_$purify to write the modified pages to the home node. This operation keeps the home node's copy of the object consistent so that other readers won't have to go to the locking node for the latest copy.

6. When the client unlocks the object, the lock manager writes any modified pages still on the client node back to the object's home node before it releases the lock. This operation ensures that the next client to lock the object will find an up-to-date, consistent copy of the data on the home node. If the object is still locked for read and the lock manager is running on the home node, it also updates the lock key to reflect the "all readers" status.

6.2.1. Lock Verification

The home node's lock table does not always reflect an object's current lock status, especially when the object is locked by processes that are running on remote nodes. A system crash on the locking process's home node, or lag time between local and remote modification to the lock tables can cause the lock table on the home node to reflect a lock that is no longer held.

To ensure that a process obtains up-to-date lock status, the lock manager's lock request service provides a lock verification procedure that checks the lock table entry on the locking process's home node against the lock table on the object's home node in the event that a lock request is denied. The verification procedure first locates and calls the lock manager on the object's home node to read the local lock table and pass back the lock entry for the object to which its client was denied access. This lock manager passes back the internet addresses of the process that has locked the object and the object itself.

If the locked object and the locking process reside on different nodes, the routine then sends a message to the remote file server on the *process's* home node to see if the process still holds the lock. This remote file in turn invokes the local lock manager to search its lock table for a lock entry on the object. If no entry exists, the verification procedure directs the object's home node lock manager to remove the lock table entry from the object's home node, and passes back "not locked" status to the calling lock manager, who then retries the denied lock request.

Note that the verification procedure will release the lock only if it can verify that there is no entry for the object. If the lock verification request can't get to the process's home node, it assumes the lock is still valid. This assumption takes into account the possibility that the process's node could be unavailable because it is a victim of a network failure rather than a system crash. A locking process on a node that is temporarily partitioned out of the network is still running and will eventually want the lock again; a process on a crashed system no longer exists.

Chapter 7
Object Location, or the Hint Manager

Because UIDs are location-independent, the AEGIS system can move objects around in the network without having to find and alter all references to them. As a result, however, the system requires an object locating service in order to find an object given its UID.

This chapter discusses the hint manager, the component of the object storage system that helps locate an object regardless of where it resides in the network. Figure 7-1 shows the relationship between the object storage system components as well as the modules that call the hint manager.

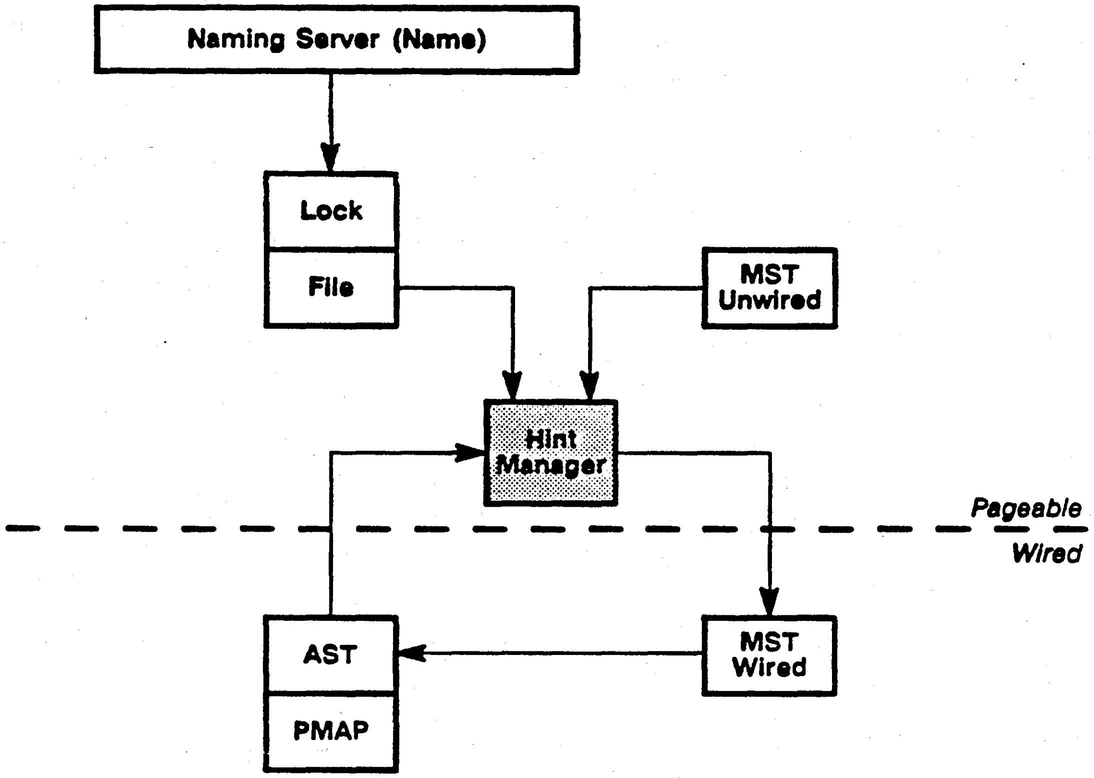


Figure 7-1. Relationship of Hint Manager to Other System Components

The hint manager performs the following functions:

- Initializes and unmaps hint files (hint_$init, hint_$shutdn)
- Adds a specified UID/internet address pair to a hint file (hint_$add, hint_$addu)
- Returns a list of internet addresses at which a given UID might be found (hint_$get_hints)

The next sections describe internal hint file structure and gives the details of hint manager functions.

7.1. Hint File Structure

Each hint file is actually composed of a hash table. At the head of this table is a version number that indicates the software version of hint file structure. The hint manager checks the version number during hint file initialization to make sure the hint file is in the current format.

The rest of the table contains 1 to 64 pointers to hint hash buckets. Each pointer is an index to the first bucket in the hash thread. Each hint bucket contains up to three hint items; each item is composed of a match_node field and a search list. The match_node field contains a node ID that identifies the UID about which location hints are to be kept. The search list contains space for a maximum of three internet addresses which are the hints about where the object presently exists. (An internet address consists of a node ID and the number of the network on which the node resides; see Chapter 24 for more information.)

Figure 7-2 illustrates the hint file format.

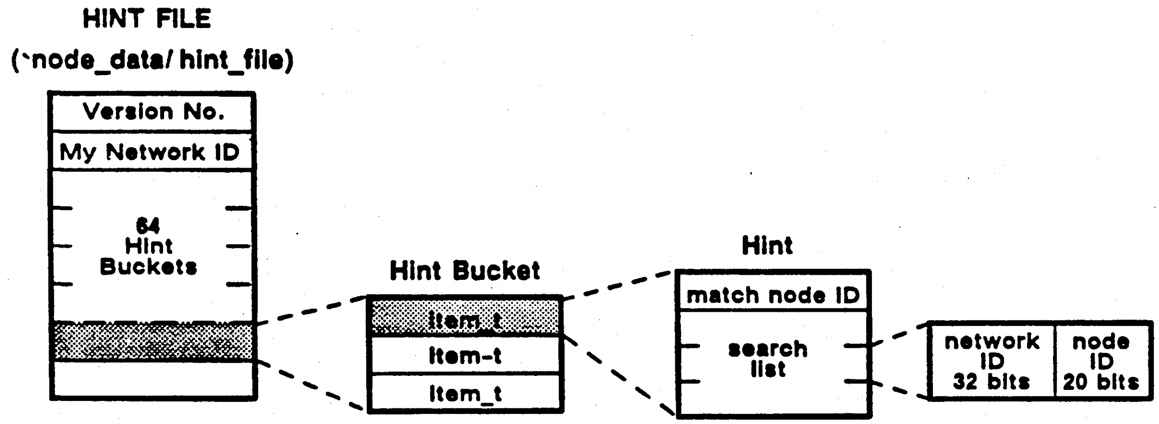


Figure 7-2. Hint File Structure

7.2. Hint File Initialization and Shutdown

The system initialization procedure os_$init calls the hint manager procedure hint_$init to create and initialize a hint file when a node is bootstrapped. The hint manager creates the file in /sys/node_data/hint_file. The hint_$init procedure initializes a hint file in the following sequence:

1. It checks to see if a hint file already exists on the node. If a hint file exists, the procedure passes its UID back to the caller.

2. If a hint file does not exist, hint_$init attempts to retrieve the UID of the node_data directory (by calling name_$resolve). If it can locate the node_data

directory, it creates a hint file (file_$create), adds the hint file name (name_$addu) to the directory subsystem, and calls the file manager to mark the file as permanent (file_$mk_permanent) so that it will not be deleted if the system crashes.

3. If the node_data directory does not exist, the hint manager calls the naming server to drop the entire node_data/hint_file name from the directory subsystem and calls the file manager to delete the file.

4. After the hint manager either locates or creates the hint file, it maps the file to the caller's process address space.

5. Next, the routine checks the version field within the hint file. If the version number indicates that the caller does not need a hint file, the routine returns.

6. If the version number is not the current version, the hint file is in an obsolete format and needs to be initialized. If this is the case, the routine truncates the file (by calling ast_$truncate) and sets the version field to the current version.

7. Finally, the routine sets up a pointer to the hint file and returns.

When a node is shut down, the os_$shutdown procedure calls the hint manager (hint_$shutdn) to unmap and unlock the hint file. The hint file persists across system shutdowns; the shutdown procedure simply unmaps the hint file, but does not delete it. A hint file is not deleted until the volume on which it resides is initialized with the INVOL utility. The system treats a hint file as it does any other object in the network.

7.3. Adding Hints to a Hint File

At present, the following AEGIS components add hints to hint files:

- The ASKNODE service (ASKNODE)
- The naming server (NAME)
- The FILE manager (FILE)

The following sections describe how the hint manager adds hints to a hint file and also describes the logic used by the above AEGIS components to add hints to a given hint file.

7.3.1. How the Hint Manager Updates a Hint File

The procedure hint_$add adds to the appropriate hint file the node ID of the node at which the specified (via UID) object is presently located. It carries out the following steps:

1. Checks to see if the hint file is mapped. If it is, the procedure hashes the node ID portion of the object's UID to obtain the index into the hint hash table.

2. Compares the node ID specified in the call to the node ID in the match_node field of the first item in the hint hash bucket. If the two node IDs match, the procedure has located the correct hint file.

3. Reads the first node ID in the search list. If this node ID is the same as the node ID to be added as a hint, then no addition to the hint file is necessary. If the second (or third) node ID in the search list is the node ID to be added, the procedure rearranges the list so that the desired node ID appears first.

4. If none of the node IDs in the search list match the node ID to be added, the procedure checks to make sure that the node ID to be added is not the local node's ID or the node ID where the object was created, and then adds the node ID to the hint hash bucket.

7.3.2. Hints from ASKNODE

The ASKNODE service is one of several information gathering services available to both users and the system. ASKNODE allows users and AEGIS to check the state of another node on the network. NETSTAT is an example of a program that calls this service.

The procedure asknode_$get_info retrieves from a given node information requested by the caller. One kind of information request concerns information about logical volumes, including the volume's directory entry UID, the number of free blocks on the volume, and the number of total blocks on the volume. If the request to asknode_$get_info is for information about a logical volume, the procedure adds to the hint file (via hint_$add) the node ID where the directory entry is located. Adding this hint enables the file system to locate the volume if it needs to access objects residing on it.

7.3.3. Hints from the Naming Server

The naming server adds hints to a hint file at two points:

- When it gets directory information for a specified pathname component (name_$get_entryu), usually as part of pathname resolution

- When it adds the name and UID of a specified node to the requestor's local copy of the network root directory

The naming server procedure name_$get_entryu searches a given directory for a specified pathname component, and if it finds the component, returns information about the component. To locate the directory, the procedure gets the hint list that hashes to the directory's UID and reads through it for hints about the directory's location.

When it finds the directory, the procedure checks the hint list. If the node ID where the directory exists is not the first item in the list, the procedure adds the hint to the list, unless the component is a link or the directory is the network root directory (//). It also adds a hint (via hint_$addu) about the component name if it locates the component within the directory.

The name_$add_node routine also updates hint files. This routine fetches a given node's local root UID and adds it to the caller's local copy of the network root directory (//). If the node UID it is adding belongs to a "disked" node, the procedure also adds a hint about its local root UID to the hint file.

7.3.4. Hints from the File Manager

Although the FILE manager is primarily a hint file user rather than a hint supplier, it makes a call to move a successfully used hint to the head of the list so that any subsequent access to the object will use the successful hint first.

7.4. Reading a Hint File

The hint_$get_hints routine, given an object's UID, returns to the caller a list of internet addresses at which the object might be found. If no hints are available, the routine prompts the calling component to search for the object locally.

The following AEGIS components call hint_$get_hints to read a hint file:

- The naming server (NAME)
- The active segment table manager (AST)
- The file manager (FILE)

The next sections discuss how the hint manager reads through the hint file data structures to find hints about an object, and describes how the AEGIS components mentioned above use the hint list in their operations.

7.4.1. How the Hint Manager Finds Hints

The function hint_$get_hints finds hints about an object's location, given its UID, by carrying out the following steps:

1. Extracts the node ID portion of the specified UID and examines it. If the node ID is 0, the object has a canned UID and will not need any hints. Since the node ID portion of the UID is zero, the hint manager assumes that the object is local; consequently, it does not need to use the hint file.

   In addition, if there is no hint file associated with the extracted node ID, the procedure returns.

2. Hashes the node ID to obtain the index into the hint hash table that points to one of the hint buckets.

3. Examines the match_node field within each of the hint items in the bucket until it matches the extracted node ID with the node ID in one of the fields.

If the routine matches the extracted node ID with a node ID in the hint bucket, it copies the internet addresses in the search list into its hint list buffer and returns this list, with the number of hints in the list, to its caller.

4. Regardless of whether or not it locates a hint, the hint manager always returns two special hints at the end of the lint list:

   - The internet address of the creator node; that is, the node at which the UID was generated. This hint is built from the UID and the local network number in cases where it doesn't appear in the search list.

   - The internet address of the local node.

   As a result, AEGIS components carry out a local search for an object *only* if if the remote search via the hint list fails. This sequence prevents a component from searching locally for a remote object.

7.4.2. Hint File Reading by the Naming Server

When the naming server needs to search a directory for a pathname component, it uses the hint manager to locate the directory. The naming server passes the directory's UID to the hint manager, and the hint manager returns a list of node IDs at which the directory might be found.

If the hint manager returns the local node ID, the naming server assumes that the directory is local and attempts to open it for the search.

7.4.3. Hint File Reading by the AST and FILE Managers

The AST manager reads a hint file when it attempts to get information (via ast_$get_info) about a remote object for which there is no local active segment table entry. See Chapter 10 for more information about referencing remote objects.

The FILE manager reads the hint file when it attempts to get information about an object, or when locking an object. See Chapter 5 for more information on this system component.

Chapter 8
The Naming Interface

The AEGIS system provides two ways to name objects. Users and user programs refer to objects using text-string names, called pathnames. The system itself refers to an object by its unique identifier, or UID.

Figure 8-1 illustrates the relationship between external and internal name spaces.

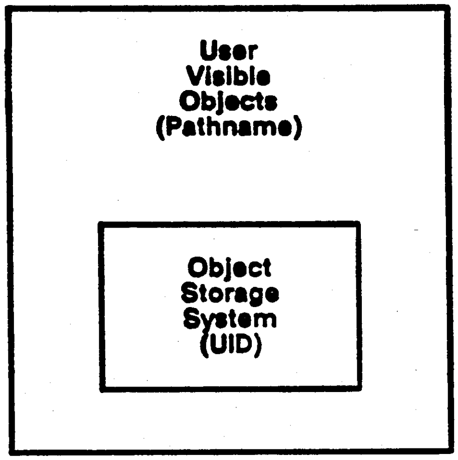


Figure 8-1. Object Naming in the AEGIS System

This chapter discusses how the system presently manipulates and stores pathnames and how it translates, or resolves, pathnames to UIDs.

8.1. Pathnames, Directories, and UIDs

The user or user program names an object by assigning it a text-string name. This text string is composed of a number of component names, each separated by a slash (/) character; together, these components make up the object's pathname.

The system stores pathnames in a network-wide name space that is structured as a multilevel directory tree. At its base is a network-wide root directory that catalogues each node. Branches of the tree represent directory objects; leaves on the tree represent particular objects.

A **directory** is an object (hence with its own UID) that contains a simple set of associations between component names and UIDs. The pathname gives the full route through the directory tree name space to the object; each pathname component provides a piece of that route.

An object's **absolute pathname** is an ordered list of component names that gives the path to the object starting from the network root directory (//). Because it begins at the root of the tree, the absolute pathname is valid throughout the entire network. Users and programs can also refer to objects by their relative pathnames; these names are abbreviated absolute pathnames *relative* to one of a number of points in the naming tree: the node entry directory, the working directory, or the naming directory. A pathname's **leaf** component is the last component in the pathname.

A pathname component can also be a **link** to a relative or absolute pathname; a link causes a jump from one point in the naming tree to another. Links are commonly used to refer indirectly to an object whose absolute pathname may change over time, and as shorthand notation for frequently used pathnames.

The AEGIS system names objects by UIDs, rather than by pathnames; UIDs are described in detail in Chapter 3. UIDs, unlike pathnames, are location-independent: they uniquely identify an object no matter where it resides, but do not always identify where an object is currently located.

8.2. Managers of the Naming Interface

The AEGIS services that implement the naming interface span both AEGIS kernel and user mode environments. The managers that maintain the naming interface are:

- The naming server
- The directory and network root directory managers
- The naming server helper (NS_HELPER) client software
- The naming server helper

This chapter briefly describes each of these managers, provides information on directory format, and explains how these managers interact to maintain the directory data structures and resolve pathnames.

8.2.1. Naming Server

The AEGIS component that links user level text-string naming to system-level UID naming is called the **naming server**. The naming server is a type manager that provides the following functions:

- Associates a pathname component with a UID or link.
- Inserts and deletes names from its database, the directory subsystem; this procedure is called **cataloguing** and **uncataloguing** an object.
- Translates pathnames to UIDs by consulting the directory manager.
- Updates the hint file on each node with information about the object's location (The hint file is discussed in detail in Chapter 7.)

8.2.2. The Directory Manager

The naming server uses the directory manager to create, delete, and operate on directories. Directory manager functions include:

- Creating and initializing directories
- Adding and removing directory entries from the directory data structures

The network root directory manager is responsible for maintaining a node's local network root directory (//). This directory differs slightly in format from the rest of the directories maintained by the directory manager. See the next section for more information on directory format.

8.2.3. The Naming Server Helper and Client Software

The naming server helper (NS_HELPER) is a server program that manages a master copy of the network root directory. It works in conjunction with the naming server, the directory manager, and the supervisor mode NS_HELPER client software to manage each node's network root (//) directory automatically as nodes are added to the internet, moved from one network to another, and removed from the internet. Once one NS_HELPER is notified of a node change on the internet, it propagates that change to the other NS_HELPERs in the internet.

The naming server helper is part of the user program environment, which is not documented in this manual. See the *System Administrator's Guide* for more information on this server. This chapter does discuss how the underlying NS_HELPER client software operates to maintain the network root directory.

Figure 8-2 illustrates the relationship between the naming server, directory, and NS_HELPER managers.

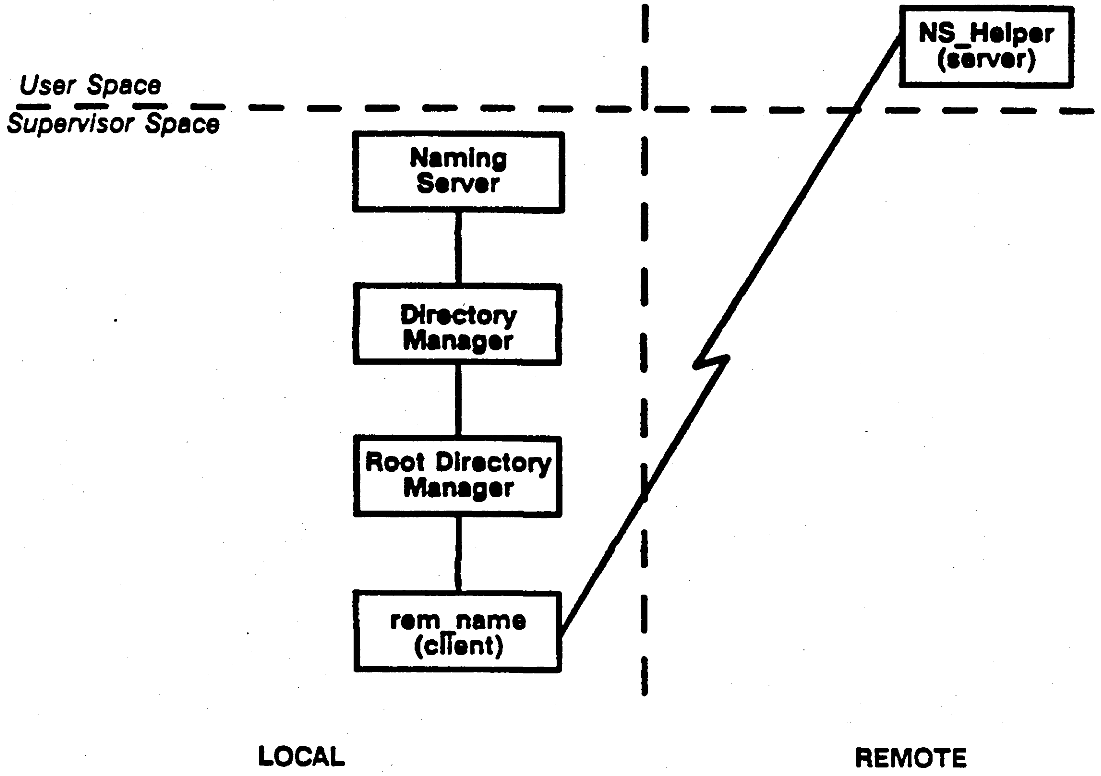


Figure 8-2. Naming Interface Managers

8.3. Format of a Directory

A directory consists of five parts:

- The directory header
- A linear list of directory entries
- An information block that contains miscellaneous information about the directory
- A hash thread table whose contents point to directory entry blocks
- Directory entry blocks that contain the actual directory entries

Figure 8-3 shows the layout of the directory.

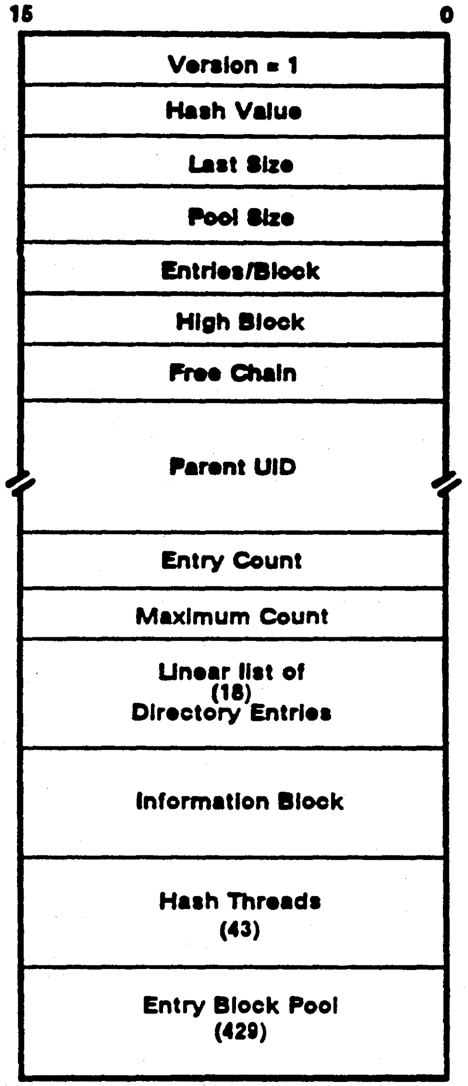


Figure 8-3. Directory Structure

8.3.1. Directory Header

The directory header defines the sizes of the directory's lists and tables. The naming server stores information about the directory here as it creates and deletes directory entries. Table 8-1 describes the fields within the directory header.

Table 8-1. Contents of Directory Header

Field	Description
Version number	Version number of the directory. Currently, this value is always 1.
Hash value	Size of the hash thread table. This value corresponds to the hash prime used to hash directory entries. Currently, this value is 43.
List size	Size of the linear list. Currently, this value is set to 18.
Pool size	Total number of entry blocks within the directory. Currently, this value is set to 429.
Entries per block	Number of directory entries within each directory entry block. Currently, there are three directory entries for each directory entry block.
High block	Number of the highest entry block that is currently in use.
Free chain	Number of the first available entry block. The naming server links available directory entry blocks together and writes a pointer to the first free block into this field.
Parent UID	UID of the directory in which this directory is catalogued.
Entry count	Number of directory entries currently catalogued in this directory.
Maximum count	Maximum number of entries the directory can hold (currently, this value is 1300).

8.3.2. Linear List

The linear list contains the first 18 directory entries. Since most directories contain less than 20 entries, the linear list provides a convenient way to manage small directories. When the naming server performs a directory search, it searches the linear list before any other data structures.

8.3.3. Information Block

The information block stores miscellaneous information about the directory. It currently contains:

- The UID for the default ACL to be applied to directories created underneath this directory.
- The UID for the default file ACL to be applied to all files created within this directory.
- Twelve spare words for future expansion.

Figure 8-4 shows the layout of the information block.

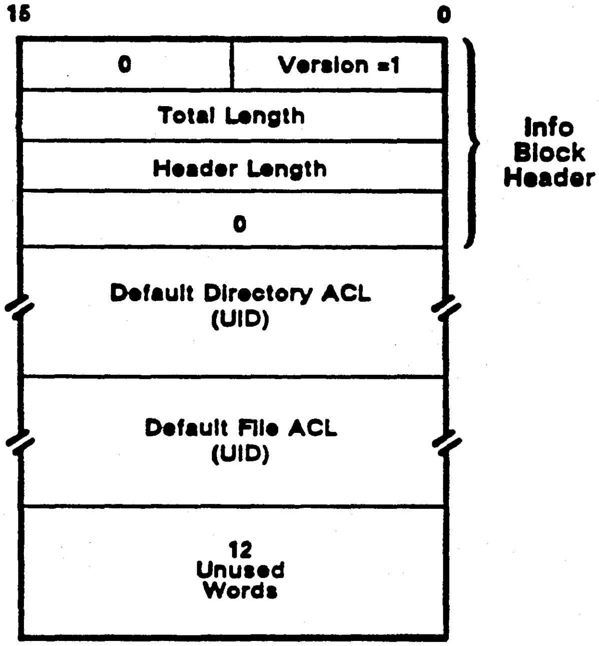


Figure 8-4. Information Block

8.3.4. Hash Thread Table

The hash thread table is an array of pointers to linked lists of directory entry blocks. Each entry in the table points to the first block in a chain of directory entry blocks. Each directory entry hash chain is doubly linked; the first two fields in each entry block point to the the next and previous blocks in the chain. The last block in the chain has a zero forward pointer. If an entry in the hash thread table contains zero, there are no directory entry blocks associated with that hash table entry.

The size of the hash thread table is determined by the hash prime value in the directory header. Presently, this value is 43. Figure 8-5 shows how a hash table entries *thread* to chains of directory entry blocks.

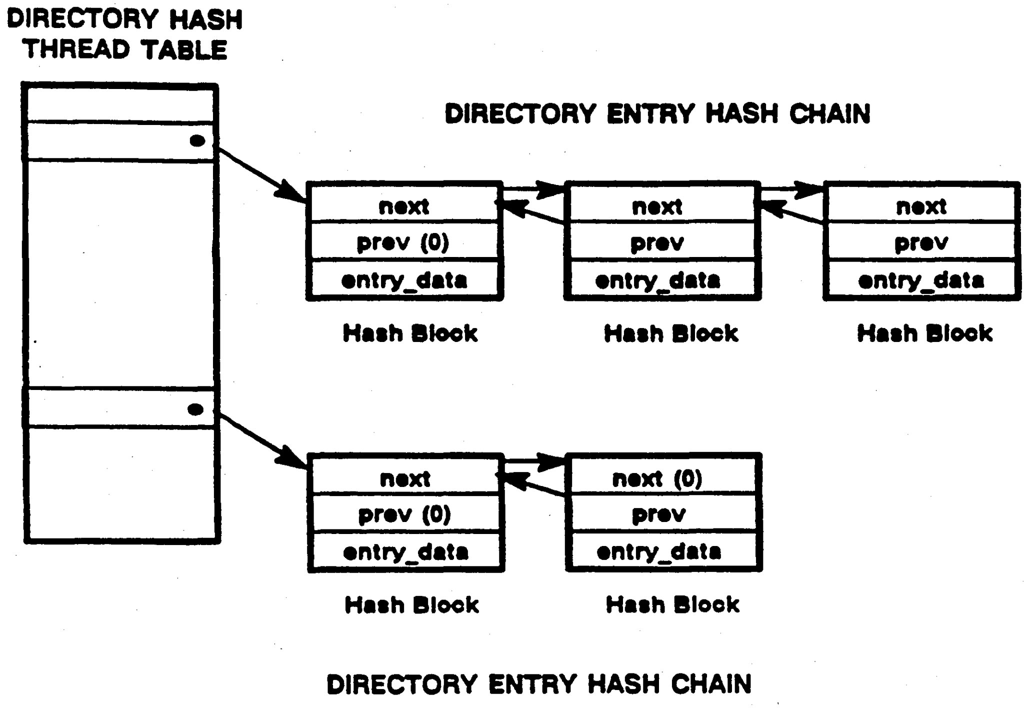


Figure 8-5. Threading to Directory Entry Blocks

8.3.5. Directory Entry Blocks

When user or system software catalogues an object in a directory, the naming server stores information about the object in the directory entry block. Each entry block can contain three directory entries or up to 144 characters of link text.

The fields within the directory entry block are:

- A pointer to the next block in the linked list of blocks.
- A pointer to the previous block in the linked list.
- The block type. The possible values for block type are:
  - 0 -- the block is free.
  - 1 -- the block contains an array of up to three entries.
  - 3 -- the block contains up to 144 characters of link text.
- A count that indicates the number of used entries within the block.

Figure 8-6 shows the structure of a directory entry block.

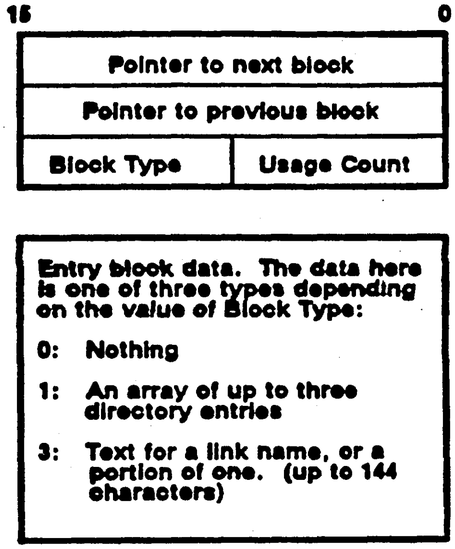

15
0
Pointer to next block
Pointer to previous block
Block Type
Usage Count
Entry block data. The data here is one of three types depending on the value of Block Type:
0: Nothing
1: An array of up to three directory entries
3: Text for a link name, or a portion of one. (up to 144 characters)

Figure 8-6. Directory Entry Block Format

The naming server creates a directory with all of its entry blocks free. As it uses blocks, the naming server links currently unused entry blocks together into a doubly linked list, called the free block list. The directory header contains a pointer to this list. When the naming server deletes directory entries within a block, and the block becomes empty, the naming server places the block back on the free block list.

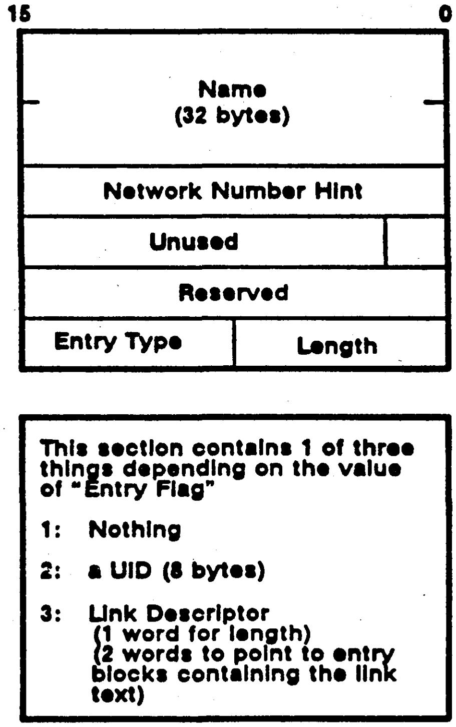
8.3.6. Entry Block Data Format

The actual data about the directory entries or link data exists in the entry block data portion of the directory entry block. Directory entry data consists of:

- The component name.
- The 32-bit network number of the catalogued node, if the directory is the local network root directory. The system uses the network number in two cases:
    - The file system uses it to locate objects in the internet (see Chapter 7)
    - Pre-internet AEGIS system services that take node IDs as arguments instead of internet addresses use it to build the internet address (via the dir_$find_net routine, which searches through the network root directory).

- The :ype of entry, which can be:
    - 0 – Not in use.
    - 1 – a UID.
    - 3 – a link descriptor. The first word gives the length of the link text; the next two words point to the entry block(s) that :ontain the link text.
- The length of the component name.
- The four words of entry data, either the UID or the link text description.

Figure 8-7 shows how the data is structured.



This section contains 1 of three things depending on the value of "Entry Flag"

1: Nothing

2: a UID (8 bytes)

3: Link Descriptor
(1 word for length)
(2 words to point to entry blocks containing the link text)

Figure 8-7. Directory Entry Format

8.4. Directory Operations

The naming server performs the following functions on directories:

- Creates and deletes directories as well as names, links and files within the directory

- Calls the directory manager to retrieve information about directory entries, naming and working directories

- Salvages the directory if it becomes corrupted

8.4.1. Opening Directories

Naming server routines that need to open directories (for example, name_$resolve, name_$addu, and so on) call the naming server internal procedure open_dir. This routine performs the following functions:

1. Tries to lock the directory. If the caller is the file server, the routine does not retry the directory lock operation. (see Chapter 6 for information on lock retries.)

2. Reads the directory's ACL to determine if the caller has the proper rights to perform directory operations. The open_dir routine also checks the *systype* field in the VTOCE header to determine that the object being opened is indeed a directory.

3. Makes sure the directory isn't a directory that is *permanently* mapped. These directories are:

   - The node relative/disk entry directory (/)

   - The /com directory

   - The working directory

   - The naming directory

   If the directory is not one of the above directories, open_dir calls mst_$maps to map the directory to the caller's address space. The map routine returns a pointer to the directory.

4. Checks the directory's version number against the directory that the returned pointer indicates. If these values do not match, the naming server returns "bad directory" error status.

8.4.2. Closing Directories

The internal procedure close_dir closes a directory by:

1. Ensuring that the directory should really be unmapped; that is, ensuring that the directory is not one of the special directories.

2. Calling mst_$unmap_priv to unmap.

3. Calling file_$unlock to unlock the directory.

8.4.3. Adding Entries to Directories

A naming server routine (add_name, add_link) that adds an entry to the directory calls the directory manager to add the new information to the directory data structures. The directory manager performs the following steps:

1. Searches the linear list for an unused entry; if a slot in the linear list is available, the routine can then write the directory entry to the list.

2. If the linear list is full, the routine must add the new entry to a hash block chain. The first step in this procedure is to hash the name; the result is an index into the hash thread table.

3. Searches the chain of blocks on that hash thread for an empty entry. If it finds an empty block in the chain, it writes the directory information to that block.

4. If it does not locate an empty block, it must allocate another entry block and thread it to the chain. It first checks the free block list; if no free blocks exist, the directory manager takes a block from the pool of available blocks.

5. When it has obtained a new entry block, the directory manager:

   - Initializes the block

   - Adds the entry block to the start of the hash chain that corresponds to the index into the hash table

   - Adds the new entry to the directory entry block as the first entry in the entry data section

6. The routine then adds the block to the head of the hash chain.

8.4.4. Searching Directories

Any internal naming server routine that must search the directory data structures (name_find, find_uid) does so in the following order:

1. The routine first searches the linear list. If it cannot locate the entry there, it hashes the name for which it is searching, thereby obtaining the index into the hash thread table. The hash thread in the table points to the first block in the hash chain for that hash value.

2. Starting with the first block, the routine searches all three entries within each block in the chain until it either finds the desired entry or reaches the end of the chain. If it reaches the end of the chain, the routine returns the message "not found".

8.4.5. Managing The Network Root Directory

The naming server, in conjunction with the user-mode NS_HELPER, carries out automatic cache management of a node's local network root directory (//).

Periodically, the NS_HELPER broadcasts its presence to nodes on the local ring. Nodes that receive this broadcast will perform the following automatic operations on their network root directories:

- When the naming server is called to search for an entry in the network root directory, either by name (via name_$get_entry_u) or by UID (via name_$find_uid), it will ask the NS_HELPER to locate the entry if it cannot find it in the // directory. The naming server calls the NS_HELPER using the NS_HELPER client software; if NS_HELPER returns the entry, the client naming server catalogues the entry in its local network root directory and returns successfully to the caller. Similarly, the // directory is used as a means of mapping node IDs to network numbers to form a complete internet address (dir_$find_net).

- If the naming server is attempting to resolve a pathname (via name_$resolve) and it cannot locate a node's entry directory, it validates the entry against the corresponding entry in NS_HELPER's master root directory, and if invalid, recatalogues the entry correctly in //. The naming server then tries to resolve the pathname again using the newly cataloged entry.

- When a call to name_$resolve specifies an absolut pathname (//node_name/...) the naming server always validates the entry *node_name* against the NS_HELPER database to ensure that it does not return obsolete information.

8.5. Pathname Resolution

Pathname resolution is a major naming server function. Since users see only pathnames, they are constantly using the pathname resolution mechanism; thus, this algorithm is very performance-sensitive. Currently, the naming interface uses the translation sequence illustrated in Figure 8-8:

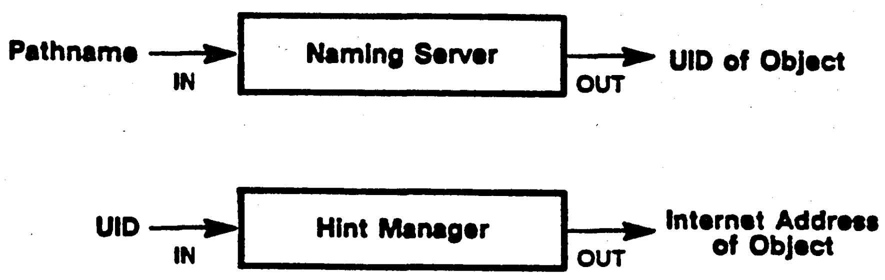


Figure 8-8. Current Resolution Sequence

8.5.1. Interaction of Naming Server and Hint Manager

Because AEGIS is a distributed system, there is no global state information kept about the location of objects in the network. Instead, the system keeps object location information on a dynamic basis through the hint manager. The hint manager is a system component that records the location of objects that various other AEGIS system components have tried and succeeded in locating. It stores the location information it receives in the hint file ('node_data/hint_file) on each node.

The naming server plays a large role in updating the hint manager for the location of objects. When the naming server locates an object as part of its name resolution sequence, it passes the object's location (the ID of the node on which it resides) to the hint manager, who adds it to the hint file. If another software component requests access to that same object and presents its UID to some AEGIS component, such as the file manager, the hint manager will have the correct location information for the object so that the file manager does not have to search for the object itself.

The hint manager is described in greater detail in Chapter 7.

8.5.2. Resolution Sequence

The naming server carries out the following steps when it must resolve a pathname:

1. Examines the pathname for its starting point, which can be one of the following points in the naming tree:

    - The network root directory (//)

    - The disk entry (node-relative) directory (/)

    - The current working directory

    - The current naming directory

2. Assigns the starting directory to "current directory," and searches the current directory for the pathname's next component and gets its UID (by calling name_$get_entryu)

3. Assigns the UID of the found entry to "current directory" and repeats the component search until no more pathname components are available

4. Returns the UID of the last component name (the leaf) to its caller

At each component search, the naming server decides if a hint about the component should be added to the hint file, and calls the hint manager to add the hint if necessary.

Chapter 9
Virtual Address Space Layout

Virtual address space is the virtual memory that PROC1 and PROC2 processes use to map objects, store data, and run programs. The size and layout of virtual address space differs from system to system. DNx60 nodes support a virtual address space of 256 megabytes, while the other nodes support a 16-megabyte address space. The next sections explain how virtual address space is apportioned for 16-megabyte and 256-megabyte systems.

9.1. Virtual Address Space on 16-Megabyte Systems

Virtual address space on 16-megabyte systems is composed of:

- Trap and PROM pages (1 segment)
- User global address space (63 segments)
- User private address space (320 segments)
- Supervisor private address space (8 segments)
- Segments reserved to Apollo (64 segments)
- Supervisor global address space (32 segments)
- I/O address space (32 segments)

These areas of address space are divided into segments; each segment consists of 32 pages, while each page consists of 1024 bytes. Figure 9-1 illustrates how virtual address space is apportioned; the next sections describe each area.

9.1.1. Trap and PROM Pages

The first segment of virtual address space is reserved for the trap and PROM pages. The trap page consists of vector addresses that the processor hardware uses to handle hardware exceptions (which are generated by traps) and interrupts. An **exception** is an event detected by the hardware that changes the normal flow of instruction execution. An **interrupt** also changes the flow of execution, but is generated by system activity that occurs independently of instruction execution. An exception always occurs as the result of instruction execution. Hardware exceptions include bus errors, zero divide, and privilege violations.

When an exception or interrupt occurs, the hardware indexes, or **traps**, to the appropriate trap vector address in the trap page and uses this address as the next instruction to execute.

The chapter on fault handling describes traps, exceptions, and faults in more detail. The system initialization section describes how the PROM page is used.

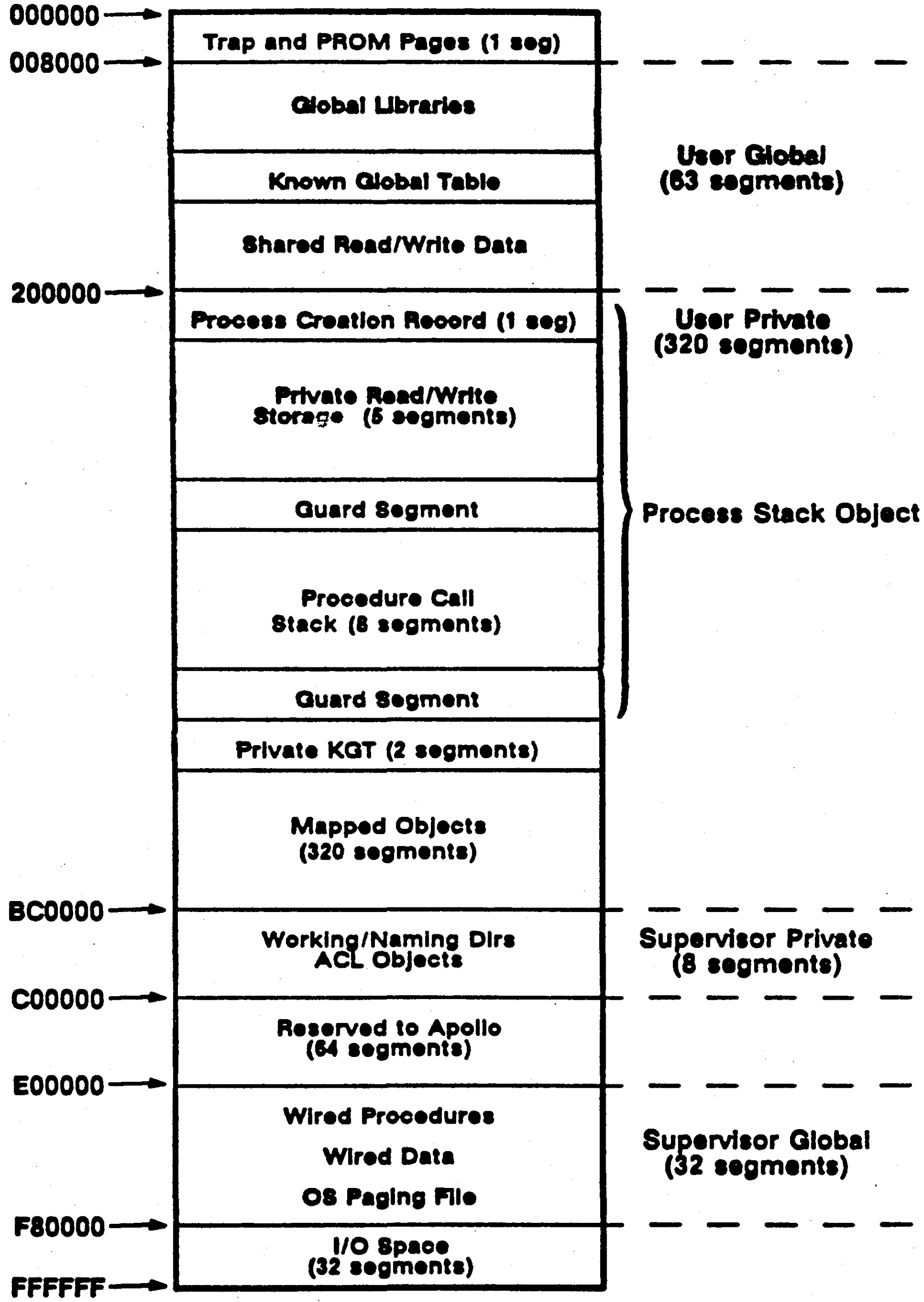

000000
008000
200000
BC0000
C00000
E00000
F80000
FFFFFF
Trap and PROM Pages (1 seg)
Global Libraries
Known Global Table
Shared Read/Write Data
Process Creation Record (1 seg)
Private Read/Write Storage (5 segments)
Guard Segment
Procedure Call Stack (8 segments)
Guard Segment
Private KGT (2 segments)
Mapped Objects (320 segments)
Working/Naming Dirs ACL Objects
Reserved to Apollo (64 segments)
Wired Procedures
Wired Data
OS Paging File
I/O Space (32 segments)
User Global (63 segments)
User Private (320 segments)
Process Stack Object
Supervisor Private (8 segments)
Supervisor Global (32 segments)

Figure 9-1. Virtual Address Space (16MB Systems)

9.1.2. User Global Space

User global address space contains virtual memory that is shared among all user processes. It stores the global libraries (read-only code and data) the global known global table (KGT) and shared read/write code and data; for example, the stream file control blocks (SFCBs) used by the stream manager.

9.1.3. User Private Space

User private address space is the region of virtual memory assigned to a user process when it is created. The process uses this section of virtual address space to access objects and run programs. Thus, the actual contents of user private address space differs for each process. User private address space contains:

- The **stack object**, which is mapped at the beginning of private address space. The stack object provides the backing storage for the user process's pageable stack and contains the process creation record, an area for private read/write storage, and an area for the process's procedure call stack. Management of the stack object is described in greater detail in Chapter 16.

- The **private known global table**, which contains the entry points to libraries installed dynamically into user private address space (rather than into global space at system initialization.) Unlike the global KGT, which is accessible to all processes, these private KGTs are private to each address space; other processes cannot gain access to them.

- Any programs and data that the process has mapped; the system reserves 320 segments for the per-process mapping of objects to virtual address space.

9.1.4. Supervisor Private Space

Supervisor private address space is per-process virtual memory that can only be accessed from supervisor mode; that is, when a process is running in the kernel. The system uses this address space to store per-process entities that it wants to protect user-mode programs from accessing. For example, the naming server uses this address space to map per-process working and naming directories. In addition, it stores directories here temporarily during pathname-to-UID resolution (see Chapter 8 for further details on pathname resolution.) The ACL manager also maps access control lists to supervisor private address space. Level 2 processes must call the system, via an SVC trap instruction, to gain access to objects in supervisor private space. Chapter 19 explains traps to supervisor mode in more detail.

9.1.5. Supervisor Global Space

Supervisor global address space contains the procedures and data that the AEGIS system uses to run and is the global address space that all level 1 processes share. There are three kinds of storage in supervisor global space: the paging file, whole cloth pages, and wired run file converter (RFC) image pages.

9.1.5.1. The OS Paging File

The OS **paging file** stores the pageable sections of the operating system, and is where AEGIS resides when it is running. At present, its size is 352 pages. The system initialization routine maps the paging file to MSTs as it would any other piece of pageable code. The system pages in and out of the OS paging file rather than from RFC image.

(The run file converter (RFC) is the pre-boot time loader. It creates an absolute bootable image from the compiled and bound AEGIS system (aegis.bin). The AEGIS RFC image is compiled and bound, but run-time initialization of global variables has not yet occurred.)

9.1.5.2. Whole Cloth Pages

Whole cloth pages are wired virtual address space *holes* that the system initialization routine creates during the RFC bind procedure to hold subsequently created system data structures. There are no objects behind whole cloth pages. Consequently, if a process generates a page fault to a whole cloth virtual address, the system will crash. However, the initialization procedure makes entries for these pages in the MST to prevent other processes from attempting to allocate these holes in virtual address space. When the system needs to use a whole cloth page, it allocates the page and, on reverse-mapped systems, installs it into the memory management unit (MMU). Many of the system's data structures occupy whole cloth pages. For example, the ast_$init procedure takes the number of pages it needs for the AST from whole cloth storage and installs these pages into the MMU. Per-process supervisor stacks are also allocated from whole cloth storage.

9.1.5.3. Wired RFC Pages

Wired RFC pages contain the code and data brought into physical memory from the RFC file during the system bootstrap sequence; for example, the wired pages for the AEGIS kernel code. Wired RFC pages are similar to whole cloth pages because they are wired only in the MMU. However, they are unlike whole cloth because they contain data that also exists in pages in the RFC file.

9.1.6. I/O Address Space

This region of address space contains the status registers and data structures for the MMU and for the I/O devices connected to the node. It also contains the I/O map, which supplies physical addresses for direct memory references by the ring/disk controller, display, and other DMA I/O devices.

9.2. Virtual Address Space on 256-Megabyte Systems

Virtual address space on 256-megabyte systems is composed of:

- The trap and PROM pages (1 segment)
- User global space (255 segments)
- User stack object (the first 16 segments of user private address space)
- User private address space (7680 segments)
- User special space for nodes with color displays (1 segment)
- Supervisor private address space (15 segments)
- Supervisor global space (240 segments)
- I/O space (16 segments)

The layout within each of these areas is the same as the layout on 16MB systems. However, virtual address space on 256-megabyte systems is allocated by **region**. Each region consists of 256 segments which in turn are allocated in order of ascending addresses.

User global address space occupies the first region of virtual memory; the first segment within this region is reserved for the trap and PROM pages and is used only by the system. Regions 1 through 30 are reserved for the per-process user and supervisor address spaces. Region 31 is allocated to user and supervisor global address space. Figure 9-2 illustrates the layout of 256MB virtual address space.

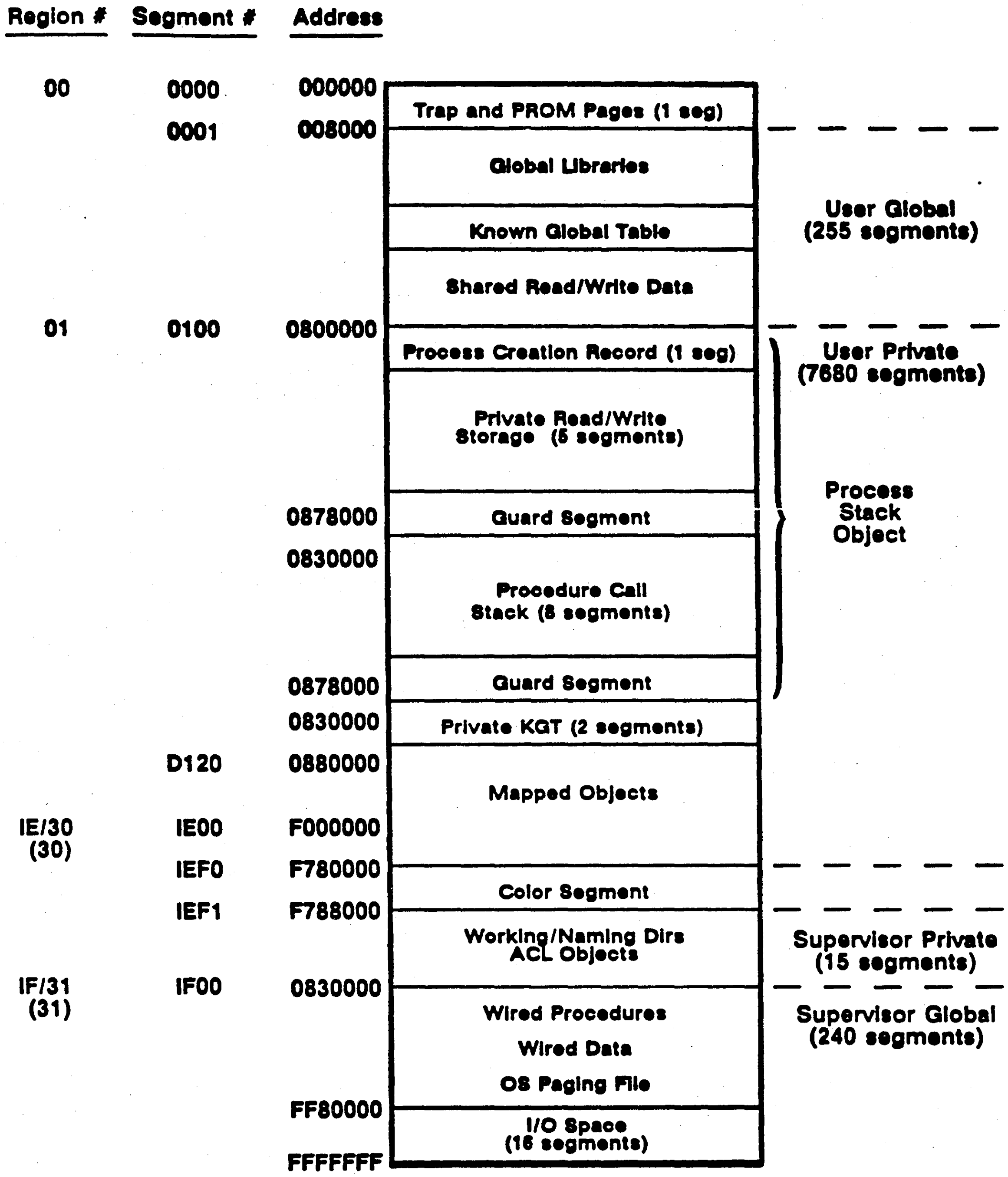


Figure 9-2. Virtual Address Space (256MB Systems)

9.3. Virtual Address Space Identification

When memory management software allocates per-process address space to a process, it assigns this address space an **address space identifier**, or ASID. There are 26 ASIDs; ASID 1 is used by the first level 2 process that runs during system initialization and by the display manager when initialization is complete. ASIDs 2 through 25 are assigned sequentially to any level 2 processes that the system (the PROC2 manager) subsequently creates.

The system assigns ASID 0 to user and supervisor global address spaces. It identifies pages of global address space by setting a hardware global bit in the MMU hardware page tables (page frame table entry or region register). Processes can only access the supervisor portion of ASID 0 while they are running in supervisor mode. Level 2 processes gain access to the contents of shared supervisor space via the SVC handler described in Chapter 19.

All level 1 processes run within the single supervisor global address space, while there is a one-to-one correspondence between a level 2 process and its ASID. Because level 1 processes share the same address space, they implicitly share objects.

Figure 9-3 illustrates per-process and global virtual address space allocation.

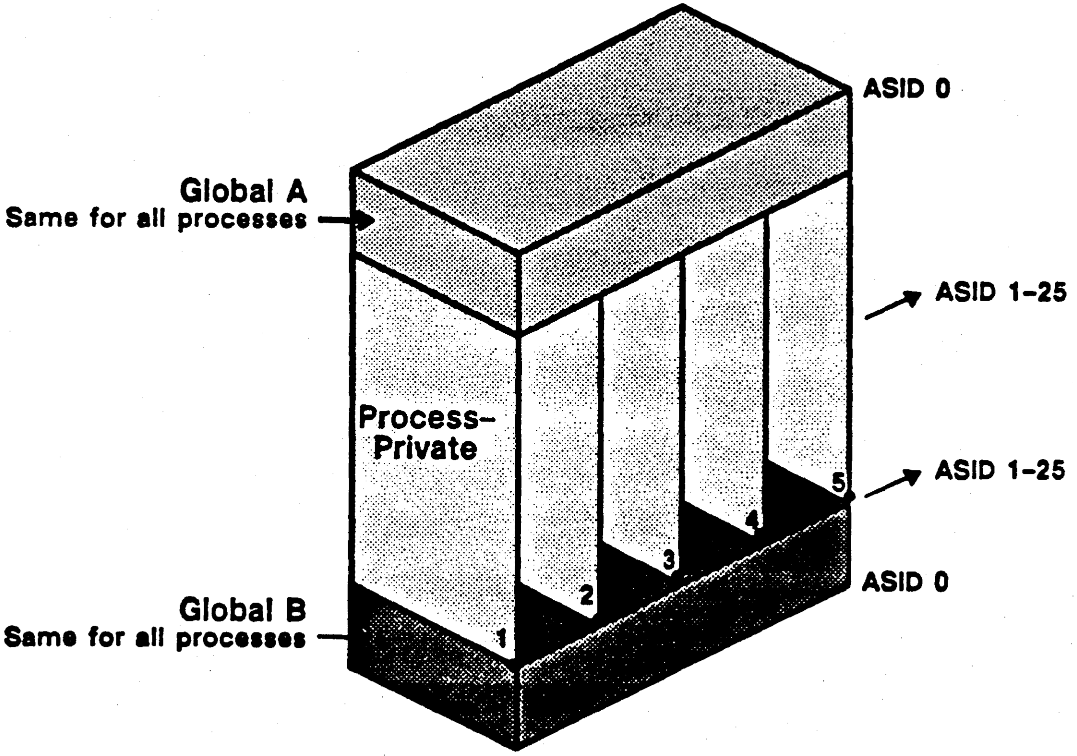


Figure 9-3. Per-Process Address Space

Chapter 10
Virtual Memory Management

The AEGIS system defines several types of address spaces:

- Object address space – the network-wide object name space, identified by 96-bit object addresses

- Virtual address space – the address space assigned to processes, identified by 32-bit virtual addresses

- Physical address space – the main memory that exists in each node, identified by physical addresses. The size of these physical addresses depends on the system in use.

- Disk address space – the 1056-byte disk blocks on a disk, identified by disk block addresses (DADDRs)

AEGIS memory management components make the following associations between address spaces:

- Mapping virtual addresses to object addresses, carried out by the mapped segment table (MST) manager

- Binding object addresses to physical addresses and disk addresses, carried out by the cached object storage system (AST, PMAP and MMAP managers)

- Translating virtual addresses to physical addresses, carried out by the memory management unit (MMU) manager and address translation hardware

This chapter describes object, virtual and physical address spaces (disk address space is described in Chapter 4) and introduces the memory management components that handle the address space associations.

10.1. Object Address Space

Object address space refers to the total object storage space available in the network. Objects within this address space are byte-addressable by their object addresses, which consist of:

- The 64-bit UID that identifies the object

- The segment number within the object

- The page number within the segment

- The byte offset within the page

Figure 10-1 illustrates the layout of object address space.

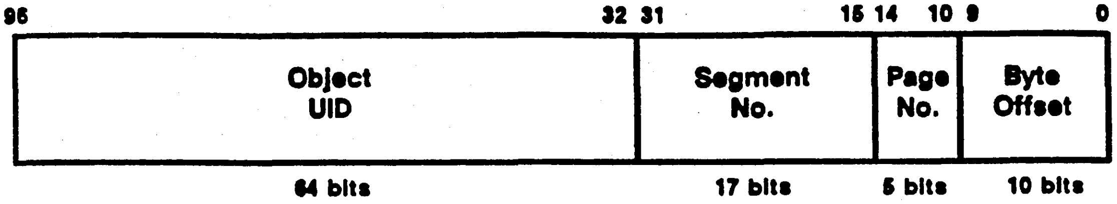


Figure 10-1. Object Address Space

10.2. Virtual Address Space

When the system creates a level 2 process, it allocates to that process its own separate and private address space, identified by ASIDs 2 through 25. The display manager is always assigned ASID 1, and level 1 processes run entirely in supervisor global space, identified by ASID 0.

The virtual memory management subsystem divides virtual address space into **segments**, which consist of 32 1024-byte pages. In addition, the virtual memory management subsystem divides an object into segments; each segment is composed of 32 consecutive object pages. While the object storage system views an object as a *collection of pages*, the virtual memory management subsystem regards an object as *some number of segments*, setting up a one-to-one correspondence between object segmentation and virtual address space segmentation. In object address space, it is the UID that identifies which object owns the object segment, whereas in per-process virtual address space, the ASID identifies which process virtual address space owns the virtual segment. This correspondence allows processes to map segments of objects within object address space into segments of their virtual address space. The mapping procedure associates an object segment with a 32-page segment of virtual memory. Figure 10-2 shows the relationship between object addreses and virtual addresses.

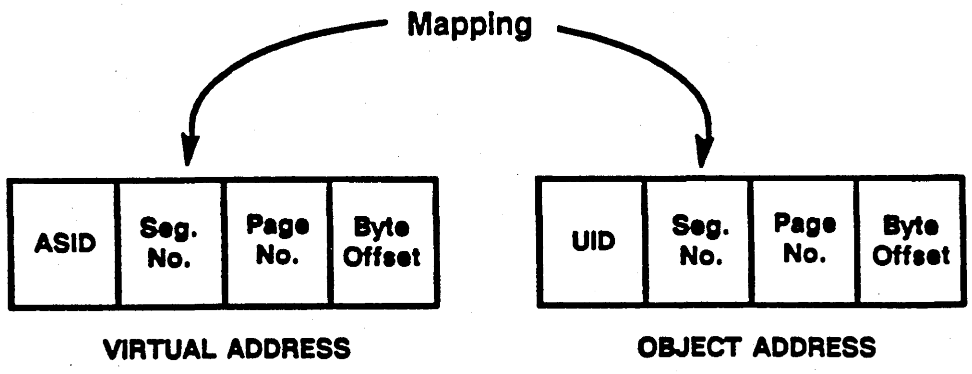


Figure 10-2. Relationship Between Object Addresses and Virtual Addresses

The format of a virtual address differs depending on the system; in addition, memory management software and harware view virtual addresses in different ways.

AEGIS virtual memory management software running on all systems except the DNx60 recognizes a virtual address that consists of:

- The virtual segment number within the ASID

- The page number within the segment

- The byte offset within the page

Virtual address space on DNx60 systems is divisible by region, segment and page. Consequently, memory management software running on DNx60 systems recognizes an additional field in the virtual address that represents the region number within the ASID. (A region consists of 256 segments of virtual address space.) Figure 10-3 illustrates this virtual addressing format.

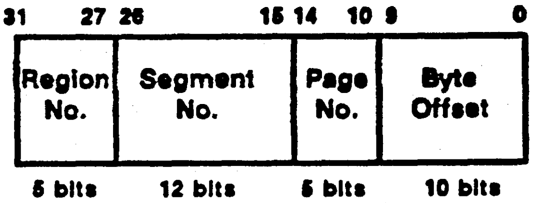


Figure 10-3. 256-Megabyte Virtual Address

10.3. Physical Addresses

Physical address space refers to the actual main memory within a node; it is byte addressable by physical address. The size of a physical address differs depending on the system. DNx60 systems, which have 16 megabytes of physical memory, use a 26-bit physical address, while other systems (which possess 4 megabytes of physical memory) use a 22-bit physical address. The fields within a physical address consist of:

- A **physical page number** (PPN) that identifies the page of memory

- The byte offset within the physical page

Figure 10-4 illustrates a physical address.

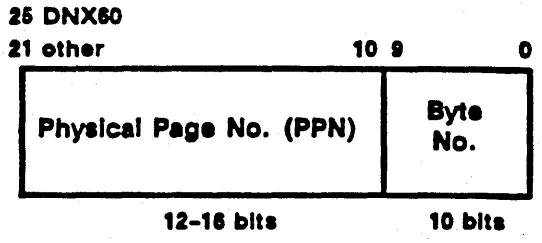


Figure 10-4. Physical Address Format

10.4. Mapping Objects to Process Private Virtual Address Space

The AEGIS system maps objects to process virtual address space in units of segments. The mapped segment table (MST) manager performs the mapping between virtual segments and object segments using a per-process data structure called the mapped segment table, or MST.

10.4.1. The Mapped Segment Table

Each ASID, and thus each process, has its own mapped segment table that lists the segments of process virtual address space and the object segments to which they refer. The MST is indexed by ASID and also by virtual segment number; each MST entry (MSTE) contains an association between an object segment and the virtual address segment to which that object segment is mapped. When an object segment is mapped to a virtual address, its MSTE contains the UID of the object that owns the segment, the segment's sequence number within the object, and other information. Figure 10-5 illustrates the object to virtual segment association in the MST.

The size of the MST depends upon the system. DNx60 systems allocate 7680 MSTEs for per-process user mapping; the other systems allocate 320 MSTEs.

10.4.2. The MST Manager

The MST manager carries out object-to-virtual address mapping and unmapping requests by filling in and clearing MSTEs. It returns information about an object mapped to process virtual address space by locating its MSTE given a virtual address.

The MST manager also initiates page fault resolution when its *touch* operation is called to fetch pages of an object segment. A **page fault** occurs when a process references a virtual address that is not resident in physical memory; that is, the association between virtual address and physical address has not been made. The MST manager reads the MST to identify (by the UID stored there) the object mapped to the virtual address, then calls the cached object storage system to locate the object, be it on a local disk or on another node in the network, and make the object to physical address association.

DOMAIN nodes support two types of MMUs: forward-mapped and reverse-mapped. On reverse-mapped MMUs, the MST manager must also call the MMU manager to load the physical page information into the MMU, which enables the virtual-to-physical address association. Chapter 13 describes how page faults are resolved in more detail. Chapter 11 fully describes mapped segment data structures for foward-mapped and reverse-mapped systems; Chapter 12 describes MST mapping and unmapping algorithms and how forward- and reverse-mapped memory management MMUs operate.

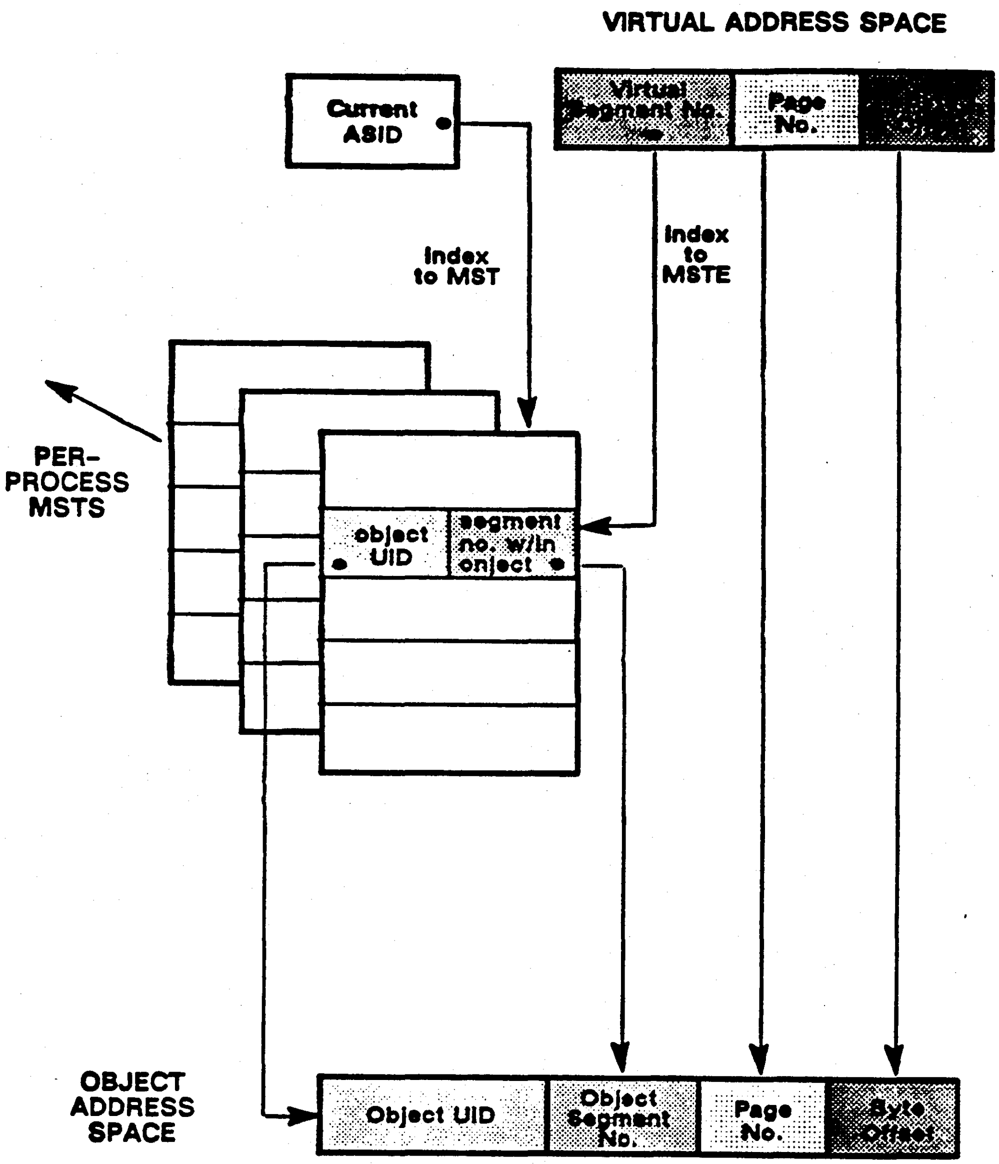


Figure 10-5. Virtual Segment to Object Segment Mapping via the MSTE

10.5. Mapping Objects to Global Address Space

Processes also map objects to global virtual address space. For example, the system initialization procedure maps system libraries into user global address space and the AEGIS supervisor-mode code into supervisor global space. These global address spaces are assigned ASID 0. User and supervisor global spaces do not have per-process virtual addresses; their addresses are the same for every process. In addition, user processes do not dynamically map objects to global space as they do to their own private address spaces. However, the global regions of virtual memory do contain objects mapped to their address spaces; consequently, they also require MSTEs to record those mapped objects.

When it builds the MST, the system initialization procedure allocates a range of MSTEs to user and supervisor global address spaces; the actual number of MSTEs allocated varies depending on the system. These MSTEs are wired at initialization, whereas per-process MSTEs are wired dynamically. Objects mapped to global address spaces use these MSTEs to record the virtual to object association. Thus, when a process references an address in user or supervisor global space, the MST used is the MST for ASID 0. Figure 10-6 illustrates the relationship between mapped segments for global space and per-process MSTs.

10.6. Binding Objects to Physical Address Space

The cached object storage system associates, or binds, object addresses with physical addresses (and disk addresses). The cached OSS consists of the active segment table (AST) manager, the page map (PMAP) manager, and the memory map (MMAP) manager. The cached OSS is part of the file system (see Chapter 3), but because it performs address space mapping, its components are described in this section.

Binding object addresses to physical address space consists of two functions:

- Activating the object segment by caching information about it into a table in physical memory called the active segment table (AST)

- Associating a page or pages of that segment with pages in physical address space; this process includes making the pages resident in physical memory

10.6.1. Activating Object Segments

When a process references a mapped object segment with an instruction, the MST manager calls the AST manager to **activate** the segment by storing information about it in the active segment table.

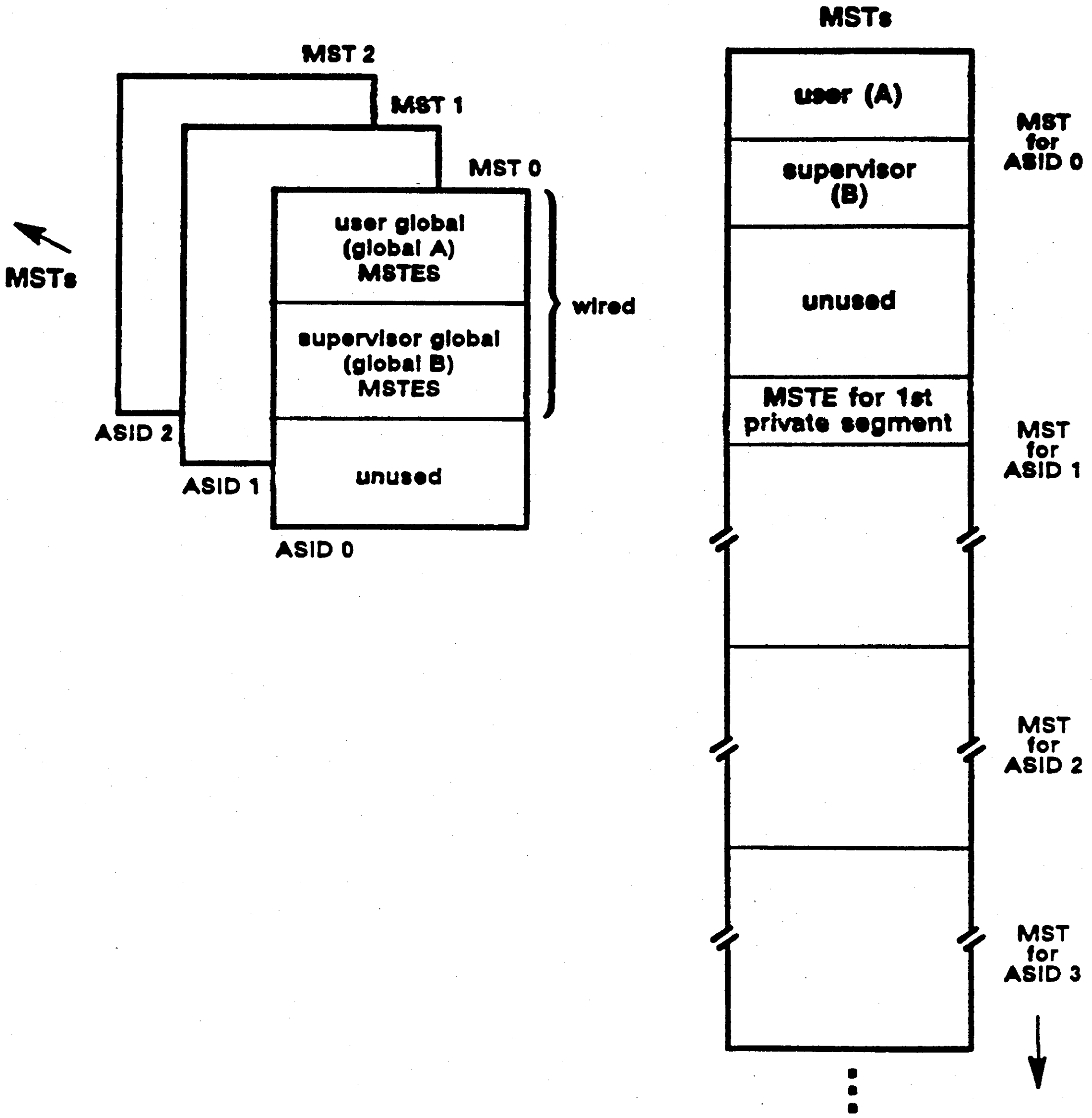


Figure 10-6. Global and Per-Process MSTs

10.6.1.1. The Active Segment Table

The active segment table is a wired, system-wide data structure that resides in supervisor global address space. To activate an object segment, the AST manager reads the VTOCE for the object that owns the segment and copies the object's UID and its attributes into an active segment table entry (ASTE). In addition, the ASTE stores dynamic information about the segment; whether it has been modified, its current length, and so on. Each ASTE is thus a cache entry over the VTOC; it caches the description of the object that is permanently stored in the VTOCE for the object, but also stores the most current information about the object while it is accessible to processes. The number of ASTEs present in the AST is a function of physical memory size; the system allocates 128 ASTEs for every one-half megabyte of memory.

Like the VTOC, ASTEs are indexed by hashing the object's UID. Consequently, two different segments owned by the same object will hash to the same ASTE hash thread (the start pointers in the AST header) and will therefore be linked to the same ASTE list. When the AST manager activates a segment, it passes back the index to the individual ASTE -- its ASTEX -- to the MST manager, which stores it in the MSTE for the object segment.

Because mapping and activation are independent procedures, the ASTEX on systems with reverse-mapped MMU hardware will not always be correct; it serves as a hint to the probable location of the segment's ASTE and so helps in memory management performance. Forward-mapped hardware sets up pointers between the ASTE and MSTs that map the ASTE, so in this case, the ASTEX is always correct. Chapter 11 describes reverse-mapped and forward-mapped MSTE and ASTE fields in more detail.

10.6.1.2. The Segment Page Map

Associated with each ASTE is a page map (PMAP) for the segment. When the segment is activated, the AST manager copies the segment's **file map** --the location of its pages on disk -- to the 32 entries in the PMAP. If any of the segment's pages are resident in physical memory, the PMAP also identifies the physical pages with which they are associated. Consequently, the ASTE identifies the object segment, while the PMAP locates each page of the segment on disk and, if resident, in physical memory. Figure 10-7 illustrates the relationship between MST, AST and PMAP data structures; Chapter 11 describes the PMAP format in more detail.

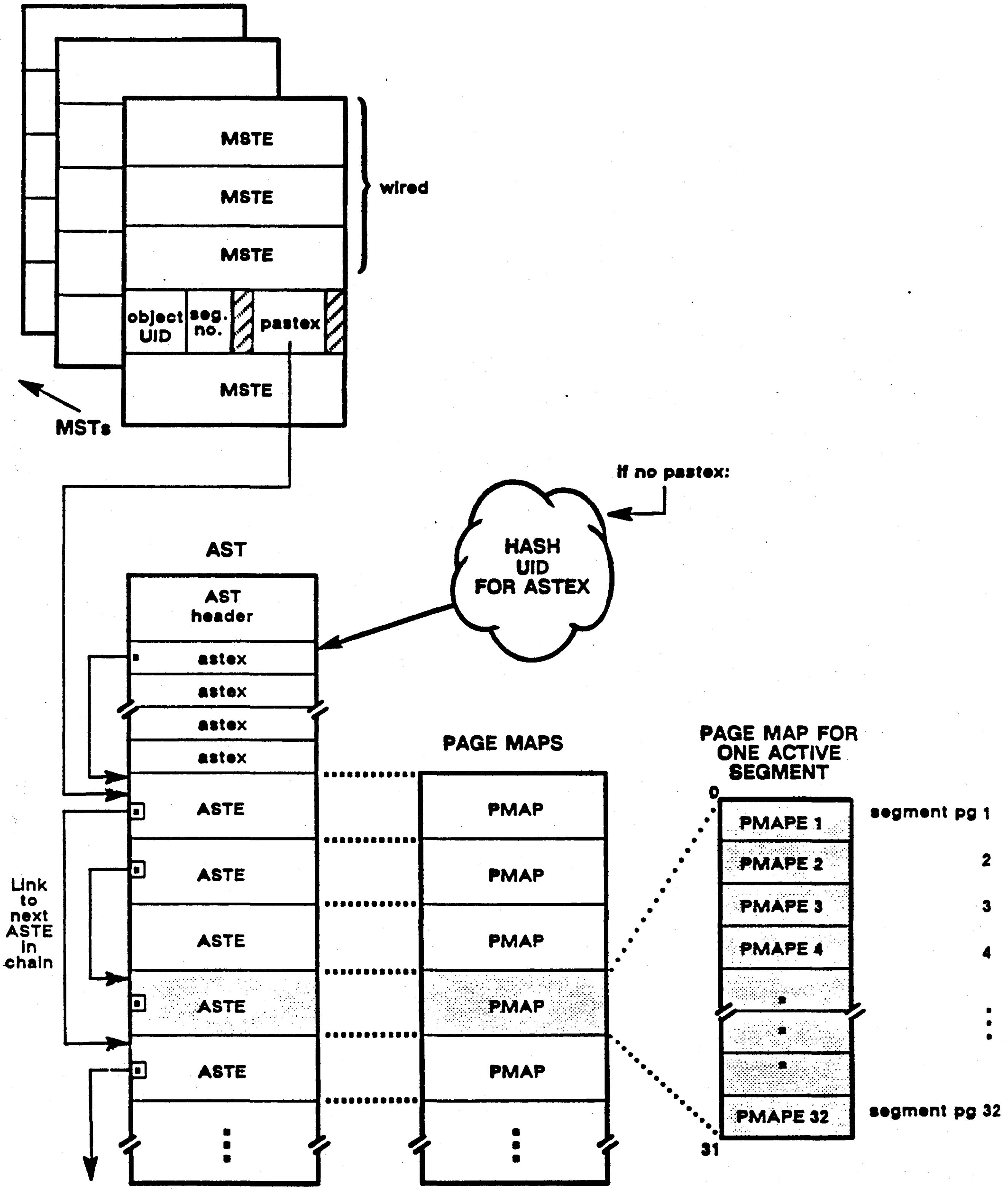


Figure 10-7. MST, AST, and PMAP Data Structures

10.6.1.3. The AST Manager

The AST manager's functions include:

- Maintaining the active segment table data structures

- Maintaining local AST-to-VTOC consistency

- Maintaining distributed cache consistency

- Performing operations on objects cached in the AST

The AST manager maintains the AST by activating and deactivating ASTEs for segments. When called to activate, the AST manager allocates free ASTEs from a free list or deactivates an ASTE already in use via a replacement algorithm. This algorithm is based on the last ASTE the AST manager replaced, how recently an ASTE has been used, and the number of segment pages that are resident in physical memory for the ASTE.

The AST manager also maintains AST-to-VTOC consistency by updating the VTOC to the state of the AST, either for a single ASTE or for the entire AST. In addition, if a process changes the object attributes of a local object cached in the ASTE, the AST manager also changes the attributes stored in the VTOCE. Chapter 12 gives more details about how the AST manager activates, deactivates, and replaces ASTEs, and how it maintains AST/VTOC consistency.

The lock manager calls the AST manager to maintain distributed cache consistency on file locks and unlocks. When it locks an object, the lock manager calls the AST manager to flush the active segments of an object from the AST if it ascertains (via a DTM attribute compare) that their data is obsolete. The flush is done by deactivating the ASTEs. In this way, the lock manager ensures that it is locking the most recent copy of the object as well as purging the distributed system of the object's obsolete versions.

When it unlocks an object, the lock manager ensures that the latest copy of the object exists on its home node. If the object is local, the latest copy exists in the AST cache; the system does not perform force-writes to disk on local objects.

However, if the object is remote, the lock manager initiates a **remote page-out** to the home node by calling the AST manager to purify all of the object's active segments from the AST; the AST manager in turn calls the PMAP manager to actually write the pages out to the remote node. Chapter 6 describes the interaction between the AST and lock managers in more detail.

The AST manager also carries out operations on objects and object segments on the FILE and REMFILE managers' behalf; these operations are described in Chapter 5.

10.6.1.4. Relationship Between Mapped and Active Segments

A segment can be mapped, but can be inactive. For example, a process can map a segment but make no references to it; because the AST manager hasn't received a message to retrieve the object information, no cached information about it exists. When the process requests the MST touch function, the AST is loaded with the object information stored in the VTOC.

A segment can be unmapped, but information about it will remain cached in the AST. While unmapping the segment breaks the binding of object address to virtual address, it does not affect the binding of object address to physical address; thus, unmapping a segment does not cause information to be removed from the AST.

The same object segment can be mapped to different MSTEs, but all the MSTEs use the same ASTE. For example, the shell program is mapped to an MSTE each time the user presses the shell key. But, because the AST is indexed by object UID, all processes that use the same object will index to the same ASTE.

10.6.2. Associating Object Pages with Physical Pages

The PMAP manager actually makes the object-to-physical association requested by the AST manager. This assocation consists of two steps:

- Calling the MMAP manager to obtain a physical page

- Retrieving the object page from local disk or from another node on the network

10.6.2.1. Allocating Physical Pages

The MMAP manager maintains a system-wide table called the memory map (MMAP) that contains, for each physical page of memory, dynamic information about its availability for use. The MMAP manager's role in page allocation is to:

- Find an available physical page to hold an object segment page by consulting the MMAP

- Allocate the page by filling in the appropriate MMAP entry

- Return its physical page number to the PMAP manager.

Main memory contains a finite number of physical pages that must be shared among a large number of object pages. Physical pages that have no object-to-physical-page associations are linked together on a free list in the MMAP. When the PMAP manager directs the MMAP manager to find a physical page for an object page, the MMAP manager first consults this list to find a free physical page to return. The free list, however, can be empty; consequently, the MMAP manager must "steal" physical pages that are still associated with an object page. As a result, an object page in physical memory has a page replacement status. A page can be:

- **Resident** -- the object page occupies a page in physical memory, but the physical page is available for replacement; that is, for use by another object page

- **Wired** -- the page is resident in physical memory and unavailable for page replacement

- **In transition** -- the page is in a transition state in the memory management process, and is therefore unavailable for page replacement until it is no longer in transition

Thus, to the MMAP manager, an *available* page (also called *replaceable*) is one that is not wired or in transition. However, pages also have pure and impure status, which further determines their availability for replacement. An **impure** page is an object page that has been modified in the course of its existence in physical memory; consequently, its contents must be written out to disk so that the copy on disk is current. The process of writing pages out to disk is called **purification** and is one of the PMAP manager's primary functions.

A **pure page** is an object page whose contents have not been modified during its existence in physical memory (or one which has been purified). Pure available pages are eligible for immediate replacement. Impure available pages have been modified and so must be purified before they can be reused. The MMAP manager is responsible both for locating pure available pages and returning the PPNs of impure available pages to the PMAP manager for purification. The PMAP manager, in turn, helps in page replacement by purifying the impure available pages, making them eligible for replacement.

Although the PMAP manager calls the MMAP manager to find physical pages to purify, it also periodically (or on request) scans the PMAP for each active segment and purifies any of its resident pages. If specified, it will also flush these resident pages from physical memory by breaking their object-to-physical (and the virtual-physical) page associations.

10.6.2.2. Fetching Object Pages

When the MMAP manager returns an available page, the PMAP manager calls the disk management subsystem to fetch the object page from local storage, or calls the network manager to retrieve the page from another node in the network if the object is remote. Once the page is written into physical memory, The PMAP manager then sets up the PMAPE for the page with the object to physical association. Although the page is then resident in memory, the virtual-to-physical address association must be made before it can actually be used. On DNx60 systems, the address translation hardware reads the PMAPE to make the virtual to physical translation. On other nodes, PMAP returns the physical page number to the MST manager via the AST manager. The MST manager then loads the MMU with the virtual-to-physical assocation. This operation resolves the page fault that occurred when the object page was referenced and not found in physical memory.

10.7. Translating From Virtual to Physical Address Space

The memory management unit (MMU) is the hardware that carries out virtual address to physical address translation. The address translation mechanism that an MMU uses depends on the MMU model. The MMUs on DNx60 systems use a forward-mapped address translation mechanism; the MMUs on the other node models use the reverse-mapped scheme.

The forward-mapped address translation is so named because its data structures are organized in a tree structure; the MMU moves in a "forward" sequence from one table to the next without ever reversing direction. A reverse-mapped MMU, on the other hand, sometimes goes back to tables it has already referenced during the address translation process.

Although MMUs can be categorized as forward- or reverse-mapped, further variation between MMU models exists, such as the size and type of registers and caches and the size and type of data structures used. The next sections introduce the forward-mapped data structures for DNx60 nodes and the reverse-mapped data structures for the other node models.

10.7.1. Reverse-Mapped Data Structures

The reverse-mapped MMU uses two hardware devices in address translation:

- The page translation table (PTT), an array of 1023 physical page numbers that allows 1 megabyte of virtual memory to be mapped to physical memory at a time

- The page frame table (PFT), an array of entries that describes the 4096 physical pages of main memory

The AEGIS system implements the PTT and the PFT by assigning them virtual and physical addresses so that the memory management software can use them.

The page translation table (PTT) is a translation buffer that gives hints to address translation. The PTT contains 1023 entries, or enough entries to map one megabyte of virtual memory. Each PTT entry points to the start of a linked list of entries in the page frame table table (PFTEs). The chain of PFTEs consists of a circular list of all the virtual pages that hash to one physical page; each entry in the list describes a virtual-to-physical page association and stores protection and status information about the physical page. The last PFTE in the list has an end of chain EOC) bit set.

Reverse-mapped memory management hardware translates a virtual address to a physical address as follows:

When the hardware gets a virtual address to translate, it:

1. Takes the virtual page number in the virtual address and uses it as an index into the PTT

2. Uses the information in the PTTE to locate the first PFTE in the chain. If there is no hint in the PTT, the hardware hashes the virtual address to obtain an index directly into the PFT.

3. Examines the PFTE, searching for a virtual-to-physical address correspondence by checking the ASID and excess virtual page number (XSVPN) in the virtual address against the ASID and XSVPN stored in the PFTE. (On a global reference, only the XSVPN must match.) If the two values do not match, the hardware searches all the PFTEs in the linked list for a match. If the virtual address that the hardware is looking for is not in the list, and it has encountered the end of chain field twice, the hardware recognizes that no virtual-physical page mapping exists for the address, and signals a page fault to install the correct mapping.

4. If a match exists, copies the index of the matching PFTE to the PTTE and calculates the physical address by combining the offset given in the virtual address with the PPN in the PTTE; this operation will optimize performance on the next search. In addition, if a PFTE is used frequently, the hardware adjusts the PTTE to point directly to it.

The memory management subsystem also uses information stored in the PFTE for page replacement and purification decisions.

10.7.2. Forward-Mapped Data Structures

The MMU hardware on DNx60 systems divides per-process virtual address space into 32 regions. Each region is composed of 256 segments, and each segment is divided into 32 pages. The segment map (SMAP) keeps track of the 256 segments in one region. The 32 hardware region registers point to the 32 SMAPs; each SMAPE points to the segment's page map. The SMAP and PMAP data structures are allocated in memory in the same way as other objects in the system (except that they are not pageable). Figure 10-8 illustrates the relationships between the data structures in the DNx60 forward-mapped scheme.

The MMU hardware uses the information contained in the SMAP and PMAP tables during virtual to physical address translation. In brief, the forward-memory management hardware translates a virtual address to a physical address by:

1. Locating the SMAP to search from the region field in the virtual address

2. Locating the offset into the SMAP at which the segment exists from the segment number in the virtual address the address (which PMAP to search)

3. Locating the PMAP from the SMAPE

4. Locating the offset into the PMAP at which the PMAPE for the page exists from the page number in the virtual address

5. Obtaining the physical page number (PPN) from the PMAP entry (or taking a page fault if no PPN exists in the PMAPE)

6. Calculating the actual physical address within the physical page from the offset field in the virtual address

The memory management subsystem also uses these tables; the MST manager fills in the SMAPE with access information when an object is mapped to virtual address space; in addition, the MMAP and PMAP managers use fields in the PMAPEs during page replacement and purification.

Chapter 11 describes the fields within forward- and reverse-mapped data structures; Chapter 12 describes how the memory management software uses the information within these data structures.

10.7.3. The MMU Manager

The MMU manager's main function is to install and remove virtual-to-physical assocations from the MMU. The processor signals the MMU manager to update the MMU page tables (page frame or page) by page faulting; once the assocation exists, the processor takes control of the translation.

On forward-mapped systems, the MMU manager is also responsible for purging the translation buffer.

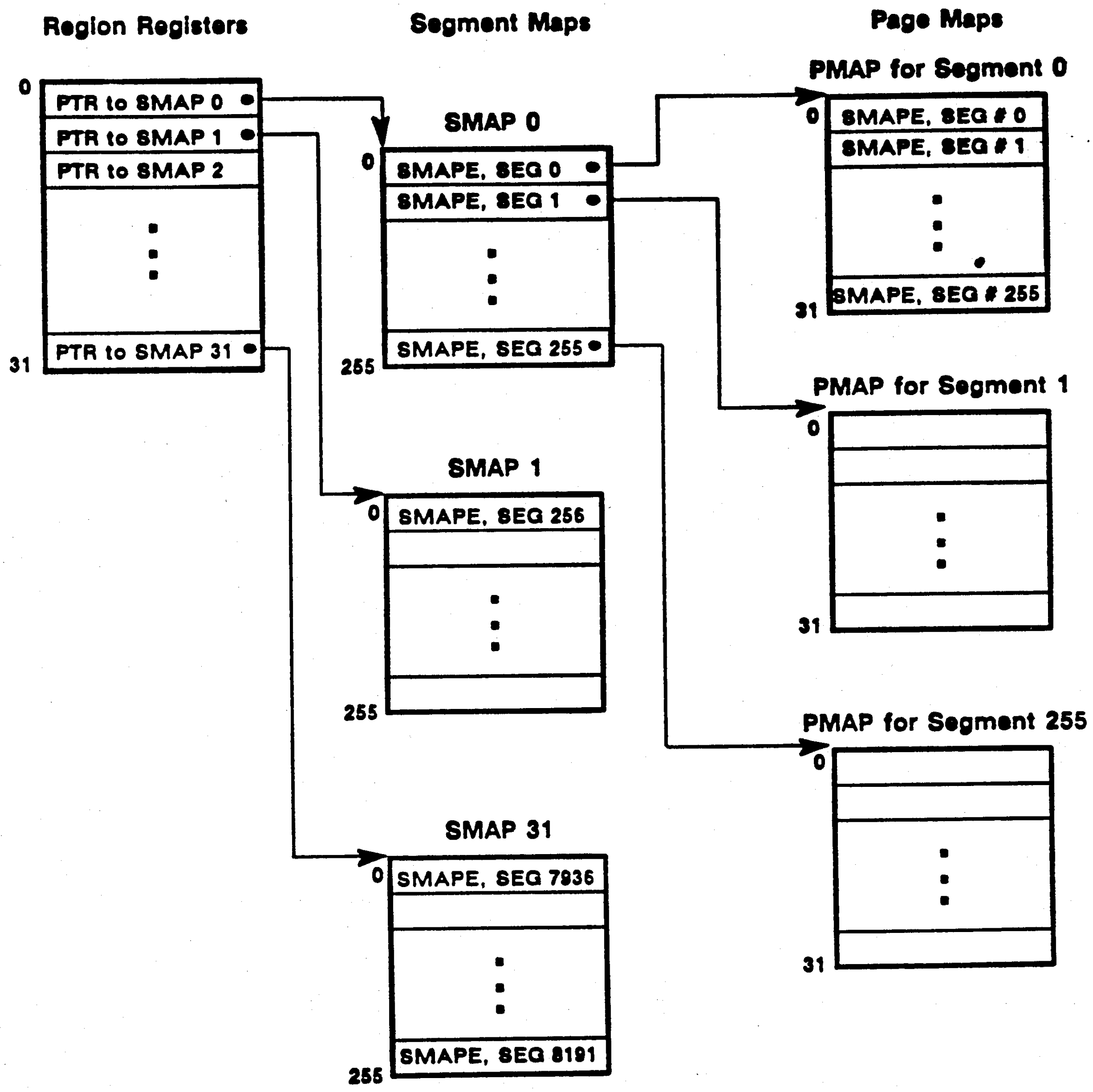


Figure 10-8. Forward-Mapped Data Structures

Chapter 11
Memory Management Data Structures

AEGIS contains both per-process and system-wide tables that thc virtual memory management subsystem uses to carry out its memory management functions. This chapter describes the memory management data structures, their relationship to each other, and the kinds of information they provide to memory management software and hardware.

The memory management data structures can be grouped into:

- Structures used in object segment/process address space mapping
- Structures used in segment activation and deactivation
- Structures used in physical page management

The next sections discuss the data structures that fall into these categories.

11.1. Mapped Segment Data Structures

The mapped segment data structures contain the object to virtual segment map, information about the object's location, and how the object can be accessed. The location of this information within the data structures depends upon the node's memory management hardware and the memory size. Nodes with reverse-mapped MMUs store all the information about the mapped segment in the MSTE for the object. Nodes using forward-mapped memory management and a large address space (DNx60) keep mapped segment information in the MSTE and in the segment map entry (SMAPE). The memory management hardware on forward-mapped MMUs partitions per-process SMAPEs into 32 SMAPs, each SMAP contains 256 SMAPEs. (User private address space is allocated 30 of these SMAPs.) However, the memory management subsystem views an SMAP as a per-process array of SMAPEs that corresponds one-to-one with the array of per-process MSTEs.

Figures 11-1 and 11-2 illustrate the mapped data structures for reverse-mapped and forward-mapped MMUs and shows the fields within the data structures. The next sections discuss the kinds of information kept in mapped segment data structures and where the information is located depending on the type of MMU hardware in use.

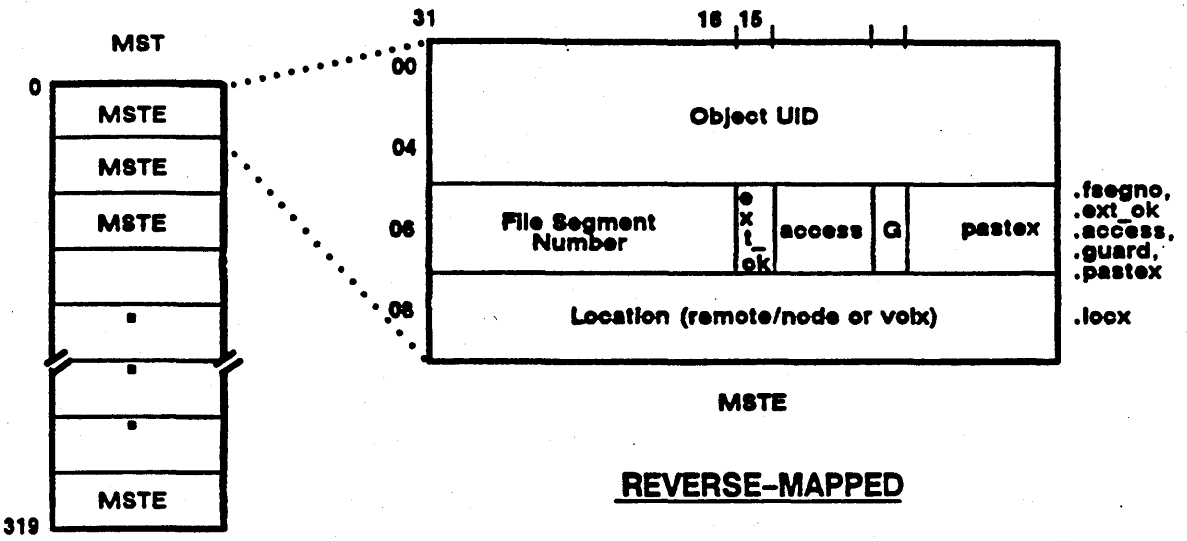


Figure 11-1. Reverse-Mapped MST

11.1.1. Object to Virtual Segment Association

The MSTEs for both MMUs contain the UID of the object that owns the segment (mste.uid) and the segment's sequence number within the object, called its file segment number (mste.fsegno).

11.1.2. Pointers to Other Structures

The MSTE contains a field to cache the index to the information about the object in the AST, called the probable ASTEX (mste.pastex). The MST manager fills in this field with information it obtains from the AST manager when it:

- Activates a segment for a mapped object
- Reads an already active ASTE for a segment being mapped

Because ASTEs are activated and deactivated independently of mapping and unmapping, the index to the active segment can change, Consequently, the PASTEX on reverse-mapped systems is a hint to ASTE whereabouts and exists to optimize memory mangement performance. If the MST manager has a valid PASTEX, the memory management function proceeds more quickly because the manager can avoid a hashing operation. If the PASTEX is incorrect, the manager must hash the object's UID to obtain the correct index into the AST.

MSTEs on DNx60 MMUs, however, contain a pointer to the physical address of the the segment's page map (mste.pmap_phadd). Because the ASTE and PMAP exist in a one-to-one correspondence, the cached ASTEX on forward-mapped systems is always correct.

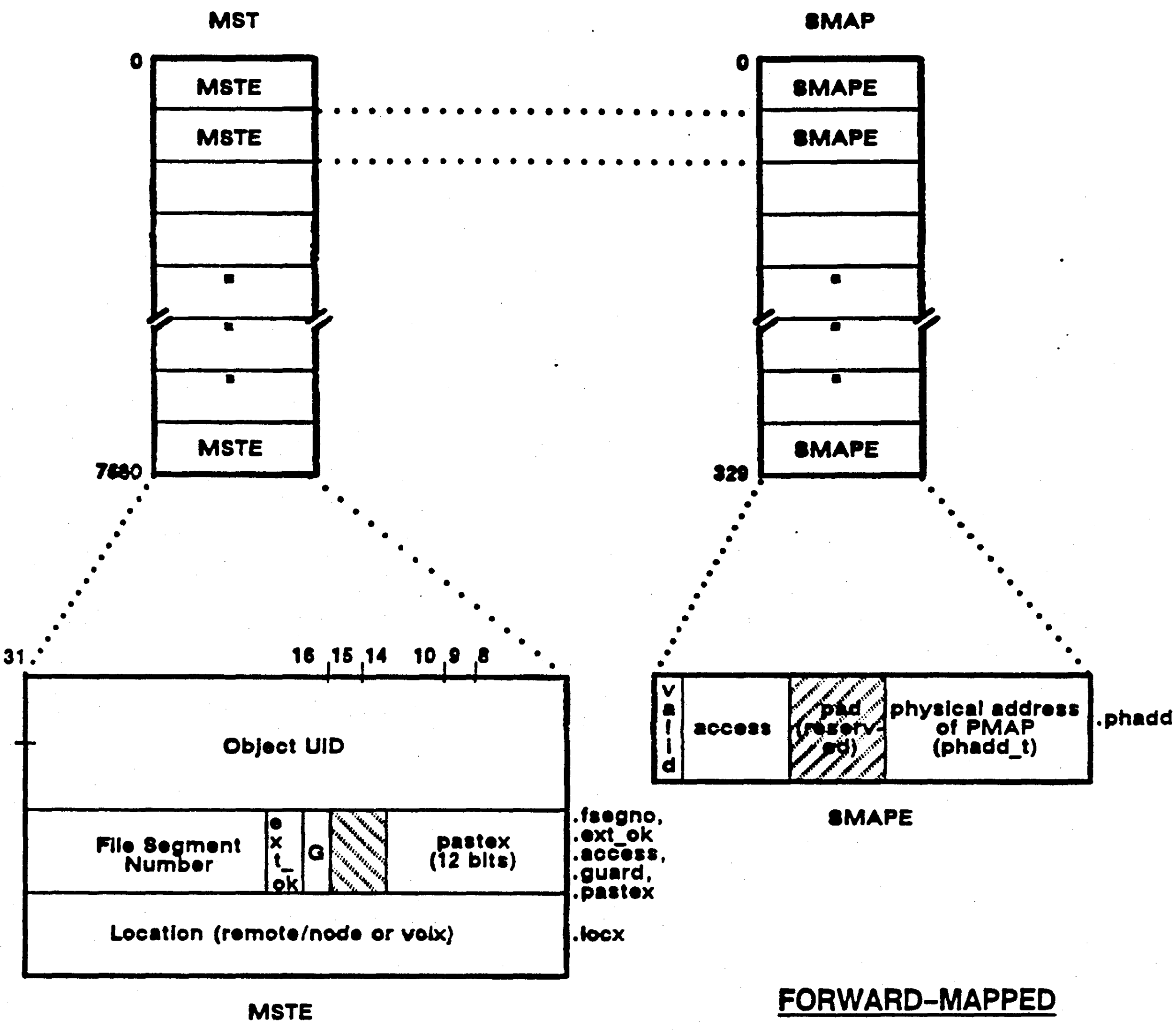


Figure 11-2. Forward-Mapped MST

11.1.3. Location Information

If the object is local, the MSTE contains the index to the volume on which the object resides (mste.volx). If it is remote, the MSTE's remote bit is set (mste.mste_remote), and the MSTE contains an index to the network number and the node ID of the remote node on which the object resides.

11.1.4. Access Modes

The processor and memory management hardware support access modes that define at the hardware level the access allowed to an object. These access modes are:

- No access (access_$nil)
- User-mode read access (access_$r)
- User-mode read/execute access (access_$rx)
- User-mode write/read access (access_$wr)
- User-mode write/read/execute access (access_$wrx)
- Supervisor-mode read access (access_$sr)
- Supervisor-mode read/execute access (access_$srx)
- Supervisor-mode write/read access (access_$swr)
- Supervisor-mode write/read/execute accesss (access_$swrx)

When a process maps a segment, the MST manager determines the access mode allowed for the segment from the process using the arguments passed in the call and the mode in which the process is running. It then fills in the access mode field in the MSTE (mste.access) for reverse-mapped hardware or the segment map entry (smape.access) for forward-mapped hardware. When a page fault brings the object page into physical memory, the MST manager takes the access rights stored in the MSTE and installs them in the MMU's hardware page table (the page frame table). Forward-mapped MMUs use the SMAPE directly, so no access rights loading is needed. The MMU address translation hardware uses the access rights as a protection check during virtual-to-physical address translation.

11.1.5. File Extension

The *file extension allowed* bit in the MSTEs (mste.ext_ok) controls whether or not a segment can *grow*; that is, whether pages can be added dynamically to the segment. For example, if a segment is mapped read-only, new pages cannot be added to it. When it is called to fetch, or *touch* a segment, (mst_$touch), the MST manager checks the access mode for the segment. If it is read-only, the manager sets this field before it calls the AST manager to activate the segment, and passes *not ok* as an input parameter to the AST manager's activation routine (ast_$activate).

11.1.6. Guard Bit

Guard segments are virtual segments mapped with *no access* rights. The AEGIS system managers use guard segments to protect important system data structures from becoming corrupted. For example, the process manager installs guard segments around the procedure call stack to prevent a stack overflow from corrupting the process creation record. MSTEs that map guard segments have the guard bit (mste.guard) set. The MST manager checks this bit during its touch operation. If the caller is attempting to touch a guard segment, the MST manager will return a guard fault.

11.1.7. Touch-Ahead Count

This field (mste.mste_touch_cnt) indicates how many successive pages can be fetched for a segment during one touch operation. Both the system and user-mode programs can adjust this value. The system initially sets the touch-ahead count to 4 pages, but it can be set to touch a maximum of 32 consecutive pages (the entire segment) at one time. The stream manager is an example of a user-mode program that uses a 32-page touch-ahead count.

11.2. Active Segment Data Structures

The AST data structure keeps track of active segments; this structure is composed of an AST header and an array of AST entries. Figure 11-3 shows the format of the AST.

11.2.1. Active Segment Table Header

The AST header contains pointers to linked lists of ASTEs and AST state information.

11.2.1.1. Linked List Pointers

The AST header contains the following pointers that the AST manager uses:

- Start pointers (ASTEXes) to 251 linked lists of ASTEs

- A start pointer to a linked list of free ASTEs available for activation (asth.free)

- A pointer to the next replaceable ASTE (asth.lru). If there are no free ASTEs in the free list, the AST replacer uses this ASTEX as a starting point in its search for an ASTE it can deactivate.

When it activates an object segment, the AST manager chains the new ASTE to the linked list that corresponds to the object's hashed UID. In addition, each of the 251 linked lists contains a number of sublists; each sublist links together all the ASTEs for a single object. The AST manager orders ASTEs within a per-object sublist by descending segment number.

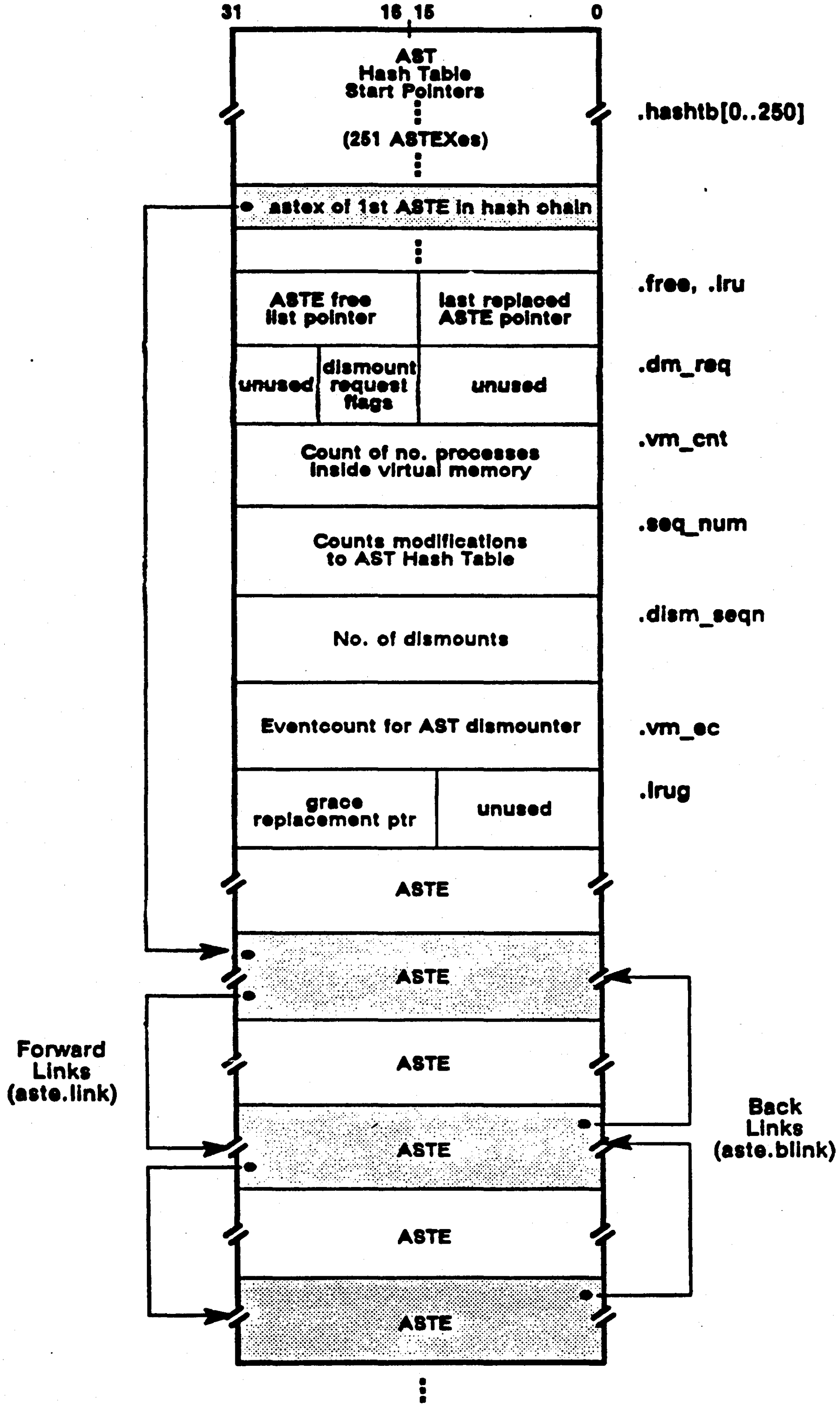


Figure 11-3. Active Segment Table Format

11.2.1.2. AST State Information

Some system routines cannot (because of possible deadlocking) hold an AST resource lock for the duration of their operation. Since these routines cannot prevent the state of the AST from changing, they use fields in the AST header to determine whether the contents of the AST has changed since they began their operation, and to retry the operation if the AST state has changed. These fields are:

- Dismount sequence number (asth.dism_seq)
- Dismount request flags (asth.dism_req)
- Volume count (asth.vm_cnt)
- Dismount eventcount (asth.vm_ec)
- AST hash table modifications count (asth.seq_num)

The first four fields are associated with the AST manager's dismount procedure, which deactivates all the ASTEs for object segments on a given logical volume. The dismounter always increments the dismount sequence number each time it is called to deactivate ASTEs. Since other routines may depend on some of these deactivated ASTEs to be active in order to complete their operation successfully, they save this field in the AST header when they begin operation, then check their saved copy against the current field before they signal completion.

The volume count indicates how many processes are using ASTEs associated with a logical volume. The dismounter checks this field before it begins its dismount procedure to make sure that no one else is using the ASTEs for the volume. Consequently, users of ASTEs on a logical volume can set this field to prevent the dismounter from taking away the ASTEs they need; for example, when they are trying to write to disk.

When the dismounter starts deactivating ASTEs for a given volume, it can encounter a locked ASTE (a positive hold count). If it does, the dismounter sets the dismount request flag bit that corresponds to the volume it was dismounting, and waits until the ASTE is freed. Meanwhile, the holder of the ASTE completes its operation, frees up the ASTE, and checks the dismount request flag field. If a dismount request exists, the routine advances the dismount eventcount, awakening the dismounter. The FILE manager is a frequent user of this field.

Routines that change the order of an ASTE linked list by adding or deleting ASTEs update the AST hash table modifications counter (asth.seq_num). Other routines that depend upon the consistency of a linked list can save this number before they begin operation, then check their saved copy against the current value in the header.

For example, the AST information gathering (ast_$get_info) routine is called to read through a linked list of ASTEs for an object by following the hash link fields in each ASTE. The routine saves the modification count and begins its operation. In the meantime, the AST manager is called (ast_$deactivate) to deactivate an ASTE in the linked list that ast_$get_info is reading. It deactivates the ASTE, then increments the counter. When ast_$get_info finishes reading the linked list, it checks the current state of the counter against its saved copy to make sure the information it is returning is accurate (that no ASTEs have been activated or deactivated). Because the counter values differ, ast_$get_info repeats the entire read operation.

11.2.2. Active Segment Table Entries

Active segment table entries contain four kinds of information:

- Information copied from the VTOCE where the object is stored
- Information about the status of the ASTE, used in ASTE replacement
- Information that records dynamic changes to the object segment
- Information about members in the ASTE linked list

Figures 11-4 and 11-5 show the fields within an ASTE for reverse- and forward-mapped systems.

11.2.2.1. VTOCE Information

The ASTE contains information about the object and the segment being activated that it obtains from the VTOCE:

- The UID of the object to which the active segment belongs.
- The segment's sequence number within the object.
- A VTOCX that points to the object's VTOCE, if the object is stored locally. If the object exists on a remote node, the VTOCX contains the object's home node ID and an index to the network number to which the home node is connected.
- A VTOCX that points to the starting disk address of the file map in which the active segment exists, if the object is local.
- The attributes of the object that owns the segment -- concurrency control, whether it is permanent or immutable, its system type, its lock key, and whether it has encountered file trouble and needs salvaging.
- An index into the device volume table (DVT) that points to the logical volume on which the segment resides; this field is useful on nodes that support multiple logical volumes, such as storage module devices and file servers.

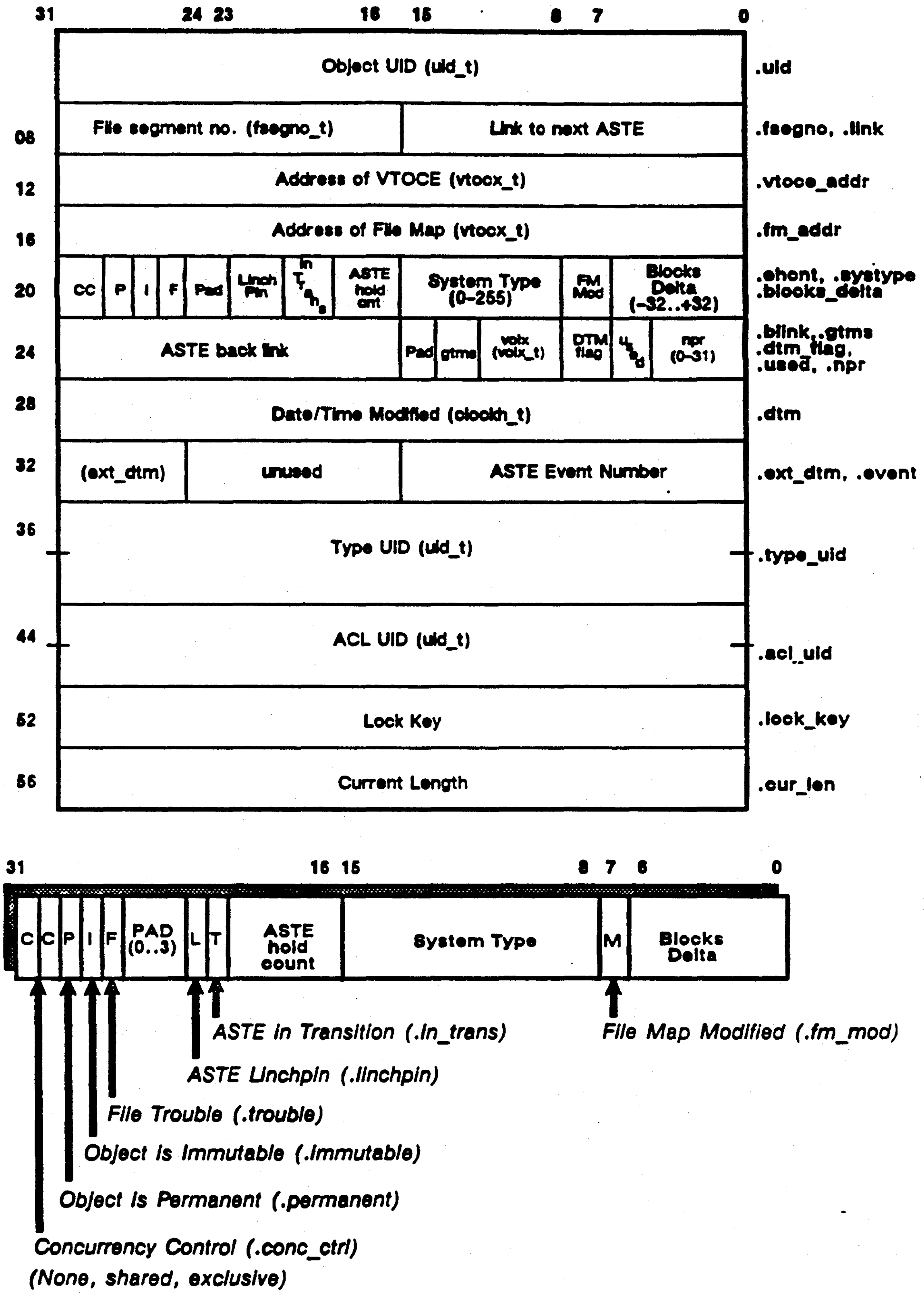
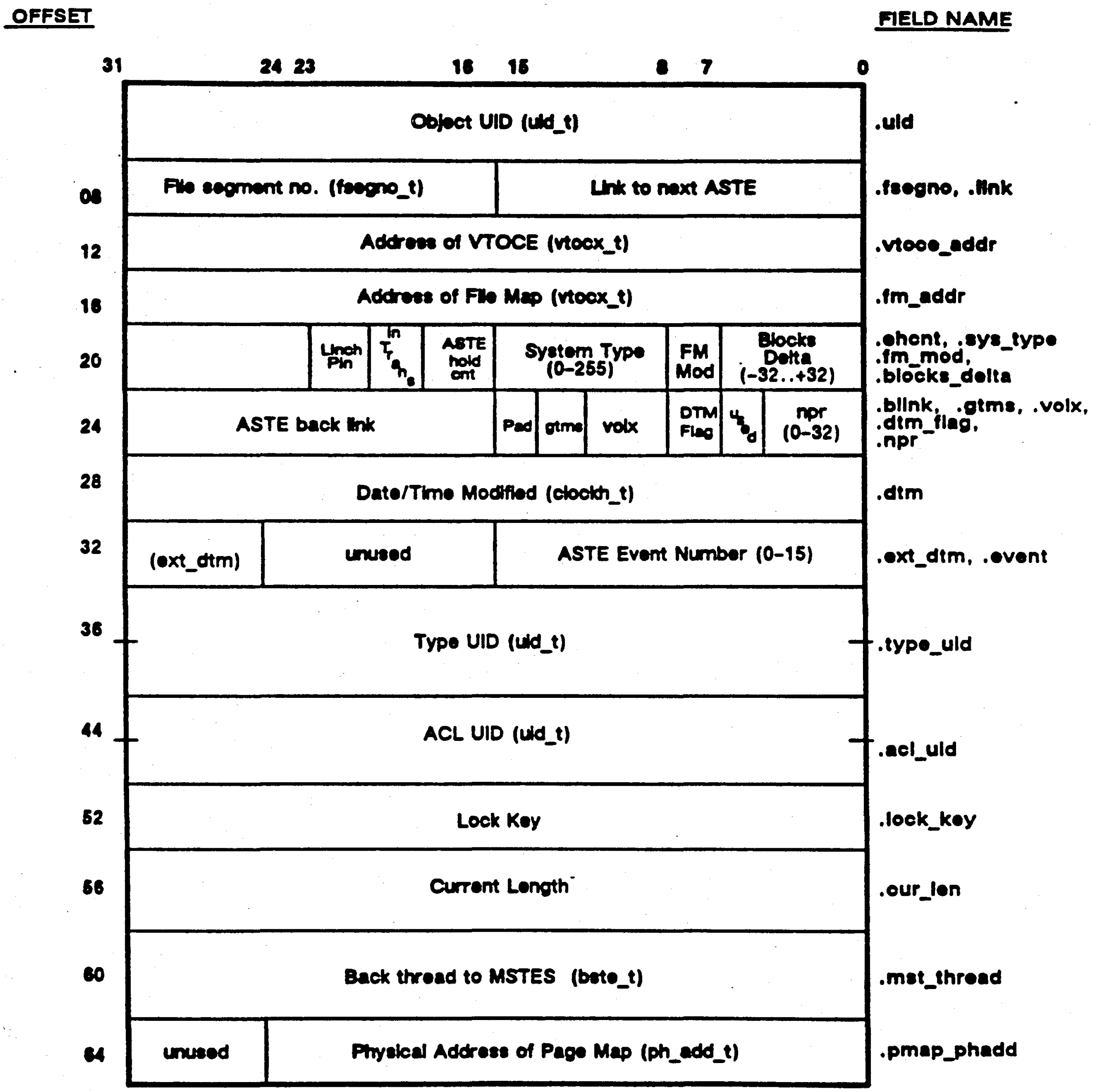


Figure 11-4. Active Segment Table Entry (Reverse-Mapped)



OFFSET	31 · 24 23 · 16 15 · 8 7 · 0	FIELD NAME
	Object UID (uid_t)	.uid
08	File segment no. (fsegno_t) Link to next ASTE	.fsegno, .link
12	Address of VTOCE (vtocx_t)	.vtoce_addr
16	Address of File Map (vtocx_t)	.fm_addr
20	Linch Pin In Trans ASTE hold cnt System Type (0-255) FM Mod Blocks Delta (-32..+32)	.ehcnt, .sys_type .fm_mod, .blocks_delta
24	ASTE back link Pad gtms volx DTM Flag Used npr (0-32)	.blink, .gtms, .volx, .dtm_flag, .npr
28	Date/Time Modified (clockh_t)	.dtm
32	(ext_dtm) unused ASTE Event Number (0-15)	.ext_dtm, .event
36	Type UID (uid_t)	.type_uid
44	ACL UID (uid_t)	.acl_uid
52	Lock Key	.lock_key
56	Current Length	.cur_len
60	Back thread to MSTES (bste_t)	.mst_thread
64	unused Physical Address of Page Map (ph_add_t)	.pmap_phadd

Figure 11-5. Active Segment Table Entry (Forward-Mapped)

11.2.2.2. ASTE Replacement Information

The ASTE has several fields that the AST manager consults in its replacement algorithm:

- ASTE in transition – this field indicates that the ASTE is inconsistent but will become available shortly; the AST replacement procedure will pass over ASTEs in transition.

- ASTE hold count – this field indicates that a process is using the ASTE and does not want the AST replacement procedure to deactivate it to hold some other active segment. The AST replacement procedure passes over any ASTEs with hold counts set (while the AST dismounter waits on ASTEs with positive hold counts).

- ASTE event number – this field is used as a hint for ASTE replacement. Whenever the system activates or changes an ASTE, an AST event occurs and the system increments a master AST eventcount (in the AST header) and copies the eventcount value into this field in the new or modified ASTE. The ASTE replacement procedure checks the ASTE event number field during its replacement scan. If the value within the field falls within a certain range of the master eventcount value, the replacement procedure passes over it; thus, the event number allows ASTEs to *mature* before they are replaced.

- Number of pages resident – this field indicates the number of segment pages that are resident in physical memory. When the system makes a segment page resident, it increments this field; when it takes a resident page away, it decrements the field. The ASTE replacement procedure bases its replacement decision on the number of pages resident for a segment.

11.2.2.3. Linchpin and Back Thread Links

Each ASTE contains a pointer to the next ASTE in the linked list (aste.link) and the previous ASTE (aste.blink). The AST manager orders ASTEs for a single object sequentially by descending file segment number (.fsegno). The ASTE for the object's highest numbered active segment is called the **linchpin** ASTE. The AST manager sets the linchpin field in the ASTE when it activates the ASTE. The linchpin ASTE marks the beginning of the linked list of active segments for the object. When the AST manager is called to activate more segments for the object, it links the newly activated ASTEs to the chain headed by the linchpin ASTE in order of descending file segment number. The AST manager also uses the linchpin ASTE as a cache for the object's attributes; once it determines that it is activating a segment for the same object, it copies the information from the linchpin ASTE into the newly activated ASTE.

ASTEs for memory management on DNx60 systems contain additional pointers to other memory management data structures. The **MSTE back thread** (aste.mste_thread) is a field within the ASTE that points to all the processes that have mapped the active segment into their virtual address space. The back thread in the ASTE contains the ASID of the first process that is using the active segment and the index to the segment map entry that is pointing (through the pmap_phadd field) to the active segment's page map. This process, in turn, points to the ASID/SMAPX of the next sharer via the back segment table (BST). Like the MSTE and the SMAPE, the BST is a per-process table that links together the processes currently sharing the same object segment, and thus are sharing its page map. The system indexes into BST using the ASID and SMAPE; each entry (the BSTE) points to the ASID of the next sharer in the chain and also to its segment map. The BSTE in the ASTE back thread field is the head of this chain.

The memory management subsystem on forward-mapped MMUs uses the BST chain to keep track of all the virtual segments that currently point to a page map. If the subsystem steals a page within the map for another use, it must invalidate all the per-process virtual/physical associations for that page. It obtains the starting BST from the ASTE, then chains backwards through the BSTs, beginning with the last process to map the segment. The subsystem then removes these invalid virtual/physical associations from the translation buffer cache in the MMU. Thus, the BST provides a way to selectively flush pages from the translation buffer.

The ASTE on forward-mapped systems also contains the physical address of the active segment's page map (aste.pmap_phadd). When a forward-mapped system maps an object segment, it sets up the SMAPE to point to the page map's physical address. The hardware uses the SMAPE to obtain the the physical address of the page map during address translation.

11.2.2.4. Object Modification Information

ASTE information about object and segment modification includes:

- File map modified -- this field indicates any increase or decrease in the extent of the object's file map.

- Blocks delta -- this field indicates the number of disk blocks (pages on disk) added to or subtracted from the object segment since its activation.

- Current length -- this field contains the actual length of the object in bytes.

- DTM flag -- this field, when set, indicates that the object's date/time modified field in the VTOCE needs updating. The memory management system stores the DTM in the ASTE and sets this flag so that it will later be copied to the VTOCE.

- DTM -- the date/time modified field provides cache control for an object segment that is being used by several different processes. The DTM is only modified at the *home node*; object modifications at remote locations are sent back to the object's home node to be recorded, and the new DTM is then shipped back across the network to the remote modifier, who stores the new DTM in the ASTE. The AST manager copies the most recent DTM to the linchpin ASTE. However, at any given time, any of the object's ASTEs can have the most recent DTM. For example, when the page purifier writes a modified page to disk, it writes the new DTM into *the ASTE that owns the page*, not into the linchpin ASTE. It is the AST manager that checks the DTMs of all the ASTEs for a given object, then copies the most recent to the object's linchpin ASTE. Consequently, any ASTE in a per-object sublist can contain the most recent DTM.

- GTMS flag -- this field, called the global transparent mode software flag, indicates that the segment is part of user or supervisor global address space. Because the segment is shared, all processes will have the latest copy of the segment and so the DTM does not need updating.

11.3. Physical Page Data Structures

AEGIS physical page data structures keep status about a physical page and store the page's location in memory or on disk. Reverse-mapped systems store physical page information in the PMAPE, the MMAPE, and the PFTE data structures. Forward-mapped MMU systems store the information in the SMAPE, the PMAPE, and the MMAPE. Figures 11-6 and 11-7 illustrate the per-MMU physical page data structures and the fields within them.

The page data structures keep the following statistics about a page:

- Whether it has been referenced
- Whether it has been modified
- Its status in physical memory
- The type of access permitted
- Its physical page number and its disk address
- Its replacement status

11.3.1. Modified Status

The virtual memory management system keeps two modified bits:

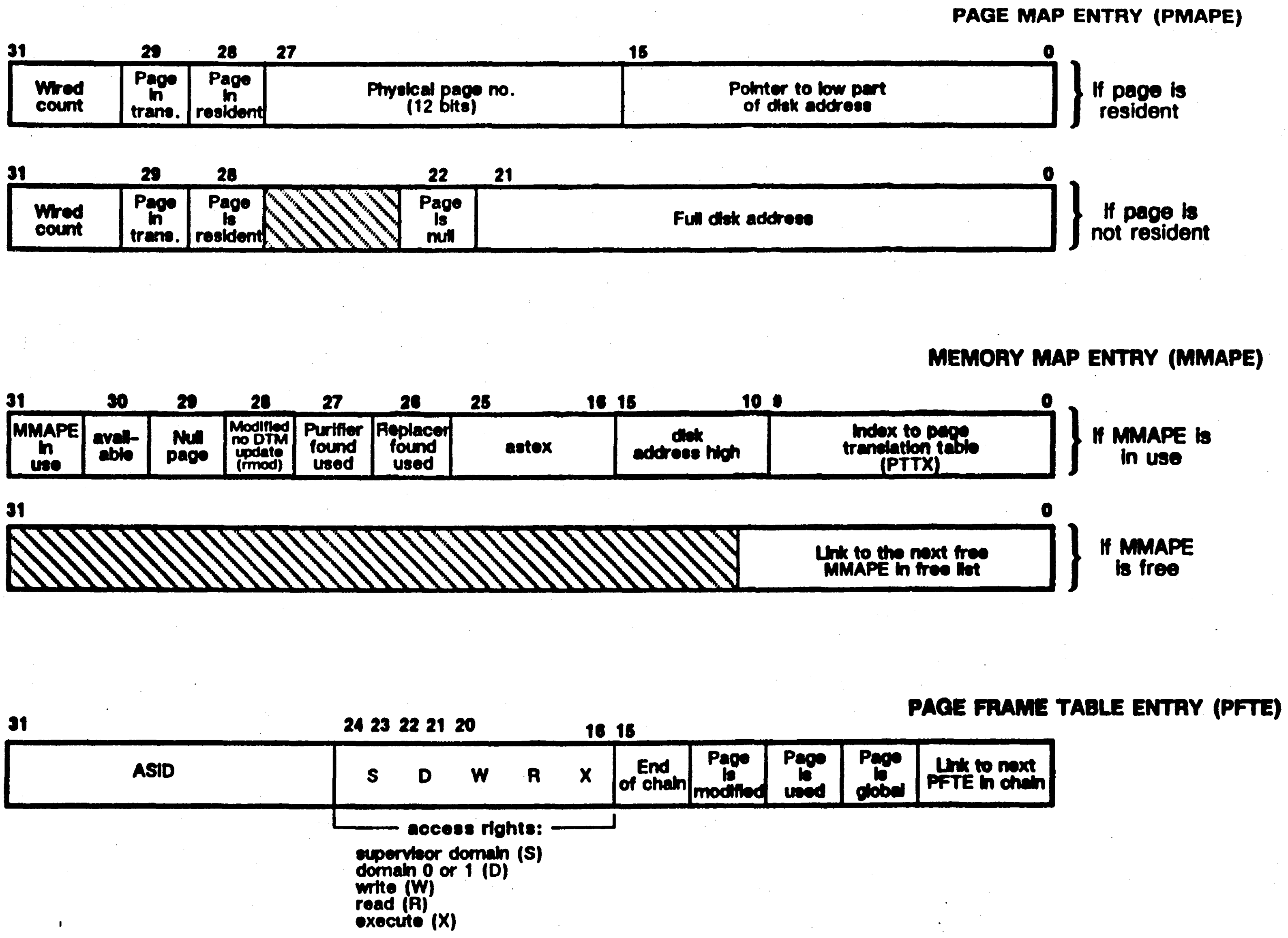
- The page-modified bit in the page map (pmod)
- The page-modified bit in the memory map (rmod)

The **page modified** bit, when set, indicates that a process has modified the contents of a physical page. In reverse-mapped systems, the modified bit resides in the PFTE (flags.pmod); in forward-mapped systems, it resides in the PMAPE (pmape.pmod). The hardware sets the page-modified bit when a process writes to the page. The system uses the page-modified bit in the MMAP to identify pages that have been modified on other nodes in the network.

Both pmod and rmod bits indicate that the page has been modified. However, which modified bit is set controls whether or not the purifier will update the DTM when it purifies the page. When set, the page-modified bit in the PFTE or PMAPE directs the purifier to update the DTM when it purifies the page. If the page-modified bit in the MMAP is set but the pmod bit is clear, the purifier will write the page to disk, but will not update the DTM.

This DTM-update flag is necessary because the purifier normally updates the DTM when it writes a page out to disk; on remotely modified pages, however, it is the *paging server* that updates the DTM. Consequently, the paging server sets the rmod bit to suppress the purifier's DTM update operation while still allowing the page to be purified. Because the rmod bit is set, the purifier will recognize that the page has been modified and so needs to be written to disk; but, because the pmod bit is clear, the purifier will not update the DTM.

Figure 11-6. Physical Page Data Structures (Reverse-Mapped)



Reverse-Mapped Page Tables

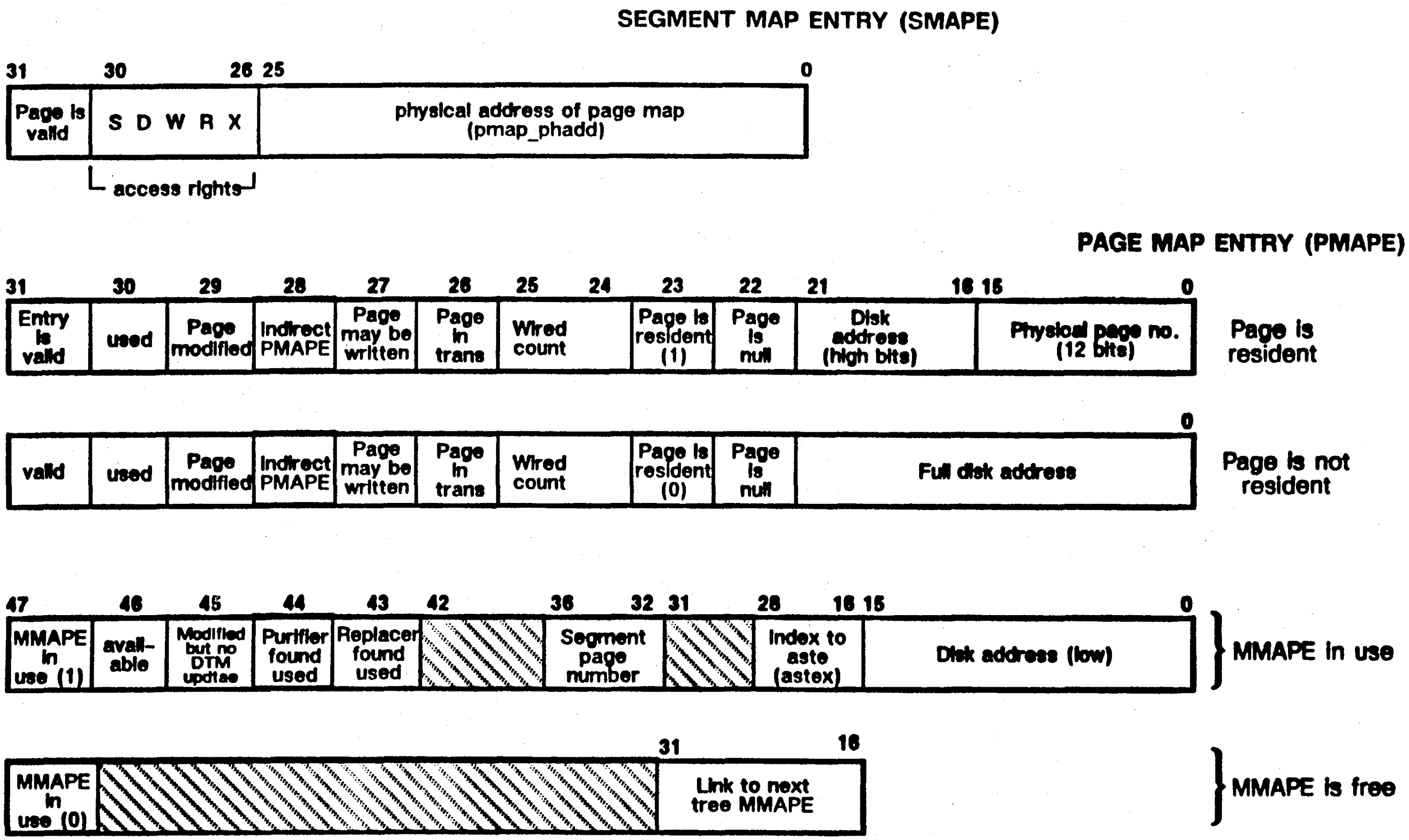


Figure 11-7. Physical Page Data Structures (Forward-Mapped)

11.3.2. Valid Status

Forward-mapped MMU hardware uses the physical page number (PPN) stored in the page map entry to carry out virtual-to-physical address translation. However, the physical page number will not always be valid; for example, when the page replacer steals the physical page for another process's use. As a result, the memory management software must ensure that the hardware does not try to translate an invalid PPN. The PMAPE valid bit, when set, indicates that the PMAPE contains a valid virtual-to-physical association for the page; that is, the hardware will not trigger a page fault if it uses this PPN. The MMAP manager clears this bit to invalidate the virtual-to-physical association when it steals the PPN during page replacement.

11.3.3. Usage Status

The virtual memory management system employs three usage bits:

- The hardware usage bit
- The replacer usage bit
- The purifier usage bit

The hardware usage bit exists in the page frame table entry or the page map entry. When set, it indicates that the processor has referenced the page (the modified bit indicates how it was referenced). Both the purifier and replacer examine the hardware usage bit to determine whether or not they can purify or replace the page; a page in use should not be purified or replaced. The replacer and purifier usage bits in the MMAPE distinguish which procedure found the page used. Thus, the MMAPE usage bits simulate the appearance of two hardware usage bits.

11.3.4. Physical Memory Status

Bit fields within the PMAPE determine the page's status in physical memory. They are:

- The **in transition** bit (pmape.in_trans); A page is marked in transition when the memory management subsystem is making changes to the state of the page.
- The **resident** bit (pmape.resident); A page is marked **resident** when an object-to-physical page association exists for the object page.
- The **wired count** (pmape.wired); A page is wired when it is resident in physical memory and also cannot be paged out.
- The **null** bit (pmape.null); A page is marked as null when the copy on disk is invalid for the page that exists in physical memory or the page has never been written to the disk.

11.3.5. Access Rights

The hardware uses the access rights to check for possible access violations. When a program requests a virtual-to-physical translation, it also specifies the way it wants to access the page. The hardware checks the requested access mode against the access mode stored in the PFTE or SMAPE and returns an access violation if the modes do not match. Thus, the hardware performs access checking as well as address translation on every operation.

11.3.6. Location in Memory and on Disk

If an object page is resident in memory, its PMAPE contains the physical page number assigned to the page, and the PMAPE and MMAPE contain the low and high portions of the disk address. Disk addresses are split between PMAPE and MMAPE because the PMAPE does not contain enough space to store both the PPN and the disk address. Forward-mapped systems store the high portion of the disk address in the PMAPE and the low portion in the MMAPE; it's the reverse in reverse-mapped systems. The MMAPE for resident pages also contains the segment page number (its sequence number within the object segment).

The PMAPE for an object page that is not resident in memory contains the full disk address.

11.3.7. Page Replacement Status

The MMAPE contains two bits: the **in-use** bit and the **available** bit (mmape.inuse and mmape.avail) that MMAP manager and PMAP manager use to determine the page's availability for replacement and/or purification. These bits describe a page's status. The MMAP manager sets the in-use bit when it allocates a physical page to hold an object-to-physical memory association, and sets the available bit when it determines that a page should become available for replacement. (Pages with wired or in transition status are marked unavailable for replacement.)

The purifier and page replacer check the in-use and available bits during their operations:

- If the in-use bit is clear, and the available bit is set, the page is free.
- If, however, the the page is marked in use and the available bit is clear, the page is unavailable for purification or replacement.
- If both in-use and available bits are set, the page is in use but available for replacement.

Together, the in-use and available bits indicate whether the page is eligible for purification or replacement, depending upon which algorithm is checking. Whether or not the purifier or replacer should *do* anything to the page is based on the hardware usage and modified bits, as follows:

- If the page hasn't been modified, purification is unecessary.

- If the page is marked used and available and modified, it is an impure available page that should be purified before it is replaced.

- If the page is in use, purification or replacement is unwise.

11.3.8. Pointers to Other Structures

If a physical page is in use, The MMAPE contains the ASTEX for the active segment to which it is allocated. Reverse-mapped systems store the index to the page translation table in the MMAPE as well. The ASTEX in MMAPE tells the MMAP manager where the active segment table is; on reverse-mapped systems, the PTTX in the MMAPE is used in address translation.

The PMAPE for DNx60 systems contains a field to indicate an indirect PMAPE; currently, this field is defined but not implemented.

Chapter 12
Mapping, Activation, and Purification

This chapter details the algorithms used by memory management software as part of their address space association functions. The chapter explains:

- Mapped segment manipulation

- Active segment activation, deactivation, and replacement

- Physical page management, including page replacement and purification operations

12.1. Summary of MST Operations

The MST manager provides:

- Modules that user programs can call; that is, the *exported* MST interface

- Modules that other AEGIS kernel managers call

- Modules that only the MST manager calls to carry out functions on the behalf of AEGIS kernel managers or user-mode programs

The next sections summarize MST module operations.

12.1.1. MST Routines Called From User Space

The routines listed below constitute the user program interface to the MST manager. When a user program calls one of these routines, the SVC catcher invokes the appropriate MST manager internal routine on the user program's behalf. The user callable routines are:

- mst_$map, mst_$map_global -- Calls the MST internal module mst_$int_maps to map segments of a specified object into user private or user global address space.

- mst_$unmap, mst_$unmap_global -- Calls the MST internal module mst_$unmap_privi to unmap the object's segments from virtual address space

- mst_$remap - maps a different or larger portion of an object that is currently mapped. Calls the MST modules mst_$unmap_privi and mst_$int_maps to perform the operation. If mst_$remap incurs an error, it unmaps the object from virtual address space and returns nil. However, if an error occurs during the execution of mst_$unmap_privi, the module leaves the address space in an indeterminate state from which it is usually impossible to recover.

- mst_$change_rights - changes the access rights for an object that is currently mapped. If the module incurs an error, the object remains mapped.

mst_$set_guard -- Sets the guard bit in the MSTE for the specified region of virtual address space, which makes that region a guard segment. The process manager (PM) calls this routine when it sets up the stack object. The MST routine calls the MST manager modules mst_$unmap_privi and mst_$int_maps to perform the operation.

- mst_$set_touch_ahead_count -- Calls the MST module mst_$priv_set_touch_ahead_cnt to set the touch ahead count for the specified object. The touch ahead count determines how many of the object's pages are brought into physical memory on a page fault.

- mst_$invalidate -- Invalidates a portion of an object that is mapped to the given range of virtual addresses. This routine eventually calls the AST manager (ast_$invalidate) to invalidate the data in physical memory.

- mst_$map_special -- On forward mapped MMUs, this routine maps the special_seg_$uid into the special segment. Special segments hold pages for AEGIS data structures such as AST and PMAP pages. DNx60 systems use special segments to reference virtual addresses that cannot take a page fault.

12.1.2. MST Routines Called from the Kernel

The MST manager provides, for the managers in the AEGIS kernel:

- Kernel entry points for mapping

- Routines for fetching pages into physical memory

- Routines for ASID allocation

- Routines used in fork process creation

- Routines used during system initialization

12.1.2.1. Kernel-Level Mapping Modules

Kernel managers that need to map objects call mst_$maps and mst_$maps_at. Like the user-mode modules, both these routines call mst_$int_maps and mst_$int_maps_at to map object segments to virtual address space, and call mst_$unmap_privi to unmap. The managers that call these mapping routines include the naming server, directory manager, ACL manager, HINT manager, PROC2 manager, and system initialization procedures.

12.1.2.2. Touch and Wire Operations

The MST touch operation brings into physical memory from local disk or from another node in the network the object pages that are mapped to pages in virtual address space. The module, mst_$touch, is called by the fault interceptor manager when a page fault occurs; that is, when a process tries to reference an object using a virtual address that has no corresponding physical address. The touch operation *resolves* the page fault; it makes the virtual to physical address association that was missing when the object page was referenced, causing the page fault to occur. To resolve the page fault, the mst_$touch routine calls the cached OSS managers AST and PMAP. For details of page fault handling, see Chapter 13.

The touch operation makes object pages resident in physical memory. However, the physical page replacement mechanism can take these pages away to map another process's virtual memory. The wire operation both *touches* and *wires* the pages. It makes them permanently resident in physical memory and so unavailable for page stealing.

The AEGIS system device drivers call mst_$wire_area to wire to physical memory an area of virtual address space for their use. The mst_$wire_area routine returns a list of physical page numbers to the driver. The mst_$wire_area routine calls the MST wired module mst_$wire to carry out the wiring operation.

The system initialization procedure, the MST manager, and some of the AEGIS device drivers directly reference the mst_$wire module.

Both mst_$touch and mst_$wire are themselves wired modules; their procedures and data must be wired into physical memory during the life of the system.

12.1.2.3. Modules used by the PROC2 Manager

The level 2 process (PROC2) manager calls special MST manager routines to carry out fork operations on the DOMAIN/IX facility's behalf (process forking is described in Chapter 16. These routines are:

- mst_$fork -- Copies the parent process's MST to the child process and copies the contents of the parent process's stack object to the stack object for the child process. (PROC2 must first allocate the child process's ASID).

- mst_$unmap_all -- Unmaps the entire address space below the protection boundary. The PROC2 routine proc2_$complete_'vfork calls this routine when it is preparing to clean up a process; the routine calls mst_$unmap_privi to do the unmapping operation.

The PROC2 manager also calls the MST manager to allocate and free ASIDs as it creates and deletes level 2 processes. These routines are:

- mst_$free_asid -- Unwires and frees the MST of the specified ASID; the proc2_$delete routine calls this routine when it deletes a level 2 process

- mst_$deallocate_asid -- Deallocates an ASID only if no objects are mapped to it.

12.1.2.4. MST Modules Called During System Initialization

The system initialization procedure os_$init calls the MST manager's mst_$init routine to initialize the MST database and create the ASID allocation list, invokes the AST initialization module, and also calls mst_$map_rem_paging_file to allocate the OS paging file. The initialization procedure is the only kernel module that calls mst_$set_priv_touch_ahead; this routine sets the touch-ahead for supervisor global modules such as the OS paging file. See Chapter 28 for an explanation of system initialization.

12.1.2.5. Modules Used in Cross-Process Debugging

The cross-process debugging facility (XPD) uses the following information gathering operations in the MST manager:

- mst_$get_uid -- Returns the UID mapped at the specified address.

- mst_$get_uid_asid -- Returns the UID mapped at the specified address for a given ASID.

- mst_$get_va_info -- Returns information about the object mapped at the specified address.

12.1.3. Modules Called Within the MST manager

The MST manager is separated into wired and unwired modules. The MST manager uses the following unwired modules during mapping operations:

- mst_$get_mste -- Returns a pointer to the MSTE at the specified address. Since the access rights are stored in the SMAP in forward-mapped MMUs and in the MSTE in reverse-mapped systems, this routine returns the access rights separately. (Calls wired modules mst_$va_to_segno to find the MSTE).

- mst_$wire_mste -- Ensures that the MSTE is wired. If it is not, the routine attempts to wire it. This procedure will wire an additional page of mst entries only if the system has not exceeded the MST entry limit.

- mst_$get_info -- Checks the object's access rights, then obtains information about the object for the mst_$maps, mst_$maps_at, and mst_$fork routines

The MST manager's wired modules contain procedures and data that must be wired into physical memory. They are:

- mst_$va_to_segno -- Converts the given ASID and virtual address into ASID/segment number indexes into the per-proces MSTs.

- mst_$int_alloc_asid -- Finds a free ASID and allocates it. The kernel routine mst_$alloc_asid calls this internal routine on the PROC2 manager's behalf.

- mst_$install_ioppn -- Installs an I/O space physical page number (PPN) into the page map (PMAP) for the given virtual address.

- mst_$remove_ioppn -- Removes an I/O space PPN from the page map. These routines are used by the PBU manager to map memory-mapped controllers into user private address space. The general purpose I/O (GPIO) facility uses the PBU manager and these last two MST routines.

The next section describes how the MST manager's routines work to map an object into virtual address space.

12.1.4. Mapping Object Segments

The MST manager's mst_$int_maps routine handles the actual mapping of an object to a series of MSTEs. The kernel managers that map segments call this routine indirectly via mst_$maps. The user-callable mapping system services call it via mst_$map or mst_$map_global.

(Note that mst_$int_maps_at carries out approximately the same steps but calls mst_$va_to_segno to find the location at which to map the object segments.)

The mst_$int_maps routine maps an object to virtual address space by:

- Determining the area of virtual address space into which the object is to be mapped
- Checking the caller's access rights against those of the object
- Getting the information about the object
- Loading the mapped data structures with the retrieved information

12.1.5. Determining the Address Space

The mst_$int_maps routine first examines the ASID and the protection boolean specified in the call to determine into which section of virtual address space the object is to be mapped.

- ASID 0, unprotected means map to user global
- ASID 0, protected means map to supervisor global
- ASID 1-25, unprotected means map to per-process user
- ASID 1-25, protected means map to per-process supervisor

Once the routine determines where in the address space the object belongs, it knows which MST (MST for ASID 0 or a per-process MST) it should use to map the segment.

12.1.6. Checking Access Rights

Some objects have a system type attribute which identifies objects that are important to the system's internal operation. Callers running in supervisor mode (for example, the naming server) can map any type of object to virtual address space. If the calling process is running in user mode, mst_$int_maps calls the ACL manager (mst_$get_info -->acl_$rights) to check the calling process's rights to the object against those specified in the ACL for the object and whether the process is attempting to map a non-file type object.

12.1.7. Getting the Information about the Object

The mst_$int_maps routine calls (via mst_$get_info) the AST manager to retrieve the information about the object from the AST, from the VTOC, or from another node in the network.

The AST manager (ast_$get_info) first searches for an active ASTE that corresponds to the UID of the object being mapped. If it cannot locate an active segment, it calls the VTOC manager to search the VTOCs of all mounted volumes for the obejct's VTOCE. If the object is not local, the AST manager asks the hint manager for likely remote locations, and examines the node ID portion of the UID to determine the node on which the UID was created; this is also a likely location. If the AST manager cannot locate the file, it returns *file not found* to the MST manager.

12.1.8. Loading the Mapped Segment Data Structures

If the object is to be mapped to per-process space, the module locates the correct MSTE from the given ASID and searches for a series of free MSTEs, starting from the MST's lowest segment. If it cannot locate free MSTEs, the process has mapped its maximum number of object segments, and the routine returns an error message.

Once it has located empty MSTEs, the module calls its internal routine to copy the information about the object into the MSTEs. The routine wires the MSTEs to be loaded and then fills in the fields. On forward-mapped mapped MMUs, this routine also copies the access rights into the associated segment map entry (SMAPE).

12.2. Active Segment Operations

The AST manager performs the following operations:

- Maintains the active segment table data structures by activating and deactivating ASTEs.
- Maintains local AST-to-VTOC consistency by updating the VTOC to the state of the AST.
- Maintains distributed cached consistency on the lock manager's behalf.
- Performs operations on objects cached in the AST on the FILE manager's behalf.

The following sections describe how the AST manager activates and deactivates ASTEs and how and when it updates the VTOC to the state of the AST. Chapter 6 discusses how the lock manager and the AST manager interact, while Chapter 5 discusses how the FILE and AST managers interact.

12.2.1. ASTE Activation

The ASTE activation routine ast_$activate activates an ASTE by finding a free ASTE, copying the object information into it, and adding it to the linked lists of ASTEs. The AST manager activates segments in the following cases:

- During the AST touch operation (ast_$touch/ast_$touch_segva)

- When the lock manager calls the AST manager to get an active segment's DTM, and the segment is not active (ast_$get_dtm)

- When the AST manager is invalidating a local file (ast_$invalidate) and the segment is not active

12.2.1.1. Finding a Free ASTE

The ast_$activate module calls ast_$allocate to allocate a free ASTE from the free list or via the ASTE replacement algorithm. Once ast_$allocate locates an ASTE, it marks it in transition to prevent it from being replaced and returns the ASTEX to ast_$activate.

12.2.1.2. Loading the ASTE

The ast_$activate routine calls its internal *install* routine to load the object and object segment information into the new ASTE and associated PMAP. The install routine first determines whether the segment being activated is the first active segment for this object or whether the object has other active segments in the AST. If other active segments exist for the same object, the routine can copy the object attributes and file map from the linchpin ASTE. If the segment being activated is the object's only active segment, the install routine calls the VTOC manager to retrieve the object attributes (via calls to vtoc_$lookup and vtoc_$read). In either case, it must retrieve the file map from the VTOC on the specified volume (via a call to fm_$read).

If ast_$activate is activating a remote segment, it calls the ast_$get_info routine to obtain the segment information from the specified node.

12.2.1.3. Adding the ASTE to the Linked List

Once ast_$activate has read the VTOCE and file map, it checks the AST list to make sure that the segment it is activating is not already activated. If the segment has already been activated, the module returns the ASTE to the free list and exits.

The module then gets the AST lock and adds the new ASTE to the appropriate linked list. Because more than one UID hashes to the same value, ast_$activate adds segments to ASTE linked lists in two steps. It adds new ASTE:

- To the ASTE chain that links together all the active segments whose UIDs hash to the same value

- To the linked list within that ASTE chain that links together all the active segments for this UID; that is, for *one* object

The first ASTE in the linked list of one object's active segments is linchpin ASTE; if the segment that ast_$activate is activating belongs to an object that has other active segments, the routine links the newly activated segment to this list in descending order of its segment number.

12.2.2. ASTE Deactivation

The routine ast_$deactivate removes specified ASTEs from ASTE linked lists and places them onto the AST free list. The AST manager deactivates segments in the following cases:

- When it must free up an ASTE during segment activation (ast_$allocate)

- When it deactivates the ASTEs for those segments that reside on a specified logical volume during a volume dismount (ast_$dismount)

- When a object is truncated (ast_$truncate)

- When an object is locked; at this time, the AST manager deactivates any stale active segments of an object (ast_$cond_flush)

The ast_$deactivate routine returns with the ast_$lock held and the ASTE marked in transition. The routine cannot deactivate ASTEs that are in transition, held, or have wired pages in the associated PMAP.

The AST manager deactivates an active segment by:

1. Writing all modified resident pages to disk (via pmap_$flush)

2. Updating the segment's VTOCE and file map if deactivation is the result of a volume dismount request or the segment has been written

3. Removing the deactivated ASTE from the ASTE linked list and adding it to the free list

4. Incrementing the AST header's hash table modifications counter (asth.seq_num) to indicate to other memory management routines that a change in the AST state has occurred

5. Advancing the ASTEs in transition eventcount (ast_$ec) to awaken any waiters for ASTEs in transition

On forward-mapped systems, ast_$deactivate also invalidates the ASTEX in the MSTEs of all processes sharing the object segment, invalidates the associated SMAPEs, and purges the translation buffer in the MMU.

12.2.3. ASTE Replacement

When ast_$allocate cannot find an ASTE on the free list, it initiates its ASTE replacement algorithm to find an ASTE to deactivate. The replacement algorithm has a window through which it scans subsets of the ASTEs in the AST, looking for an ASTE to deactivate. If it cannot find a deactivatable ASTE within one subset, it shifts its window to another group of ASTEs and searches that next subset.

To begin the search, the replacement algorithm indexes into the AST to the ASTE that it last replaced (pointed to by the AST header field *asth.lru*). It then carries out a three-part scan of each ASTE subset, passing over any ASTEs held or in transition. The three-part scan consists of the following actions:

1. Checking the number of pages resident field (aste.npr) to find an ASTE with no pages resident.

2. Searching for the ASTE with the fewest number of pages resident.

3. Searching for any deactivatable ASTE (one that is not held, in transition, or has wired pages)

If the routine locates a replaceable ASTE during any one of its scans, it calls ast_$deactivate to make it free. The ast_$allocate routine marks the returned ASTE in transition to prevent the AST manager from replacing it on another caller's behalf.

If the routine cannot find any deactivatable ASTE in the AST because all the ASTEs are either held or in transition, it crashes the system with *ast_$unreplaceable* error status.

12.2.4. Updating the VTOC

The AST manager either updates the entire VTOC to the state of the AST (ast_$update), or it updates selected VTOCEs (ast_$update_vtoce). The purifier process is the only system component that calls the AST manager (ast_$update) to update the entire VTOC to the current state of the AST. The AST manager updates selected VTOCEs on:

- Force-writes (file_$unlock calls ast_$purify, which calls ast_$update_vtoce)

- Volume dismount, as part of ASTE deactivation (ast_$dismount calls ast_$deactivate, which calls ast_$update_vtoce)

12.3. Page Purification

Page purification exists for two reasons:

- To ensure that physical pages continue to be available to all processes in the system. The system cannot reissue a physical page that contains a modified object page until those modifications are written to disk.

- To ensure that modified pages are written to disk in a timely manner in the event that no physical memory pressure exists to force their purification

The purification process that provides available pages is called **demand-based purification**, while the purification procedure that enforces writes to disk is called **time-based purification**.

12.3.1. Demand-Based Purification

The page fault resolution procedure includes an operation to bring an object page from its home disk into physical memory. This operation invokes the page allocation procedure to locate a physical page to hold the object page. The page allocation procedure then circulates through the pages of physical memory looking for a page to allocate. A page that is eligible for allocation is either a **free page** or a **replaceable page**. A free page is marked unused and available in the MMAPE. A replaceable page has three criteria:

- It is marked available in the MMAPE (the in-use and available bits are set)
- The processor is not currently using it (the hardware usage bit is clear)
- The page has not been modified and is thus a pure page (both the pmod and rmod bits are clear).

The page allocation procedure cannot allocate pages that have been modified or pages that the processor has recently referenced. Thus, when the demand on physical memory is great, the page allocation procedure can run out of free and replaceable pages.

In this case, it advances the eventcount on which the demand-based purifier is waiting (purifier_$ec). The demand-based purifier cleans pages for the allocation procedure. When it is awakened, this purifier circulates through the pages in memory looking for modified pages that have remained unused since the last time the allocation procedure examined them. The purifier then writes the modified contents of eligible pages to disk, making them pure and therefore available for re-use.

In the meantime, the page allocator has suspended the process that requested the pages by waiting on the pages eventcount (pages_$ec). When physical memory is no longer available on the node, many processes will be waiting on this eventcount for the purifier.

When the demand-based purifier determines that it has purified enough pages, it advances the pages eventcount, which awakens all the processes that want to bring object pages from disk into physical memory.

12.3.2. Time-based Purification

Demand-based purification is sufficient for nodes that contain a limited amount of physical memory and thus will experience frequent contention for physical pages. However, some node models provide a large amount of physical memory. The virtual memory management system on these types of nodes can satisfy the demand for physical pages without having to free up modified pages by writing them to disk. As a result, a lag time can develop between the time that the page is modified and the time it is written to disk. If a power failure were to occur during this interval, all of the page modifications sitting in physical memory would be lost.

The time-based purifier exists to remedy this situatation: it writes modified pages out to disk in the event that no demand to force them out exists. The time-based purifier ensures that the interval between the time a process modifies a page and the time the system writes it to disk is only about two minutes.

The time-based purifier periodically circulates through a subset of the pages in physical memory and purifies modified pages that it determines to be least recently used; that is, pages that are not likely to be modified in the near future. It also writes the contents of the ASTE through to the VTOCE.

12.3.3. Local Page Purification

The time-based and demand-based purifiers call the write_page (one page) and write_list (list of pages) procedures to carry out the write to disk. The write_list procedure sets all the pages it is going to write in transition and breaks the virtual-to-physical assocations in the memory management tables. It then builds a block header from the information in the ASTE and calls disk_$write to do the local I/O, passing it the block header.

12.3.4. Remote Page Purification

When either purifier encounters remote pages that have been modified locally, it sends these pages back to their home nodes to be stored via a **page-out request** to the NETWORK manager (write_page also does this operation). In this case, the purifier is the NETWORK manager's client; the NETWORK manager writes the pages out across the network (via network_$write) on the purifier's behalf.

12.3.4.1. Building the Page-Out Request

Before it calls the NETWORK manager, the write_list (or write_page) procedure builds the block header from the ASTE as in the local case. However, it inserts the DTM from the ASTE into the block header along with the rest of the information to be passed. The purifier then calls the NETWORK manager (network_$write) to package up the block header into a page-out request packet. The NETWORK manager sends the resulting packet across the network to the remote paging server on the object's home node.

12.3.4.2. Handling the Page-Out Request

The remote paging server on the home node determines that the request is a page-out request and calls the AST manager (ast_$touch), giving the UID and segment page number of the modified page and the PPN containing the new object page contents.

The AST manager checks for possible concurrency violation. If the lock key check does not pass; that is, if the node ID in the lock key is not the same as the node passing in the pages to be written to disk, the AST manager simply drops the pages and the update does not occur. In this case, the client purifier at the remote node receives *concurrency violation* status.

Otherwise, the AST manager calls pmap_$assoc to make the association between the new object page and a physical memory page. The pmap_$assoc routine locates the PMAPE for the page and waits if it finds the page in transition.

The object page that the modified page is replacing may already be associated with a physical page (resident in memory). If this is the case, the PMAP manager breaks the virtual-to-physical and cbject-to-physical associations and frees up that physical page for some other process's use. The pmap_$assoc routine then updates the PMAPE for the new object page, marking the page resident with no copy on disk because the disk is still storing the old copy. It then calls the MMAP manager to install the PPN/object page association into the MMAP (via mmap_$install) and finally returns to ast_$touch. The local purifier will eventually purify the page in physical memory by writing it out to disk.

If the pmap_$assoc routine finds that it has a modified page with no backing storage, it returns status of *bad association* to the AST manager.

The ast_$touch routine returns to the paging server (unless pmap_$assoc encounters a bad association), which obtains a new DTM and inserts this value to into the reply packet. The paging server then sends the packet back to the network manager on the modifying node.

12.3.4.3. Page-Out Post-Processing

When the purifier (write_page) receives the reply packet from the home node, it extracts the new DTM and calls the AST manager to update the ASTE with the new DTM.

If the purifier receives *file not found* or *bounds fault* status from the remote node, it eliminates the page entirely by breaking the virtual-to-physical associations, setting the page to null, and freeing up the physical page for some other process's use.

12.3.5. Page Allocation for Remote Operations

The remote file system permits processes on remote nodes to allocate physical pages on a local node for remote operations. The **remote paging pool** exists to limit the amount of physical memory that remote nodes can use so that remote processes do not steal physical memory away from the local processes.

The system allocates pages for the remote paging pool from high end of physical memory. Once the system creates the remote paging pool, remote paging requests can only use pages from remote paging pool.

When the page allocation procedure (mmap_$alloc) is called to allocate a physical page, it checks to see if its caller is the remote paging server and whether the system has supplied a remote paging pool. If both are true, the allocation procedure limits its search for an available page to the remote paging pool.

If a user adds multiple remote paging servers to the system, the system adds more pages to the pool. As a result, two remote paging servers can double the size of the remote paging pool.

Chapter 13
Page Fault Resolution

A page fault occurs when the processor tries to reference a virtual address that has no corresponding physical address mapping in the memory management tables. (With the forward-mapped MMU, the in-memory page tables, with the reverse-mapped MMU, the page frame table and page translation table.) The managers in the memory management subsystem are then called to resolve the page fault.

This chapter describes how the managers in the memory management subsystem interact to handle page faults. There are several types of page faults that can occur:

- Typical page faults, where the pages exist on local disk and need to be brought into physical memory
- Growth faults, where a segment's page has no associated disk storage
- Null page faults, where the page's backing storage has no valid data
- Resident page faults, where the pages are already resident in memory but no virtual-to-physical association exists
- Sharing faults, where two or more process virtual addresses translate to the same physical page
- Remote page faults, where the pages to be brought into memory exist on a disk attached to a remote node

The chapter begins by describing each manager's function to resolve a typical page fault. Subsequent sections then describe the flow of control during growth, null, resident, sharing, and remote page faults.

13.1. Handling a Typical Page Fault

When a process generates a page fault, the memory management unit hardware signals a bus error, which vectors through the trap page to a routine in the wired portion of the kernel-level fault interceptor manager (FIM). The FIM handles the page fault by calling the memory management subsystem to resolve the fault. (FIMs on reverse-mapped systems check for sharing faults; see Section 13.6.)

Figure 13-1 shows the flow of control between the managers in the subsystem and summarizes the operations that each manager performs to resolve the page fault. The next sections detail the operations performed by each manager.

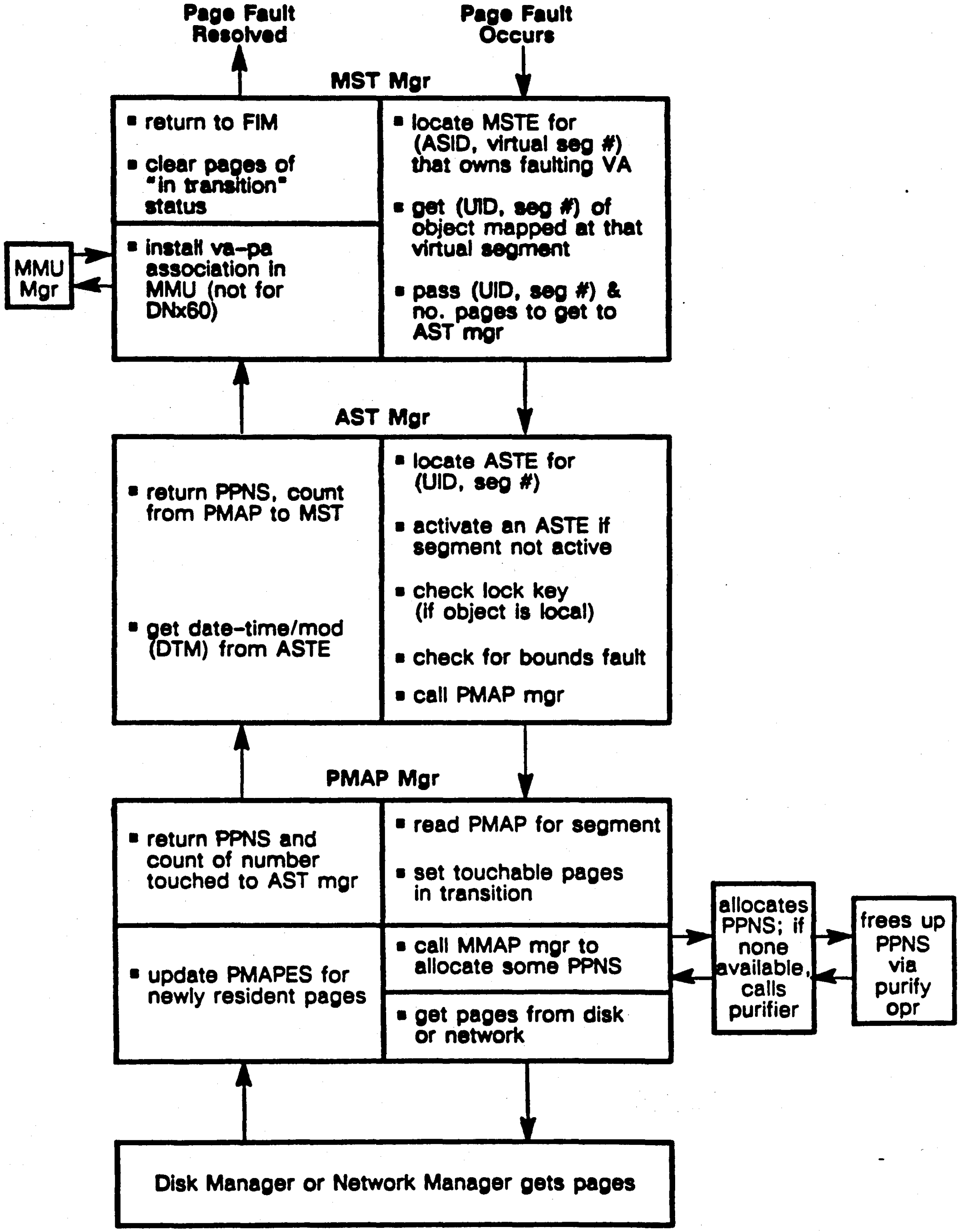

Page Fault Resolved
Page Fault Occurs
MST Mgr
▪ return to FIM
▪ clear pages of "in transition" status
▪ install va-pa association in MMU (not for DNx60)
MMU Mgr
▪ locate MSTE for (ASID, virtual seg #) that owns faulting VA
▪ get (UID, seg #) of object mapped at that virtual segment
▪ pass (UID, seg #) & no. pages to get to AST mgr
AST Mgr
▪ return PPNS, count from PMAP to MST
▪ get date-time/mod (DTM) from ASTE
▪ locate ASTE for (UID, seg #)
▪ activate an ASTE if segment not active
▪ check lock key (if object is local)
▪ check for bounds fault
▪ call PMAP mgr
PMAP Mgr
▪ return PPNS and count of number touched to AST mgr
▪ update PMAPES for newly resident pages
▪ read PMAP for segment
▪ set touchable pages in transition
▪ call MMAP mgr to allocate some PPNS
▪ get pages from disk or network
allocates PPNS; if none available, calls purifier
frees up PPNS via purify opr
Disk Manager or Network Manager gets pages

Figure 13-1. Page Fault Handling

13.1.1. MST Page Fault Handling

The fault interceptor manager calls the MST manager routine mst_$touch to initiate page fault resolution. The mst_$touch routine's function is to locate the virtual segment that contains the faulting address and get the UID and segment number of the object segment mapped to that virtual segment. The mst_$touch routine carries out this operation as follows:

1. Calls mst_$va_to_segno to convert the ASID/virtual address passed to it by the fault interceptor manager to an index into the per-process MSTs.

2. Determines whether the page fault is for a page within a private or global segment, and increments the appropriate per-process page fault counter (in procl_$stats).

3. Checks for special segments, if running on a forward-mapped MMU. Special segments store resources that cannot be paged in and out of physical memory; for example, AEGIS kernel data structures.

4. Gets the number of pages that should be ideally fetched into memory from the touch ahead count count specified in the MSTE and passes this value, plus the object UID and segment number, to the active segment table manager.

13.1.2. AST Page Fault Handling

The AST manager's function in page fault resolution is to locate the active segment table entry for the segment, and to activate an ASTE for the segment if it is inactive.

13.1.2.1. AST Locking During Page Fault Handling

Use of the active segment table is controlled by the **ast_$lock resource lock** and by the **hold count** and **in transition** fields in the ASTE. The AST lock gives the AST manager's caller exclusive use of the AST linked lists. It prevents other AST callers from making changes to the state of the AST while the AST manager is searching for an ASTE.

However, the AST lock must be released once the AST is located so that other processes can access ASTEs. Giving up the AST lock makes it possible for the AST manager to deactivate the ASTE before any of the segment's pages become resident and for other callers to change the contents of the ASTE while it is in use.

To prevent the AST replacement algorithm from deactivating the ASTE, the AST manager increments the ASTE hold count each time it is called to use the ASTE. The hold count is similar to the wired page count; other processes can use the ASTE, but it cannot be deactivated. The AST manager later will decrement the hold count when its caller is finished holding the the ASTE. To block other callers from changing the state of the ASTE, the AST manager sets the in transition bit when it activates or deactivates an ASTE. The in transition bit indicates that the ASTE is changing and so its contents cannot be trusted. Callers that want to use an ASTE in transition will wait on an eventcount.

13.1.2.2. Locating and Activating the Segment

The AST routines used to locate and possibly activate a segment differ depending on the type of MMU hardware in use. On reverse-mapped MMUs, the MST manager calls ast_$touch to locate the ASTE and, if necessary, to activate an ASTE for the segment.

On forward-mapped MMUs, mst_$touch calls ast_$touch_segva before calling ast_$touch. The ast_$touch_segva routine carries out the following steps. It:

1. Gets the AST resource lock, takes the ASTEX specified by the caller and uses it to locate the corresponding ASTE. If no ASTEX has been given, the routine hashes the given UID to get an ASTEX.

2. Compares the UID and segment page number in the ASTE with the values specified by mst_$touch. If they match, the routine reads the in transition bit in the ASTE to make sure that no other caller is using the ASTE; if the bit is set, ast_$touch_segva suspends, waiting on the AST eventcount for the bit to clear. Once the ASTE is no longer in transition, the routine must obtain a new ASTEX (by hashing the UID) because the ASTE on which it was waiting may have been deactivated during the waiting period.

3. Takes the following actions when the target ASTE has cleared in transition:

   - Increments the ASTE hold count; this ensures that the segment will not be deactivated during the call to ast_$touch

   - Returns to mst_$touch with a probable ASTEX to be passed to ast_$touch.

If ast_$touch_segva cannot locate an ASTE that corresponds to the given UID, it activates an ASTE for the segment by calling ast_$activate, and makes associations the memory management tables and the faulting virtual addresses. The routine holds the AST resource lock while it sets up the tables; it releases the lock before it returns to mst_$touch with the ASTE held and with a probable ASTEX.

The ast_$touch routine locates the ASTE for the segment in the same manner as ast_$touch_segva:

1. Gets the AST resource lock and locates the ASTE that corresponds to the object segment via the ASTEX or a UID hash.

2. Checks for in transition on a matching ASTE, or activates a new ASTE if it cannot locate a match.

3. Increments the ASTE hold count and relinquishes the AST resource lock, once it obtains an ASTE.

The AST manager (via ast_$touch and an internal routine *touch*) next carries out some checking (described in later sections) and calls pmap_$touch to bring the requested pages into memory from disk or from another location in the network. Note that the the AST manager (touch) does *not* have the AST lock when it calls pmap_$touch.

13.1.3. PMAP Page Fault Handling

The pmap_$touch routine fetches pages from physical memory, from disk, or from elsewhere on the network. The routine first obtains the resource lock pag_$lock.

13.1.3.1. Page Locking

Most page fault handling is carried out under the pag_$lock resource lock. The page resource lock must be held by any routine that makes a change to the state of the paging database: the PMAP and MMAP. However, the page lock cannot be held during I/O; releasing the lock allows other processes to run while the I/O completes. Relinquishing the page lock means that physical pages allocated may be available for page stealing. Therefore, the modules involved in page fault resolution always set the fields in the MMAPE for the physical page to unavailable, making the page wired, and set the in transition field in the PMAPE; setting these fields prevents the MMAP page replacement routine from stealing the physical pages into which I/O is occurring, and prevents other PMAP callers from seeing those PMAP entries.

13.1.3.2. Determining the Type of Page Fault

Once pmap_$touch obtains the page lock, it gets the PMAP for the active segment and examines the PMAPE for the first page requested. This page can be in one of the following states:

- The first page is in transition; someone else is using it. If pmap_$touch finds the first page in transition, it releases the page lock and suspends itself waiting on the page-in-transition eventcount. When the page clears the in transition state, pmap_$touch restarts its first page examination.

- The first page is resident -- the object-to-physical page association already exists; this is a resident page fault or a sharing fault

- The first page is not resident, in which case it can be:

  - Non-resident and existing on a remote disk; this initiates a remote page fault

  - Non-resident and existing on local disk; this is the typical page fault

  - Non-resident with no backing storage on disk; this state initiates a growth fault

  - Non-resident with invalid data in its backing storage on disk; this is a null page fault

The next section explains the typical page fault -- how pmap_$touch operates when it finds that the first page is not resident and exists on local disk.

13.1.3.3. Fetching Pages from Disk

When pmap_$touch determines that the first page is not resident and exists on disk, it next determines how many more of the pages requested really can be touched by by reading the rest of the PMAPEs for the active segment. The routine only attempts to touch consecutive pages, beginning with the first page requested and stopping at the point that:

- It exceeds the maximum touch ahead count that the MST manager has specified in the call to ast_$touch

- The touch ahead count causes a segment boundary to be crossed

- It finds a page in transition

- It finds a page that is already resident in memory

- It finds a null page

The routine then sets the PMAPEs for the pages it is going to read to in transition so that it can release the pag_$lock resource lock, and then calls its internal routine alloc to allocate enough physical pages to hold the pages to be read into memory. The alloc routine in turn calls the MMAP manager (mmap_$allocate), which actually allocates the pages from the free list or by its replacement algorithm.

If the MMAP manager cannot locate enough physical pages to satisfy the number of pages that alloc has requested, it indicates to alloc that there are no physical pages available. The alloc routine then releases the page lock and advances the page purifier's eventcount, which awakens the purifier to free up some physical pages by writing their contents out to disk. The alloc routine waits for the purifier to return it some physical pages; when it does, the routine then obtains the page lock again and returns to its caller with the page lock held.

Once enough physical pages have been allocated, pmap_$touch unlocks the page_$lock lock and calls the disk driver (disk_$read_ahead) to get the pages off disk into physical memory. When the disk driver returns, pmap_$touch then calls the MMAP manager (mmap_$install_list) to update the MMAP with the new physical pages, and updates the PMAPE for each newly resident page by:

- Copying in the disk addresses (low for reverse-mapped MMUs)

- Setting the resident bit

- Setting the valid bit (on forward-mapped MMUs)

The pmap_$touch routine also updates the ASTE number of pages resident count.

13.2. Completing the Typical Page Fault

Once pmap_$touch has brought the requested pages into memory, it returns the count and the list of PPNs to the internal touch routine. The PMAP manager has marked these pages as unavailable for replacement by the MMAP manager and in transition. The touch routine returns to mst_$touch a count of the pages made resident, their PPNs, and the ASTEX of the active segment for these pages.

The ast_$touch routine then returns to mst_$touch. On nodes with reverse-mapped MMU hardware, mst_$touch calls the MMU manager (mmu_$install) to install the virtual-to-physical page assocation into the MMU's page frame table. On forward-mapped MMUs, the MMU hardware will read in the virtual-to-physical association directly from the page tables. The mst_ $touch routine on reverse-mapped MMUs also copies the ASTEX returned by ast_$touch into the probable ASTEX field in the MSTE; this step has already been completed in forward-mapped MMUs.

The MST manager next clears the in transition bit for the returned pages, and on forward-mapped MMUs, calls ast_$release to decrement the ASTE hold count that ast_$touch_segva left incremented.

The mst_$touch routine also checks the guard field in the MSTE to make sure that the caller is not trying to reference a guard segment; if a guard segment is being referenced, mst_$touch returns a guard fault error.

Finally, the MST routine returns to the fault interceptor manager, which resumes the process that generated the page fault.

13.3. Handling Growth Faults

A **growth fault** occurs when a new page is being created for an object; its PMAPE indicates that it is not resident and also does not exist on disk (its disk address field in the PMAPE is zero).

Growth fault handling begins in the AST manager when its internal *touch* routine checks for a bounds fault before calling pmap_$touch to *grow* the object. A **bounds fault** occurs when mst_$touch's caller is attempting to access a page that is beyond the current length of the object and the object is not permitted to grow. The touch routine checks the faulting address against the segment's current length; if the address extends beyond the current segment limit, the routine checks the ext_ok argument passed by mst_$touch to make sure that the caller is allowed to add pages to the segment. If it is not, the memory management subsystem returns a bounds fault to the caller.

When pmap_$touch finds, during its PMAPE examination, that the first requested page is a non-resident page with no associated disk block, it calls the BAT manager to allocate a disk block for it (via bat_$allocate). The pmap_$touch routine attempts to allocate the new disk block as close as possible to the other disk blocks belonging to the se⁻ment. Note that there is no touch-ahead for growth faults; the system adds pages to objects one page at a time.

If the page to be created is the first within the segment, or the previous page also has no disk block associated with it, pmap_$touch gets the address of the segment's file map from the ASTE and passes that address to bat_$allocate as a place to start.

Otherwise, pmap_$touch obtains the disk address from the segment page that resides just before the one being grown and passes that disk address to bat_$allocate.

Before calling bat_$allocate, pmap_$touch sets the page in transition and releases the page lock. Once the BAT manager returns a disk address, pmap_$touch:

- Adds the disk address to the PMAPE for the new page
- Sets the null bit to indicate that the contents of the disk address assigned to the segment page is not valid for the "new" physical page
- Indicates in the ASTE that the segment map has been modified
- Writes to the blocks_delta field in the ASTE the number of pages it has added to the segment (the blocks_delta field tracks the number of disk blocks added to the segment since its activation).
- Allocates and zeroes the new page and sends null page status back to ast_$touch.

13.4. Handling Null Pages

A null page is a page whose copy on disk is invalid. The page is not resident and has a disk address associated with it, but the contents of the block at this address are no longer valid for the physical page. If pmap_$touch encounters a null page, it searches through the segment's PMAP for more contiguous null pages, stopping if it encounters anything else.

Then, pmap_$touch puts the null pages it has found in transition in their PMAPEs and calls the alloc routine to allocate some physical pages. Once the alloc routine returns these pages, pmap_$touch then calls an internal routine to invalidate the contents of these physical pages.

Then, pmap_$touch returns the null page status to the AST manager (touch), which checks to see if new pages have been created for the segment. If the segment contains additional pages, and its ASTE is the first one in the linked list of ASTEs for an object, the touch routine updates the current length field in the ASTE. The AST manager eventually copies this length into the linchpin ASTE; see Chapter 11 for a description of the linchpin ASTE.

13.5. Handling Resident Page Faults

If pmap_$touch finds that the first page to be fetched in the PMAP is already be resident in physical memory, an object-to-physical association exists but the virtual-to-physical association in the MMU does not. An object page can already be resident in memory because another process has already brought it into memory, or because the process itself has previously brought it into memory and then unmapped it from its address space. (Unmapping the object removes the virtual to physical associations from the MMU, but the object is still resident in physical memory.)

The pmap_$touch routine handles a resident page fault like the typical page fault except that no disk activity occurs. It marks the pages in transition in the PMAPEs, calls the MMAP manager to mark them unavailable for replacement, and, on forward-mapped MMUs, it also sets the valid bit in the PMAPEs.

13.6. Handling Sharing Faults

Reverse-mapped MMUs can only recognize one virtual-to-physical page association at a time. Although two or more process virtual addresses can point to the same physical page, the MMU can contain only *one* process virtual mapping to that page at a time. Thus, if several processes are all referencing the same physical page, these processes must share the one table entry. Each time one of sharing processes becomes the curren' process and references that physical page with a virtual address, the system must install the correct process virtual address space association into the MMU's page frame table; this procedure is called a **sharing fault**.

The fault interceptor manager (FIM) handles sharing faults. When it receives a fault on a virtual address, it calls the virtual memory management managers to read the memory management tables to determine whether or not the page is resident in memory (the resident bit in the PMAPE is set). If it finds that the page is resident, it assumes it is handling a sharing fault. It then removes the current virtual-to-physical page association from the MMU and installs the new virtual association for that physical page.

The forward-mapped MMU can hold more than one virtual-physical page association, so sharing faults do not occur. When the fault interceptor manager on these systems recevies a virtual page fault, it proceeds directly to the MST manager to resolve the fault.

13.7. Remote Page Fault Handling

When pmap_$touch determines that the page or pages requested by ast_$touch reside on another node in the network, it calls the NETWORK manager to read in the pages from the remote node. The NETWORK manager handles two types of paging requests:

- Page-in requests (multipage reads) for multiple pages, where the NETWORK manager reads several pages at once from remote to local memory. The NETWORK manager can handle a maximum of 16 page reads at once, but the PMAP manager's maximum is 32. Consequently, if the PMAP manager issues a multiple page-in request for more than 16 pages, the NETWORK manager will perform another page-in operation until it gets the requested number of pages. Only when it has satisfied the request will the manager return to the caller.

- Page-out requests (single page writes), where the NETWORK manager sends pages modified at a remote site to their home nodes so that the modifications can be written to local storage. An explicit page flush initiates a page-out request; examples include FILE manager force-writes and AST manager page flushing during ASTE replacement. Note that the system does not currently perform page-outs of more than one page (no multiple page writes) over the network.

The modules in the NETWORK manager that carry out these paging requests are collectively known as the NETWORK **client side** because the NETWORK manager is carrying out operations on the caller's behalf. Figure 13-2 illustrates a remote page-in request; the next sections discuss the sequence of events that occurs when the PMAP manager makes a multiple page-in request to the network manager.

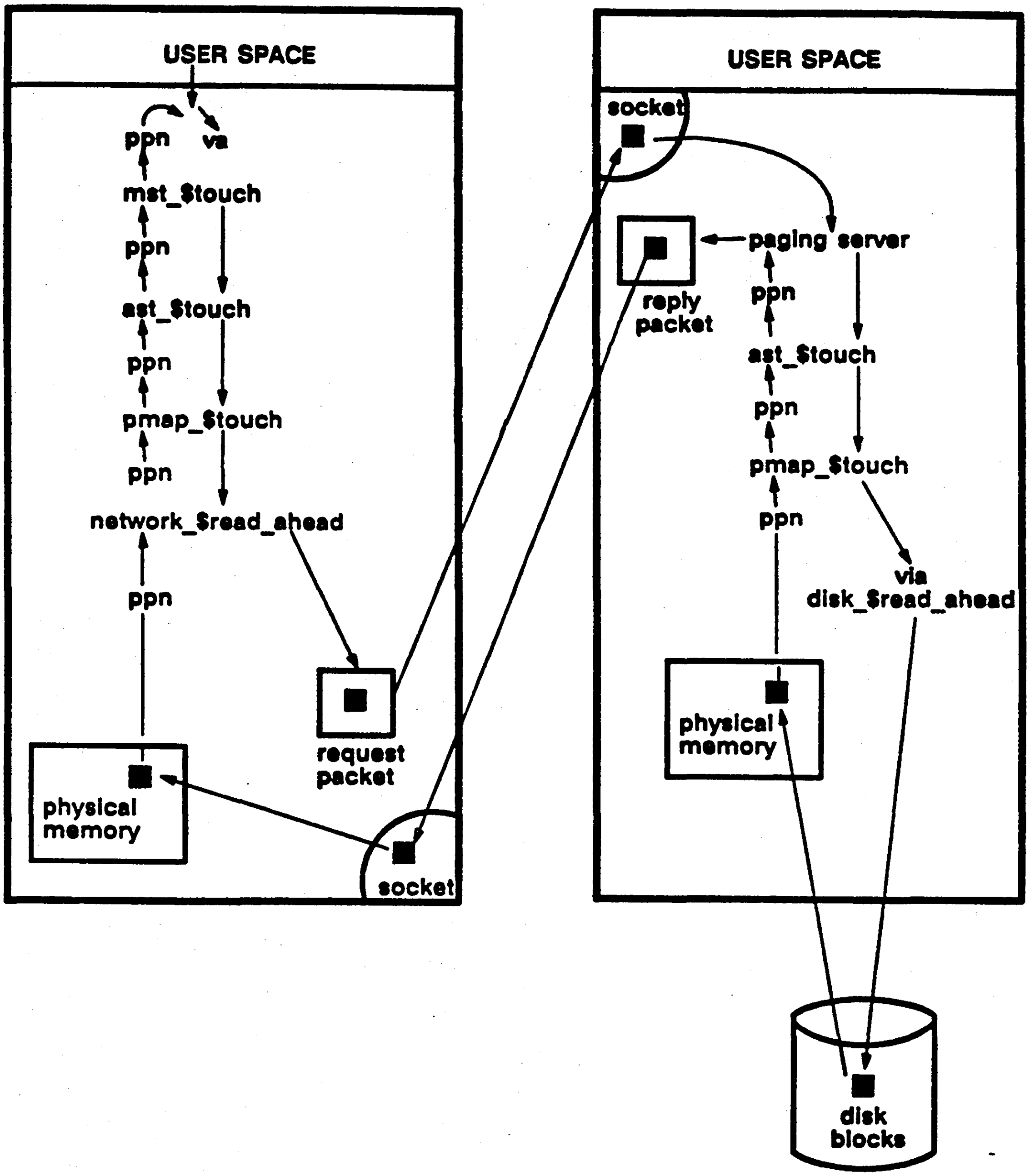


Figure 13-2. Remote Page-In Request

13.7.1. Allocating Network Buffer Pages

The network buffer pool is a pool of virtual pages that clients of the system's low-level IPC service use to hold incoming or outgoing messages. To maintain the pool in a steady state, these clients must allocate as many wired pages to the network buffer pool as they intend to take from it. and they must give the buffer pool these wired pages before they remove any pages from the pool.

Because the PMAP manager is requesting delivery of low-level IPC messages -- the remote pages -- it must follow the convention and allocate pages to the network buffer pool. As a result, pmap_$touch calls the MMAP manager to allocate some physical pages to hold the object pages, then calls the network buffer pool manager (NETBUF) to add these physical pages to the network buffer pool. It then calls network_$read_ahead to carry out the remote request. Chapter 20 describes the network buffer pool and the low-level IPC mechanism in more detail.

13.7.2. NETWORK Client Side Paging Operations

To initiate the network read, the network_$read_ahead module allocates a reply socket to which the socket manager will send a message indicating that the requested pages have arrived in the network buffer pool and then builds the request data into a network packet; in this case, a multiple page-in request.

The network_$read_ahead routine then calls another network manager module (send_it) to transmit the packet through the ring to its destination, and waits for the reply packet from the remote node by waiting on the eventcount associated with the reply socket it has allocated.

13.7.3. NETWORK Server Side Paging Operations

The remote paging server is the "server" part of the network manager; it retrieves paging request packets sent across the network and then then processes the packets according to their request type, which, in this discussion, is a multiple page-in request.

13.7.3.1. Processing the Page-In Request

For a multiple page-in request, the paging server gets the object UID and the segment page numbers within the object from the block headers passed in the packet, and then calls ast_$touch to fetch the pages. In the remote case, there is no ASTEX for ast_$touch to use, so it to obtain the index to the ASTE.

13.7.3.2. Concurrency Control Checking

The AST manager locates the correct active and not *in transition* ASTE just like a local page-in request, then performs concurrency control checking. It compares the node ID stored in the lock key field in the ASTE against the node ID of the requesting node (specified in the call) to make sure that the node ID of the page-in requestor is the same as the node ID that activated, and thus locked, the segment (when the segment is activated, the caller's node ID is stored in the ASTE. In the local case, the node ID is always node_$me.) If the lock key check passes, the AST manager transfers control to the touch routine, which calls pmap_$touch to get the pages. See Chapter 6 for more information on concurrency control.

13.7.3.3. Fetching Pages for a Remote Request

If the requested pages are already resident, pmap_$touch carries out the same sequence of steps described in Section 13.5. The only difference between the local and remote case is that the touch routine returns the DTM stored in the ASTE along with the list of PPNs to its caller, who, in this case, is the remote paging server. If the first page that pmap_$touch tries to touch is not resident, pmap_$touch goes through the same search process as for a local page fault, eventually calling alloc to allocate some free pages from physical memory. In both local and remote cases, the alloc routine calls the MMAP manager (mmap_$allocate) to locate some free pages. However, once mmap_$allocate determines that the paging server is the calling process, it searches the remote paging pool for the physical pages; if there are none available, or it can't obtain enough to satisfy the request, it returns the PPNs of the pages it has obtained to the alloc routine.

Then, instead of calling the demand-based purifier, which happens in the local case, the alloc routine calls the purify_$local module to free up some more physical pages; purify_$local, unlike the purifier, will purify *only* those physical pages that are associated with local objects. If the module encounters a physical page associated with a remote page, it skips over it.

The alloc routine calls purify_$local to avoid deadlocking between the paging server process on one node and the purifier process on another node, where each process is suspended waiting for the other to complete.

Once the alloc routine returns some free physical pages, pmap_$touch proceeds as in the local case: it unlocks the page lock, calls the disk manager, updates the MMAP and PMAP and returns the pages wired and in transition to ast_$touch. The ast_$touch routine then updates the DTM in the ASTE and returns the PPNs to the paging server.

The paging server next builds the page-in reply packet using the block header passed to it (note that the reply packet contains a new DTM) and clears the in transition bits in the PMAPEs, and calls the network manager (send_reply) to carry out the network transfer. Note that the paging server passes the PPNs directly to the network manager; it does not need to allocate pages from the network buffer pool on a page-in reply.

13.7.4. Remote Page Fault Completion

Once the page-in reply packet arrives in the network buffer pool, the socket manager notifies the network_$read_ahead module, which then retrieves the PPNs of the physical pages from the network buffer pool and returns them to pmap_$touch. The pmap_$touch routine then resolves the page fault as if it obtained the pages from local disk and follows the completion path outlined in Section 13.2.

Again, system convention dictates that low-level IPC clients remove all the pages they put into the network buffer pool when their network operations are complete. If the number of remote pages returned to pmap_$touch is less than the number of pages that pmap_$touch added to the network buffer pool, the routine takes those pages back from the network buffer pool and frees them in the MMAP.

13.7.5. Network Errors During Remote Page Faults

All remote paging server operations are **idempotent**; that is, in the event of an error, the remote paging server initiates the same sequence of operations, regardless of the type of error that occurs. Consequently, should a network error occur during a remote paging server reply to a client request, the server will simply retry the operation once the error condition has been resolved.

13.7.6. Creating Additional Paging Servers

The system can create additional paging servers on request (via the NETSVC command). Multiple remote paging servers are not necessary when there is no parallel remote access to a single node. However, if a node incurs frequent remote access, as it would if it were a file server acting as partner to several diskless nodes, then multiple remote paging servers can improve performance by overlapping paging I/O operations; for example, one paging server can perform a disk transfer to memory, while the other server carries out a network transfer from memory.

Creating multiple remote paging servers has some disadvantages. Each remote paging server uses up a process control block (PCB) that would otherwise be available to a level 2 process. Thus, creating several paging servers reduces the number of processes available to user mode. Multiple remote paging servers also tie up more physical memory and wired stack pages. And, finally, faster parallel remote access for the group means less disk bandwidth for each group member.

Chapter 14
Process Management Overview

A process is fundamentally a *thread of execution* defined by its program counter (PC) and stack pointer (SP). Although a process has additional context, its thread of execution is the basic context that distinguishes it from other processes.

Process context is divided into two levels. Level 1 context is a process's physical state, which consists of its M680x0 processor context. Level 2 context is a process's virtual state, which it acquires through the virtual memory support that exists between the two process levels. Every process in the system has level 1 context, and some processes in the system *only* have level 1 context.

Processes with level 1 context are called level 1 processes. Level 1 processes can only run supervisor-mode code; that is, they only run the protected AEGIS kernel services. Consequently, level 1 processes run exclusively in supervisor mode. In addition, a level 1 process does not possess its own virtual address space. Instead, all of the level 1 processes share supervisor global address space.

A level 2 process is a level 1 process with additional virtual state: its own private virtual address space. Level 2 processes can run in either user mode or supervisor mode; they run in supervisor mode via the SVC mechanism described in Chapter 19. The level 2 process layer exists primarily to create the environment for user-mode programs. However, user-mode AEGIS system services also run level 2 processes; for example, the display manager and the mailbox (MBX) helper.

The separation of process context into two levels allows the AEGIS kernel to use the process mechanism to carry out operating system services such as virtual memory management while simultaneously permitting user-mode processes to take advantage of virtual memory. In principle, the two-level design permits an unlimited number of level 2 processes whose virtual context can be swapped in and out of, or *bound and unbound* from their level 1 hardware context; the system saves the virtual context at unbind. Currently, however, the AEGIS system does not save the context of unbound level 2 processes; of the 32 available processes, 25 can be augmented to level 2 processes, and 8 remain level 1 processes in order to carry out AEGIS kernel functions. Level 2 process unbinding is equivalent to process deletion; when one of the level 2 process is unbound, its context is deleted, and the level 1 process context underneath it is deleted as well. Figure 14-1 shows the relationship between process managers and process levels when a user directs the display manager to run the shell program.

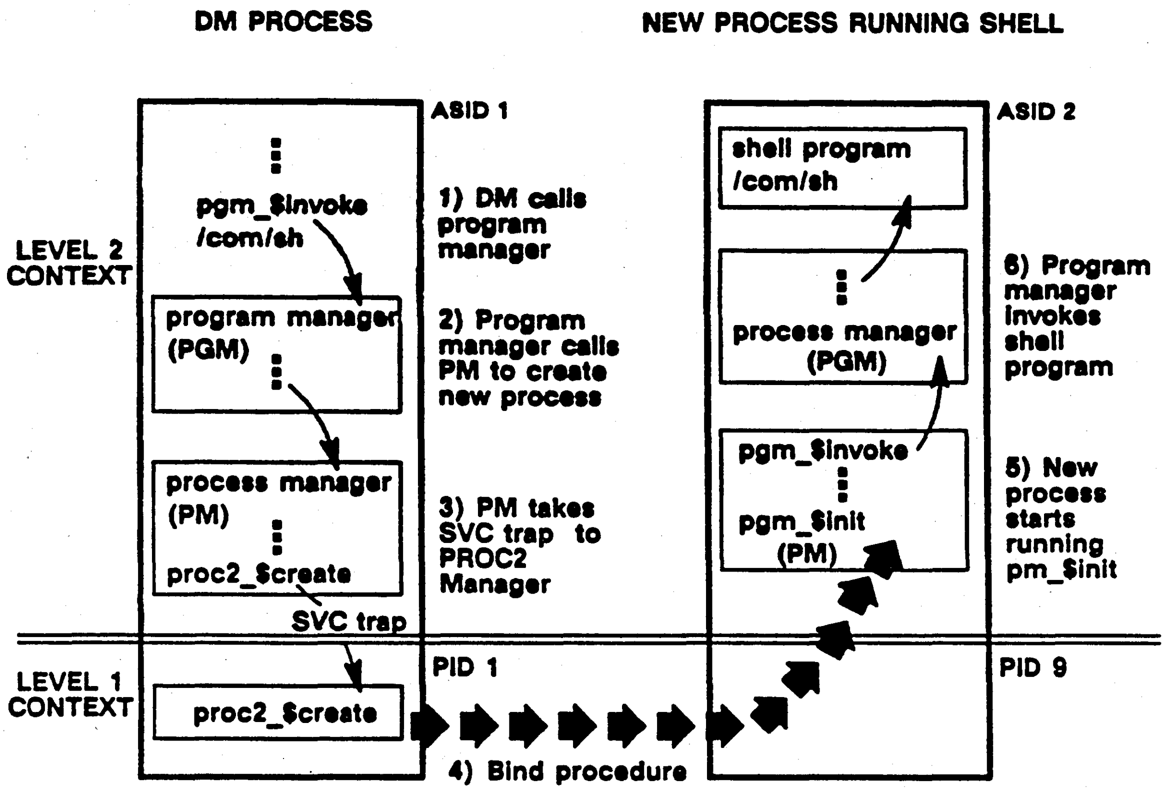


Figure 14-1. Relationship between Process Levels

Several managers make up the process management environment in the AEGIS kernel. They are:

- The level 1 process manager (PROC1), which creates and deletes level 1 processes, maintains level 1 process context and data structures, and carries out system scheduling and dispatching operations.

- The level 2 process manager (PROC2), which creates and deletes level 2 processes, manages the level 2 process UID namespace and context, and plays a role in asynchronous fault delivery.

- The level 1 and 2 eventcount managers (EC and EC2), which synchronize operations between level 1 process and between level 2 processes, respectively.

- The kernel-level mutual exclusion manager (ML), which provides mutual exclusion of kernel resources to level 1 processes.

- The kernel fault interceptor manager (FIM), which handles traps, interrupts and faults and fields faults to user processes.

- The SVC catcher, which fields traps from user mode to supervisor mode.

The next chapters discuss these managers in more detail.

Chapter 15
Level 1 Process Management

This chapter discusses level 1 process context and PROC1 manager operations. Level 1 process context consists of processor state and scheduling information. PROC1 operations include maintaining the level 1 context, creating and deleting processes, and carrying out process scheduling.

15.1. Processor State

Processor state is the information a process places into the processor when it becomes the **current process**; that is, the process that is currently using the CPU. Processor state includes:

- Pointers to the process's stacks.
- The process's address space ID.
- The amount of time during which the process has exclusive use of the processor, called **process virtual time, CPU time** or **time slice**. This value is stored in the system's process virtual time clock.

The next sections discuss processor state in more detail.

15.1.1. Process Stack Pointers

The M680x0 processor register set consists of address registers A0 through A6, data registers D0 through D7, program counter (PC), user (USP) or supervisor (SSP) stack pointers, and the processor status register (SR), which indicates whether the processor is running in user or supervisor mode, and also indicates condition codes and interrupt level. Of these processor registers, the PROC1 manager needs only to preserve the stack pointer on context switch; the rest of the registers are saved as part of the standard calling sequence imposed upon all the languages that run on the AEGIS system.

Processes with an active USP are considered to be running in user mode, and are thus level 2 processes. In this case, the stack is pageable, private to the process, and allocated from the stack object by the PROC2 stack allocation module. Any process with an active SSP is running in supervisor mode. Each level 1 process has its own wired supervisor stack; the PROC1 bind module allocates these per-process supervisor stacks from a whole cloth area in supervisor global space.

15.1.2. Address Space ID

Level 2 processes have their own private virtual address space; the system keeps track of each process-private virtual address space by assigning it an ASID. Because this virtual address space is unique to each process, the same virtual address can refer to different objects, depending upon the ASID of the process making the reference. For example, virtual address 20000 in ASID 1 can map a completely different object than virtual address 20000 in ASID 2. Consequently, the system needs to identify the current process's private address space to the memory management unit (MMU) so that it can translate the virtual references issued by the processor to the correct process virtual addresses.

At context switch, also called process **dispatching**, the PROC1 manager installs the process's ASID into an MMU control register. When the process makes a reference to its private address space, the MMU hardware compares the current ASID to the ASID/virtual address pairs in its hardware tables to find the correct per-process virtual address.

Global virtual addresses, on the other hand, are common to all processes. Processes that reference the same global address will all reference the same object, regardless of each process's ASID. Consequently, the MMU can disregard the process ASID on a reference to a global address.

The system identifies global address space with ASID 0; however, ASID 0 acts as a mapping key for the MST manager rather than an MMU translation key. See Chapter 9 for more details.

15.1.3. Process Virtual Time Clock

The AEGIS system keeps track of process virtual time with the virtual time clock. This clock is one of three 16-bit timers that reside on one hardware timer chip. The virtual time clock advances, or *ticks* every eight microseconds. The clock accumulates the current process's CPU time and stores its remaining **time slice**; that is, the amount of time that left to the process during which it has control of the central processor. The amount of time slice is determined by the process scheduler (see Section 15.6.3). When a process exhausts its time slice, the timer overflows, generating an interrupt. The dispatching procedure switches the values kept in this timer from one process to another.

15.2. Scheduling State

A process's scheduling state determines the conditions under which a process becomes the current process. Scheduling state includes:

- Process priority
- The set of resource locks the process currently holds
- Process state: whether is is bound, waiting, suspended, suspend pending, or has incurred time slice end while holding a resource lock
- The amount of CPU time remaining to the process (its remaining time slice)
- The amount of CPU time since the process last waited on an eventcount

The next sections discuss scheduling state in more detail.

15.2.1. Process Priority

Process priority is an integer value that ranges from 1 to 16; 16 is the highest priority, and 1 is the lowest. Some processes have a fixed priority value; for example, the DM always has priority 16.

Both the system and users (with the PPRI command) can set priority bounds for a given process; the process can only attain priority values within the specified range. The system assigns newly created level 2 processes priority bounds of 3 to 14; this allows users to create background processes (high = 1, low = 1) whose operation will not interfere with (in terms of CPU time, at least) normal newly created level 2 processes. Note, however, that giving processes non-overlapping priority ranges can lead to process deadlocks, where the higher-range priority process permanently blocks processes with the lower priority range from being able to run. Also note that a forked process duplicates its parent's priority.

The PROC1 manager creates level 1 processes with priority bounds between 16 and 1. Level 1 processes that become special system processes also have priority bounds of 16:1.

The null process always has priority 0 so that it does not contend with any priority 1 processes, and so that there will always be a current process. If there are no other processes that are ready to run, the system runs the null process, thereby maintaining the scheduling mechanism.

15.2.2. Resource Locks

Resource locks are the means by which level 1 processes synchronize access to system resources; a process can only obtain a resource lock while it is running in supervisor mode. There are 32 resource locks available, 26 of which are currently used. Resource locks are ordered in priority from 0 to 25; the higher the number, the higher the lock's priority.

Resource locks are the system's way of detecting level 1 process deadlocks; that is, when process A gets lock 1 and tries for lock 2 while process B gets lock 2 and tries for lock 1. Processes can only obtain locks in increasing order, and must release them in the order in which they have been obtained. If a process gets a high lock and then tries to get a lower one, the system crashes with lock violation status. A process that holds a resource lock cannot be suspended; arbitrary suspension of a process holding a resource lock could tie up an important resource and possibly cause the system to fail.

For the most part, resource locks control access to I/O devices and system data structures. Disk and network I/O devices have high priority locks to ensure that processes have been able to gain access to all the other resources they need (by locking and unlocking these resources with their associated resource locks) before they proceed with disk or network I/O.

The highest resource lock is the clock process ready lock (time_$lock). This lock is not really a resource lock; it exists to give a priority boost to the clock process. Because resource locks are used in process scheduling, the clock process holds this highest lock to ensure that the scheduling algorithm will always run it if it is ready to run.

Special CPU locks exist for nodes with dual M68000 processors (the DN400 model). These nodes contain two CPUs; the second CPU (CPU B) handles all page fault resolution. Consequently, processes that run on DN400 nodes use these CPU B locks to control access to CPU B; because processes running on CPU B cannot take a page fault, a process that wants to run on CPU B must ensure that it will not fault before it can run on B.

Table 15-1 shows the available resource locks and the system resources they affect. The resource lock word in the process control block identifies the resource locks that a process currently holds.

Table 15-1. Resource Locks

Lock Name	Lock No.	Resource Locked
network_$serv_lock	0 (low)	
mt_$lock	1	Magtape drive
xpd_$lock	2	Cross-process debug tables
term_$lock	3	Display
proc2_$lock	4	PROC2 database
file_$lock_lock	5	Lock manager tables
ec2_$lock	6	Eventcount database
smd_$respond_lock	7	Screen manager driver
smd_$request_lock	8	Screen manager driver
pbu_$lock	9	Multibus controller
acl_$lock	10	Access control list
proc1_$create_lock	11	PCB array
onb_$lock	12	Faulted to CPU B
bok_$lock	13	Runnable on CPU B
disk_$mnt_lock	14	Disk volume
vtoc_$lock	15	Volume table of contents
bat_$lock	16	Block availability table
ast_$lock	17	Active segment table
pag_$lock	18	Page in memory
sm_$lock	19	Storage module device
flp_$lock	20	Floppy disk
win_$lock	21	Winchester disk
ring_$xmit_lock	22	Ring controller
ml_$free7	23	Unimplemented lock
time_$proc_lock	24	Clock process database
time_$lock	25 (high)	Clock process ready

15.2.3. Process State

A process can be in one or more of the following states:

- Bound, which determines whether the process is runnable at all; a bound process has a PCB and an initialized stack, and is ready to run

- Waiting, which determines whether or not the process wants to use the CPU; a process in the wait state is waiting for an eventcount to reach a specific trigger value. It does not currently need to use the processor, but will resume CPU contention when the eventcount is satisfied. (Chapter 17 discusses level 1 and 2 eventcounts in detail.)

- Suspended, where process context has been frozen and the process must be explicitly resumed.

- **Suspend pending, where the process is in a state that prevents immediate suspension, but will be automatically suspended once that state has passed. A process cannot be suspended immediately if it holds a resource lock.**

- **Time-slice end with resource lock held, where the process has exhausted its time slice, but still holds one or more resource locks. This state becomes important when the PROC1 manager is releasing resource locks, because holding a resource lock affects a process's scheduling priority even at its time-slice end.**

15.2.4. Time Slice

The time slice is the length of process virtual time during which a process has exclusive use of the processor. The time slice allotted to a process is inversely proportional to its priority; the higher a process's priority, the smaller its time slice. Processes with priority 16 get 1/10 second time slices, whereas priority 1 processes get 1/2 second time slices. Because the virtual time clock ticks every 8 microseconds, 1/2 second is the largest time slice a process can obtain.

The dispatcher loads the virtual time clock with a process's time slice when it becomes the current process; when the time slice expires, the timer chip generates an interrupt that the PROC1 manager fields.

15.3. Special Level 1 Processes

Some level 1 processes remain level 1 processes; they never acquire the per-process address space that makes them level 2 processes and always run entirely in supervisor global address space. These processes are *internal* to the AEGIS system; they are processes devoted to maintaining low-level system functions in virtual memory management, network management, and so on. Because these processes do not possess the additional, pageable level 2 context, they are cheaper to run.

Special level 1 processes include:

- The initial system process used during system intialization to create the level 1 and 2 process databases. (Unlike the other special level 1 processes, this process becomes a level 2 process after initialization is complete.)

- The null process, which figures in process scheduling.

- **The clock process, which keeps track of a node's real time so that processes can wait on real-time events.**

- **The purifier process, which writes modified pages out to disk as part of virtual memory management operations.**

- **The terminal helper, which handles keyboard input.**

- **The network receive server, which handles all incoming messages to a node.**

- The network paging server(s), which handles remote page faults.

- The network request server, which is a master server that invokes other servers, such as the file server and ASKNODE server, as necessary.

- The memory lights process, which runs the memory lights utility.

- The internet routing process and IIC guardian process, which handle packet transmission to other networks in an internet. These processes only run on routine nodes; see Chapter 24 for more information.

The processes created during system initialization are identified by PIDs 1 through 8; note that the correspondence of PID to these internal level 1 processes is arbitrary and may change across system revisions. If the system is called to create additional internal level 1 processes, such as the memory lights process, internet processes, or additional paging or request server processes, it allocates another level 1 PCB/PID pair from the pool of remaining PCBs that have not yet been bound to level 2 processes.

15.4. Level 1 Process Data Structures

Because the AEGIS system is designed to promote information hiding and modularity, it does not store all process context in one control block. In AEGIS, each manager is responsible for storing only the information it needs to carry out its functions. Thus, the PROC1 manager maintains a small process database that stores scheduling and dispatching information. The PROC1 data structures that store this data include the process control block (PCB) and PCB array, the current process global variable and the ready list. All of these data structures are wired into physical memory.

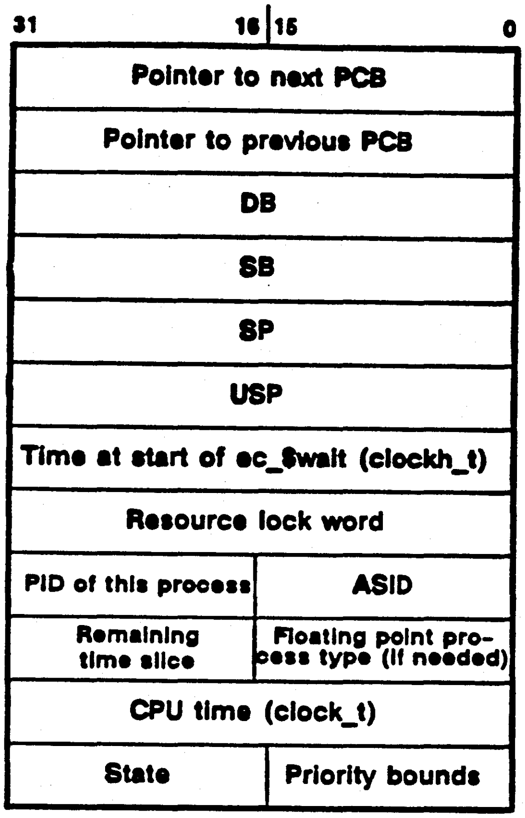
15.4.1. Process Control Block

The process control block stores a process's state and scheduling information; one PCB exists for each process. Each PCB is linked to an array of PCBs; this array is indexed by PID. Figure 15-1 shows the fields within the PCB.

15.4.2. Ready List

A process is considered **ready** to run when it is bound and not waiting or suspended. The scheduler keeps track of ready processes by placing them into a list. This **ready list** is a doubly linked list of process control blocks ordered conceptually by scheduling priority from highest ready process to lowest ready process, and implemented as a circular buffer. Processes are inserted into the ready list through pointers to their PCBs. PCBs on the ready list are linked together so that any PCB points forward to the next highest PCB and backward to the next lowest PCB on the list. Because the list is circular, the highest ready PCB points back to the lowest ready process, while the lowest ready process (which is always the null process) points back to the highest ready process. The global address proc1_$ready_pcb identifies the highest ready process, while proc1_$current_pcb identifies the process that is currently running.

Using a doubly linked list allows the system to remove a PCB from the middle of the ready list by simply locating its immediate lower and higher neighbors, instead of chaining down from the head of the list to find the next highest PCB.



31 16	15 0
Pointer to next PCB	
Pointer to previous PCB	
DB	
SB	
SP	
USP	
Time at start of ec_$wait (clockh_t)	
Resource lock word	
PID of this process	ASID
Remaining time slice	Floating point process type (if needed)
CPU time (clock_t)	
State	Priority bounds

Figure 15-1. Process Control Block

15.4.3. Process Type ID

The process type attribute identifies the level 1 processes reserved to the system and distinguishes, for the remaining processes, whether they are level 1 or level 2 processes. A process's type is stored in the type list structure; this list is indexed by PID. Table 15-2 shows the process types in the AEGIS system.

Table 15-2. AEGIS Process Types

PID	Process Type
1	Initial system process
2	Null process
3	Clock process
4	Purifier process
5	Terminal server
6	Network receive server
7	First network paging server
8	First network request server
9-32	Additional system processes, level 1 process or level 2 process

Some AEGIS managers need to determine process type in order decide how to proceed during a given operation. For example, the MMAP manager must determine that its caller is the remote paging server before it allocates physical pages because the remote paging server can only use pages from the remote paging pool. Other managers that depend on process type identification are the AST manager, the directory manager, the PMAP manager, and the remote file (REMFILE) manager.

15.5. PROC1 Manager Operations

Level 1 process manager operations include:

- Process creation, which includes binding, stack allocation, and creating new level 1 processes

- Rescurce lock handling

- Process suspension

- Process scheduling, dispatching, and interrupt handling

None of these PROC1 operations are exported to user space. (There are, however, PROC1 inquiry operations that the PROC2 manager can call on a user-mode program's behalf to obtain information about a level 1 process.)

15.5.1. Process Creation and Deletion

Process creation of level 1 processes consists of binding a process to a PID and PCB, and allocating a supervisor stack to the process. Process deletion unbinds the process and can also result in its supervisor stack being freed.

15.5.1.1. Binding and Unbinding

Binding and unbinding are the low-level process creation operations. The PROC2 manager actually creates the level 2 process, then calls the bind procedure to make the new process ready to run. The bind procedure (procl_$bind) locates an unbound PCB and initializes the PCB and the stack to start execution in the new process at a specified procedure entry. It also initializes the process performance counters when a process is bound to a PID.

Unbinding is the lower half of the PROC2 delete operation; any time a level 2 process is deleted, its PROC1 context is unbound as well. (AEGIS does not save level 2 process context in an intermediate state.) The unbind procedure (procl_$unbind) suspends the process (if necessary), frees its supervisor stack (if it is using a big stack), and unbinds the PCB, thereby making it available for use when another level 1 process is created.

15.5.1.2. Stack Allocation

The system initialization procedure allocates a pool of pages within a whole cloth area in supervisor global address space to be used for the per-process supervisor stacks. Pages of this stack pool are allocated to a level 1 process during the process bind operation. The PROC1 manager's stack allocation routine (procl_$alloc_stack) allocates and wires pages from the stack pool to each newly bound level 1 process. The routine calls the PMAP manager to retrieve the pages; on reverse-mapped systems it also calls the MMU manager (mmu_$install) to install them into the MMU.

The bind procedure specifies how many pages to allocate for the per-process stack; currently, there are only two sizes. Level 1 processes that will not take page faults request a smaller stack size, which is less than a page. Processes that take page faults or run in user space need a larger stack, which is currently three pages.

If a level 1 process has been allocated a large stack and it is unbound, the PROC1 manager (via proc1_$free_stack) will free the large stack for re-use.

15.5.1.3. Creating Special Level 1 Processes

The system calls a special PROC1 operation (proc1_$create) when it needs to create special system process such as additional remote paging servers. The PROC1 create operation, in turn, uses the stack allocation, bind, and resume operations to carry out its function.

15.5.2. Resource Lock Handling

Level 1 processes call PROC1 resource lock operations to set and clear bits in their resource lock set. Setting a resource lock bit raises the process's CPU scheduling priority; in addition, a subset of these bits makes the process runnable on the CPU B processor on DN400 systems. A process must set resource lock bits in increasing order; setting a lock out of order is illegal and causes the system to crash with the error proc1_$ill_lock.

Because the resource locks that a process holds affect its scheduling priority, clearing a bit in the current process' resource lock set may cause the process to lose the CPU as a result. In addition, if the PROC1 manager clears the last resource lock bit that a process has set, it can trigger completion of a pending suspend operation.

Calling the PROC1 manager to set a lock does not lock a resource; these PROC1 calls exist primarily to alter a process's scheduling priority by setting and clearing the resource lock bits. Processes call the kernel mutual exclusion manager (ML) to actually lock the resource associated with the lock bit.

The PROC1 manager also contains a special clear lock function used for CPU B to A transitions. This function clears the lock, but also prevents dispatching in case the current process loses CPU priority. The fault interceptor manager (FIM) is the only manager that calls this function; it is called following page fault resolution and prior to switching back to the A processor.

15.5.3. Process Suspension

The PROC1 suspend operation makes a ready or waiting process non-dispatachable; it removes a ready process from the ready list and marks a waiting process so that that it cannot be placed on the ready list when it is awakened. The process must exist, be bound, and not already be suspended; otherwise, the module will simply return with no suspension performed. Interrupts must be disabled during the suspend operation, and the caller should eventually dispatch (proc1_$dispatch).

A process is most often suspended by:

- The display manager, which suspends a process when the user issues the DS command

- The cross-process debugging program (XPD), which suspends the process so that it can look through the process's stack

- Itself on process deletion; this is the mechanism that procl_$unbind uses to delete a level 1 process

If the process is waiting on an eventcount, the suspend operation sets the suspend bit and takes the PCB off the ready list. When the eventcount on which the process is waiting advances, the level 1 eventcount manager (via ec_$advance), as part of its operation, reads the state bits in the PCB and notes that the suspend bit is on, so does not return the process to the ready list.

If the process to be suspended holds a resource lock, the suspend operation sets the suspend pending bit in the PCB state word. The PROC1 suspend inquiry module should be used following a call to suspend that did not complete immediately due to resource locks held. If the process is not in a wait state and holds no locks, the suspend operation suspends the process by removing it from the ready list. In all cases, the suspend operation advances the global suspend eventcount (procl_$suspend_ec) when suspension actually occurs.

Suspended processes are brought out of suspension by procl_$resume. In order for procl_$resume to be successful, the process must exist, be bound, and be suspended. If these conditions are not met, the appropriate status code will be returned. If the process to be resumed is not in a wait state, the resume operation adds the PCB to the ready list and dispatches.

15.6. Implementation of PROC1 Operations

All of the PROC1 routines follow this code sequence when carrying out their operations:

1. Check call validity

2. Disable interrupts by setting interrupt level bit in processor status word (SR) to 7; consequently, the processor will not service any interrupts that occur below interrupt level 7.

3. Modify the PCB to reflect the operation (suspend, set lock, and so on)

4. Reorder the ready list (scheduling)

5. Dispatch (make the first ready process current)

Previous sections have discussed the PROC1 operations that modify the PCB; the next sections discuss dispatching, interrupt handling, and scheduling.

15.6.1. Dispatching

Dispatching switches the processor from one process to another: it makes the context of the first process on the ready list *current* while saving the context of the previous current process. Dispatching is also called process exchange or context switching.

All of the PROC1 operations cause the ready list to be reordered. In most cases, when the ready list is reordered, the highest priority process changes. As a result, the PROC1 manager contains an internal routine known as the dispatcher that locates the highest ready process and makes it the current process.

15.6.1.1. The Dispatching Algorithm

The dispatcher must be called with interrupts disabled to provide interrupt exclusion. If the dispatcher were to run with interrupts enabled, the pushing and popping of multiple interrupt frames on the stack during interrupt servicing and subsequent stack switching during dispatching could cause the same interrupt to be handled twice. When the dispatcher completes, it returns with interrupts enabled.

The dispatcher proceeds in the following sequence. If the process that is ready to run is already the current process, the dispatcher simply enables interrupts and returns; otherwise, it dispatches the process that exists at the head of the ready list (indicated by the global proc1_$ready_pcb).

The dispatcher first saves the processor state of the current process; that is, it saves the registers that must be preserved across procedure calls according to run-time conventions. Currently, it saves the USP in the PCB and pushes the registers A5, A6, and A7 onto the stack. It then accumulates the CPU time that the process has used and places it into the PCB.

The dispatcher then establishes the processor state of the ready process by:

1. Placing the new process's time slice into the virtual timer chip

2. Restoring registers and the user stack pointer

3. Making this the current process by adjusting proc1_$current to point to the new process's PCB

4. Switching ASIDs (proc1_$switch_asid calls mmu_$install_asid)

Finally, the dispatcher enables interrupts and returns.

15.6.1.2. Dispatching and the Null Process

If there are no other processes on the ready list that are ready to run, the dispatcher runs the null process, thereby maintaining the scheduling mechanism. The null process is always ready, and always has the lowest priority (0) so that it does not contend with any priority 1 processes. The null process *cannot* take a page fault, or the system crashes. Removal of the null process destroys processor management operation. When the null process runs, it checks to see if the ready list is out of order (if it is running and shouldn't be), and crashes the system if the list is out of order.

15.6.2. Interrupt Handling

The M680x0 processor supports seven interrupt levels (IL). The AEGIS system implements software interrupts as follows:

- Every interrupt service routine (ISR) runs at IL 6
- All interrupts are enabled at IL 0
- All interrupts are disabled at IL 7

The hardware records the interrupt level in the processor status register.

The AEGIS system does not support interrupt priority levels for interrupt routines for the following reasons:

- The M680x0 processor does not support an interrupt stack.
- The AEGIS system is designed to keep wired memory at a minimum.
- The AEGIS system is not designed to be a real-time system.

Unlike other hardware processors, the M680x0 processor saves interrupt context on the current supervisor stack. When an interrupt occurs, the processor simply pushes the interrupt context (formatted as an interrupt frame) onto the stack pointed to by the SSP that exists in the processor stack register. If AEGIS were to implement multiple interrupt levels for drivers, many simultaneous interrupts could occur, each generating an interrupt frame on the current supervisor stack. As a result, the system would need to allocate more wired per-process supervisor stack to account for the possible need to store several interrupt frames.

All interrupts vector directly to driver interrupt service routines (ISRs). In general, the system stores the entry points to these driver ISRs in the trap page. When an interrupt occurs, the hardware vectors through the trap page to the appropriate driver ISR.

In the AEGIS system, driver interrupt routines execute quickly and complete either by advancing an eventcount to awaken a waiting process, or by simply returning. An ISR decides whether or not to exit or advance an eventcount by examining the value stored in A0; if this value is zero, the ISR simply exits.

Although all interrupt routines disable interrupts to level 6 at routine entry, it is possible for other interrupts to occur before the ISR gains control and blocks them with the disable. Should subsequent interrupts occur, additional interrupt frames are pushed onto the stack while the system is handling the first interrupt. As a result, interrupt routines cannot simply dispatch upon completion, because any additional interrupt context saved on the current stack will be lost when the dispatcher carries out the stack switch to the new current process. Thus, all interrupt routines jump to routines in the PROC1 manager that check the stack for the existence of additional interrupt frames before calling the dispatcher.

15.6.2.1. Interrupt Eventcount Advance

Interrupt routines that advance an eventcount cannot simply call ec_$advance and dispatch. In such a case, the eventcount advance would reorder the ready list, and a subsequent dispatch would destroy any interrupt context left on the switched-out stack. Consequently, interrupt routines that complete by advancing an eventcount must carry out the following steps:

- Save all registers on the stack (currently, D0-D7, A0-A4).
- Save the address of the eventcount to be advanced.
- Jump to the PROC1 interrupt handling routine proc1_$int_advance.

The advance operation actually advances the eventcount by calling the EC1 internal advance routine. Then, if there are no other interrupt frames on the stack, it jumps to the dispatcher. However, if it finds that subsequent interrupts have occurred, pushing interrupt frames onto the stack, it restores the current interrupt's registers from the stack and jumps to the fault interceptor manager's exit routine (fim_$exit) to return from the interrupt. Although the interrupt routine being handled may be a higher priority process than the current process, it is not made current until the system has a chance to get all interrupt information from the current process's stack. When the last interrupt frame has been removed from the current process's stack, the proc1_$int_advance routine can call the dispatcher.

15.6.2.2. Interrupt Exit

Interrupt routines that simply exit jump to another PROC1 routine (proc1_$int_exit). This routine carries out the check for nested interrupts; if the interrupt that is completing is returning to another interrupt, dispatching cannot occur, so the routine restores registers and calls fim_$exit. Once there are no more interrupt frames, it determines whether or not a dispatch is necessary; if so, it jumps to the dispatcher. The proc1_$int_advance on the other hand, assumes a dispatch is needed because the ISR has advanced an eventcount.

Note also that, unlike ISR calls to proc1_$int_advance, interrupt routines that jump to proc1_$int_exit do not need to save registers on the stack. In this case, the ISR saves only the interrupt's fault frame: its PC and SR. The proc1_$int_exit routine will save registers itself if a dispatch is necessary.

15.6.3. Process Scheduling

Compute-bound processes give up the machine in two ways:

- Voluntarily, by waiting on an eventcount
- Involuntarily, because an interrupt occurs

The scheduling mechanism exists to remove compute-bound processes from the processor when they will not remove themselves voluntarily, and when no interrupts are occuring. The AEGIS scheduling mechanism is implememented as a collection of PROC1 operations that move processes around in the ready list based on time slice, priority, and resource locks held.

These PROC1 operations are separated into the the scheduler -- the modules that adjust a process's scheduling priority -- and the ready list routines, which insert PCBs onto the list (using pointers) according to the priority set by the scheduler.

In general, the scheduler calculates a process's scheduling priority based on the following maxim -- the longer the process waits, the higher its priority; if it exhausts its time slice, its priority is lowered. The following sections elaborate on this maxim.

15.6.3.1. Priority and Time Slice End

The scheduler decrements a process's priority by one at each each time slice end as follows. When a process's time slice end occurs, the timer chip generates an interrupt. The time manager (TIME) handles the interrupt and then calls the PROC1 manager's time slice end handler (proc1_$time_slice_end). This routine:

1. Calculates a new time slice for the process

2. Removes the process from the ready list

3. Decrements its priority by one

4. Restores the process to the ready list in the appropriate position

Reaching time slice end may cause the process to lose control of the processor, as its priority may no longer be high enough to keep it current.

Decrementing priority at time-slice end causes two CPU-bound processes of equal priority to sink to the bottom of the ready list within a minute or so, where they contend for priority 1. Thus, priority 1 implements round-robin scheduling.

15.6.3.2. Priority and Eventcount Waits

The scheduler accounts for time lost to eventcount waits, and increments process priority by one for each 1/4 second of wait time when a process finishes its eventcount wait. The level 1 eventcount routine ec_$advance actually calculates the amount of time the process has waited.

Consequently, an I/O bound process gets a priority boost if it waits long enough for a disk or network I/O.

15.6.3.3. Priority and Resource Locks

Resource locks are also used to determine a process's scheduling status, as follows:

- A process that holds a resource lock has priority over a process that holds no locks.

- A process with a higher resource lock has priority over a process with a lower resource lock.

- A process that holds a resource lock cannot be suspended.

If the process is holding a resource lock and its time slice expires, the scheduler moves it to the end of its priority class when the process releases the last of the resource locks it holds; this means that the scheduler does not reorder the ready list until resource locks are released.

15.6.3.4. Maintaining the Ready List

Processes are either on the ready list or not. Processes are taken off the ready list in three cases:

- When the process initiates a wait for an eventcount (via ec_$wait)
- When the process is unbound (proc1_$unbind) as a result of a call to the level 2 process manager delete function (proc2_$delete)
- When the process is suspended (proc1_$suspend)

Processes are added to the ready list as the result of an eventcount advance (ec_$advance), a resume (proc1_$resume), or a bind (proc1_$bind) operation. A process does not need to be current to be placed on the ready list, nor does it need to be current to be taken off the ready list.

The PROC1 bind/unbind suspend/resume operations call the ready list routines to remove and restore processes from the ready list; the level 1 eventcount manager (ec_$advance, ec_$wait) calls these routines as well to take waiting processes off the ready list and restore awakened waiters to the list.

The ready list routines must be called with interrupts disabled. They perform the following tasks:

- Remove a PCB from the ready list.
- Add a PCB to the ready list as determined by priority and resource locks held.
- Reorder the ready list after an event occurs that alters the scheduling priority of the current process.

The add and remove routines assume that their callers will call the dispatcher. The reorder ready routine, however, checks to see if the first process on the list is the same as the current process after it reorders the ready list; this operation avoids unnecessary calls to the dispatcher. For example, if the current process's scheduling priority has changed because it has obtained a higher priority lock, it will still be the first process on the list, because the higher lock only makes the process more current. In this case, a call to the dispatcher is not needed, since it will simply do the same check for ready equals current and return.

Except for the reorder ready routine, the ready list routines add a process to the end of its priority class, placing the PCB after all the PCBs of equal priority. Adding processes to priority class end guarantees round-robin cycling/scheduling for priority 1 processes.

Chapter 16
Level 2 Process Management

This chapter describes level 2 process context and PROC2 operations. Level 2 process context consists of:

- The UID of the process's stack object
- Whether or not the process is an orphan
- Whether or not the process is a server
- Whether or not the process is the result of a fork operation
- Process ID information
- Process group information

Level 2 process operations include:

- Creating and deleting level 2 processes
- Forking level 2 processes
- Maintaining the level 2 database
- Participating in asynchronous fault delivery

16.1. Level 2 Process Context

Level 2 process context is stored in the PROC2 manager's database. This database contains a table of level 2 context for each level 2 process (p2_$info_t). Figure 16-1 illustrates the layout of this database.

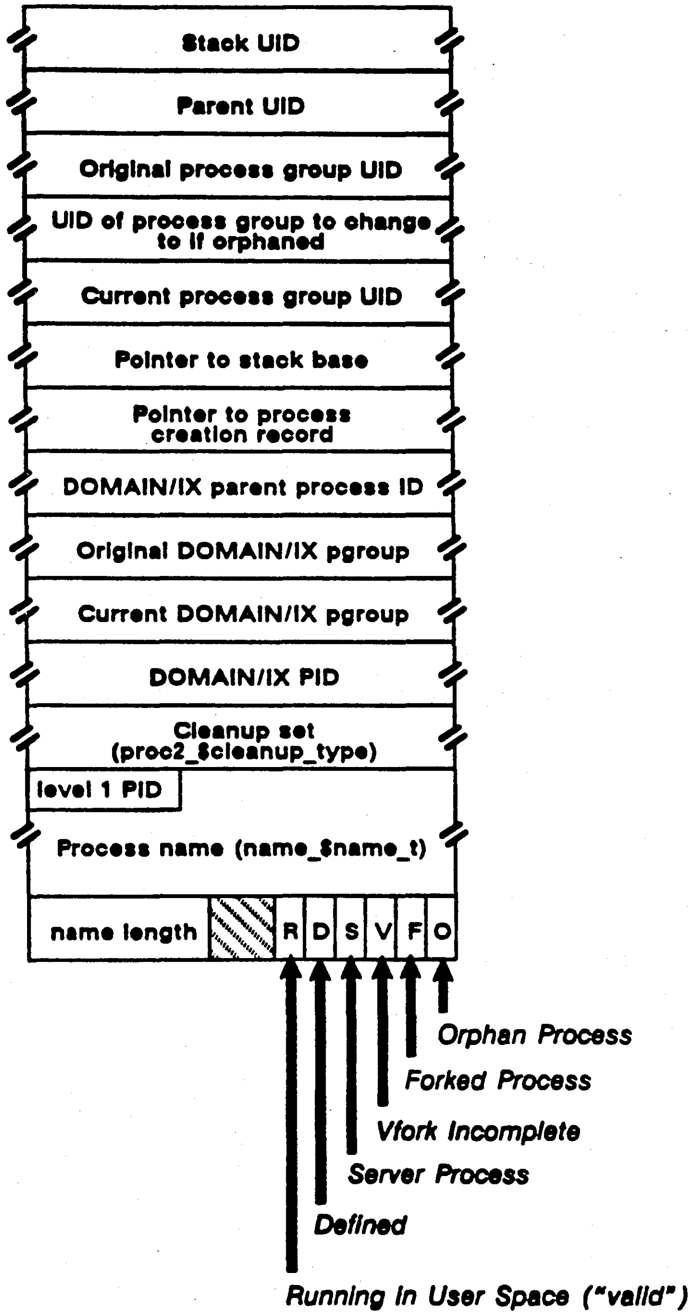

Stack UID
Parent UID
Original process group UID
UID of process group to change to if orphaned
Current process group UID
Pointer to stack base
Pointer to process creation record
DOMAIN/IX parent process ID
Original DOMAIN/IX pgroup
Current DOMAIN/IX pgroup
DOMAIN/IX PID
Cleanup set (proc2_$cleanup_type)
level 1 PID
Process name (name_$name_t)
name length
R
D
S
V
F
O
Orphan Process
Forked Process
Vfork Incomplete
Server Process
Defined
Running In User Space ("valid")

Figure 16-1. Level 2 Process Context Table

16.1.1. The Stack Object

Level 2 process stacks, also called **user stacks**, reside in a portion of a level 2 process's private virtual address space. Because user stacks are pageable, they require backing storage; the PROC2 manager provides this storage through the **stack object**.

When a process is created, a stack object is mapped into its process virtual address space. The stack object contains the process creation record, read/write storage, and the procedure call stack.

16.1.1.1. The Process Creation Record

The process's **creation record** resides in the first segment of the stack object and is mapped to the same process virtual address in every process. It contains all the information that the user-mode process manager (PM) needs to carry out its operations and contains the information passed by the process to any child processes it creates. For example, the process creation record identifies the streams and arguments passed to the process and any programs to be invoked.

16.1.1.2. Read/Write Storage

The process manager and the read/write storage manager (RWS) manage read/write storage as a **heap** rather than as a stack. Heap management divides the storage into free areas and areas in use. When called upon to allocate some storage, the managers in charge of maintaining the heap search through a list of free space for a portion that is large enough to accomodate the request, then hand out the right portion to the requestor. A heap permits storage to be allocated and freed in any order. In contrast, a stack allocates and frees storage on a last-in/first-out basis.

The RWS manager does, however, contain an allocation procedure (rws_$alloc) that manages the read/write storage as a stack. On a program-level exit, the RWS manager cuts back read/write storage to the amount the program had when it entered the program level.

16.1.1.3. The Procedure Call Stack

The procedure call stack contains the stack frames created at each procedure call; this stack consists of eight segments. Two guard segments reside at each end of the procedure call stack; A **guard segment** is a virtual segment that generates a guard fault (a kind of access violation) when a process accesses, or **touches** it, via memory management routines. These guard segments protect the process creation record and the rest of user private address space from destruction by a procedure call stack overflow or underflow.

16.1.2. Orphan Status

When a process creates or forks a new process, the new process is called the **child process**, and the spawning process becomes the **parent process**. An **orphan process** does not have a parent process. Orphan status is significant during cleanup operations during process deletion; while parent processes involve themselves in resource cleanup when any of their children are deleted, an orphan process cleans up its own resources. Section 16.2.3 provides more detail.

A process is usually orphaned (through proc2_$make_orphan) so that it can run as a background process; because an orphan process handles its own cleanup operations on deletion, it can complete without interfering with the process that spawned it.

16.1.3. Server Status

Server processes are identical to normal level 2 processes except that the system does not delete them at logout. The PROC2 operation proc2_$make_server assigns server process status to a given process by setting the server bit in the process's level 2 database. The display manager checks this bit at logout; if the bit is clear, it generates a stop fault for the process. If the bit is set, the process continues to run.

16.1.4. Process ID Information

The AEGIS system supports three types of process ID: the level 1 process ID (PID), the level 2 process UID, and the PID used in the DOMAIN/IX environment. Level 2 process context includes all of these styles of process identification.

The AEGIS PID information stored in the PROC2 database includes the PID of the level 1 process bound to the level 2 process, and the process UID of the process's parent, if one exists.

The PROC2 database also stores process IDs required by the DOMAIN/IX environment. When it creates a process, the PROC2 manager assigns it a DOMAIN/IX process ID. This PID cycles from 2 to 30,000; the PROC2 manager takes the next available number in the range, and simply restarts numbering from 2 when the range is exhausted. (This method of PID allocation exists for compatibility with other UNIX systems.) Consequently, the PROC2 database stores the process's DOMAIN/IX PID and that of its parent, if one exists.

16.1.5. Process Group Information

The AEGIS system also supports the concept of the process group, which the DOMAIN/IX environment uses primarily in fault delivery to send a fault to all processes within a given group, rather than to a single process. Normally, processes in a group are ancestrally related; that is, a group usually consists of the parent process, its children, its childrens' children, and so on. For example, each time a new DOMAIN/IX shell is created, a new process group associated with that process is also created. The DOMAIN/IX environment also allows processes to move in and out of different process groups; the C-shell job control facility is based on this feature. While the AEGIS system identifies a process group with a UID, the DOMAIN/IX environment identifies it by an integer value.

The process group information stored in the PROC2 database includes:

- The UID and DOMAIN/IX number of the process group to which the process originally belonged

- The UID and DOMAIN/IX number of the process group to which the process currently belongs

16.2. PROC2 Operations

The level 2 process manager carries out the following tasks:

- Creates, forks, and deletes user processes

- Allocates and frees user stack files

- Maintains level 2 process context

- Manages level 2 process name space (UIDs)

- Participates in asynchronous fault delivery

- Suspends and resumes processes (via the level 1 process manager)

16.2.1. Process Creation

To create a level 2 process, the PROC2 create operation:

1. Takes a stack UID, a starting PC, and the orphan status, and gives back the UID of the newly created process

2. Allocates a separate virtual address space (from the MST manager routine mst_$alloc_asid) and maps the user stack into this address space

3. Calls the PROC1 manager to allocate a level 1 process and supervisor stack for the level 2 process

4. Binds the level 1 and level 2 processes together and starts the process running in user mode at the specified PC

5. Assigns the process to the process group of the parent

When the creation is complete, the new process's address space maps the stack object whose UID was passed in the call; the initial stack pointer is set to the base of the procedure call stack.

16.2.2. Process Forking

Forking a process (proc2_$fork) proceeds like process creation except for the resulting address space owned by the new process. In particular, the fork operation does the following:

- Copies the parent's stack object to the forked process's address space, including all pages that the parent has touched. Thus, the forked process's stack is a replica of the original.

- Duplicates in the forked process lock database all of the file locks held by the parent process. The forked process holds all of the parent's locks, even if the lock status specifies no cowriters. As a result, forking overrides the lock mechanism.

- Maps all of the objects that the parent has mapped into its address space into the forked process's address space. Consequently, the parent and forked child have identical MST entries.

The PROC2 manager then binds the forked process to a separate level 1 process; thus, there are never more level 2 processes than level 1 processes. When the fork is complete, the parent and forked process share the objects mapped to their respective address spaces during the fork; a modification to one of those objects can be seen by both parent and child. The exception to this sharing is the forked process's stack object, which the fork operation has copied. Figure 16-2 illustrates the relationship between parent and forked child processes.

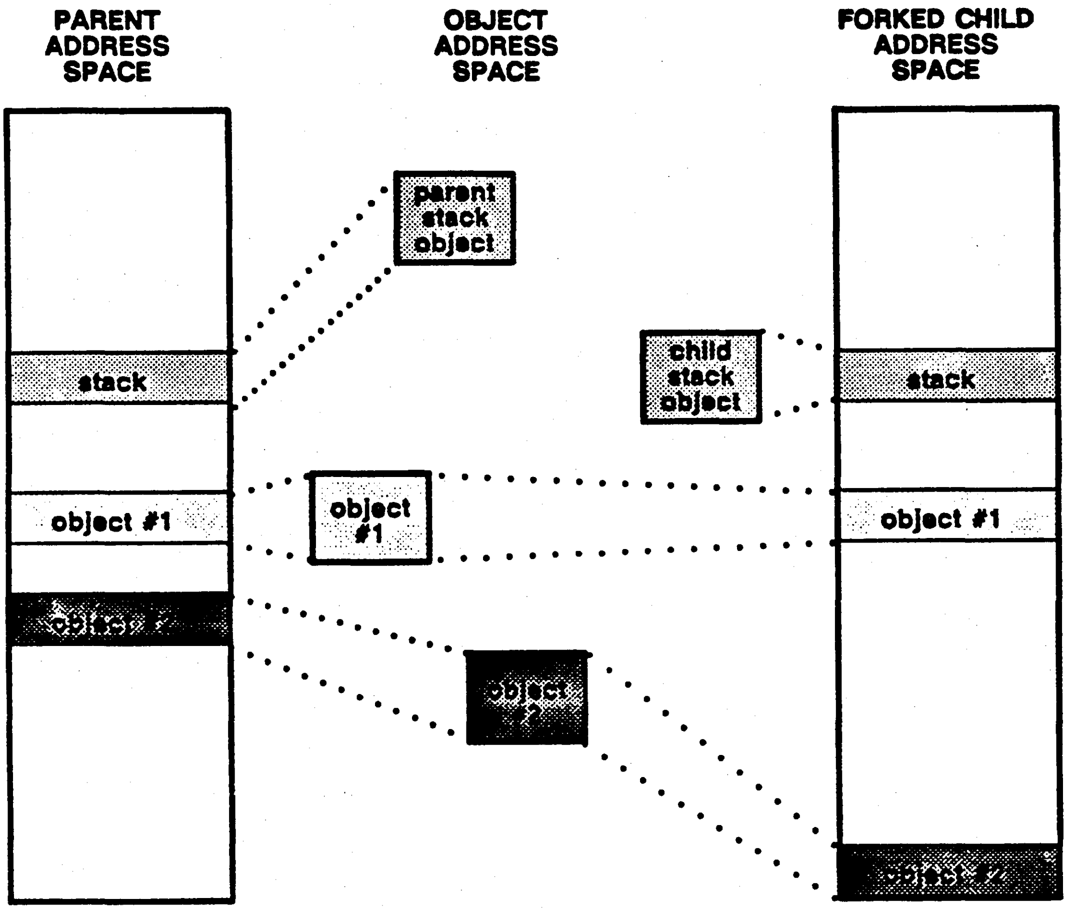


Figure 16-2. Mapping Between A Forked Process and Its Parent

Because the fork operation exists for DOMAIN/IX, and because DOMAIN/IX has no concept of orphan processes, forked processes seldom become orphan processes.

16.2.3. Process Deletion

The PROC2 delete operation (proc2_$delete) deletes the level two process that calls it by:

1. Releasing its associated resources and freeing its ASID

2. Uncataloguing it from the naming server directory database if it is a named process

3. Turning any of its children into orphans by setting the orphan status bit in their level 2 databases

4. Notifying its parent (if one exists) of its deletion

5. Freeing its stack object if necessary

6. Unbinding the level 1 process underneath it by calling the PROC1 manager

The delete operation cannot be called on another process because the MST procedure that frees an ASID cannot clear the memory management unit of another process's virtual-to-physical address associations.

16.2.3.1. Releasing Per-Process Resources

When deleting a process, the PROC2 manager first frees any resources owned by the dying process, including mapped objects, locks, sockets, and ACLs. To free these resources, the delete routine calls the proc2_$cleanup_proc routine. This procedure (proc2_$cleanup_proc) calls all the AEGIS kernel managers that keep per-process information. Each manager cleans up its private database of the per-process state. Resource cleanup occurs before the operation frees the stack object to ensure that any wired stack pages have been unwired when the stack is freed.

Unfortunately, the cleanup procedure has poor locality; it makes many procedure calls which touch many pages, making the delete operation very expensive in terms of system performance.

16.2.3.2. Notifying the Parent Process

When a process creates or forks a new process, it keeps a copy of the child process's creation record mapped in its address space. Within the creation record is an eventcount that the PROC2 delete operation advances to notify the parent if the child process is deleted. The creation record also contains a field that is used to record the amount of CPU time the child process has consumed. Upon deletion of the child process, proc2_$delete operation:

- Advances the creation record eventcount

- Writes the child's accumulated CPU time into its creation record

- Sends a trace fault to notify the parent process of its child's deletion (proc2_$trace_fault_enq)

16.2.3.3. Freeing the Stack Object

When proc2_$delete deletes a child process, it directs stack object cleanup to the parent process. When the parent process receives notification of a child's deletion, it frees the stack object as follows:

- Truncates it back to the creation record (the stack's first segment)
- Collects the information stored in the child's creation record and unmaps it
- Returns the stack object to the pool of free stack objects that the PROC2 manager maintains

Orphan processes clean up their own stacks when they are deleted (by calling proc2_$free_stack_file).

16.2.4. Stack Object Allocation

The PROC2 manager maintains a pool of stack objects that the first level 2 process creates during system initialization. When the system is up and running, the PROC2 manager keeps a maximum of eight stack objects in this pool at any given time. The system uses this pre-existing pool of stack files to avoid the overhead incurred from calling the FILE manager (file_$create/delete) to create stack objects dynamically.

The stack pool is an array of stack records (stack_rec_t); each record contains the UID of a stack object, its owner's subject identifier (SID), and an index to the next stack file in the pool. Free stacks are chained onto a free list, while stack files in use are chained to an "in use" list.

At process creation or fork, the PROC2 manager stack allocation procedure (proc2_$alloc_stack_file) allocates a stack file for the new process. The procedure first attempts to locate a stack file in the free pool that has the same subject ID as the newly created process; if no free entries exist in the stack pool, it checks the "inuse" list for the least recently used stack file. Note that the PROC2 manager creates a new stack file by calling the FILE manager (file_$create) ONLY if it cannot locate a free or replaceable stack file.

The PROC2 delete operation frees the stack objects of orphan processes; the procedure proc2_$free_stack_file frees the stack object associated with the UID specified in the call. This routine truncates (via file_$truncate) the stack object and places it into the free pool for later re-use by another process.

The PROC2 manager also frees stack objects at logout (via proc2_$cleanup_stack_files); it frees the stack objects associated with a specified subject ID.

16.2.5. Maintaining Level 2 Context

The PROC2 manager contains operations that maintain a process's level 2 context by setting and clearing fields in its PROC2 table. These operations include:

- Creating server processes and orphan processes
- Maintaining process group status by setting the process group of a process and converting between DOMAIN/IX PIDs and group PIDs and level 2 process UIDs
- Assigning a name to a process by adding the name to the process's PROC2 table

16.2.6. Maintaining Process Names

The user-mode process manager (PM) allows programs to assign names to processes when they are created. In turn, the PM naming operations call the PROC2 manager, which stores the name in the process's PROC2 database (proc2_$set_name) and catalogues it in the directory 'node_data/proc_dir.

When it deletes a process, the PROC2 manager refers to the process name field in the PROC2 database. As part of its operation, proc2_$delete must remove the process name from the file system directory proc_dir. Because the process's name is stored in the PROC2 database, the delete operation can simply drop the name from the target proc_dir directory. If the name were not stored, the delete operation would proceed more slowly; in this case, proc2_$delete would need to open the directory and then ask the naming server to find the name/UID pair in its directory data structures.

16.2.7. Suspend/Resume Operations

The PROC2 manager suspend and resume operations call the PROC1 manager to carry out process suspension. Unlike level 1 suspension, Level 2 process suspension waits for the suspension to happen (instead of setting a suspend pending bit).

Chapter 17
Eventcounts and Mutual Exclusion

The EC and EC2 managers provide the level 1 and 2 eventcount operations. In addition, the ML manager exists within the AEGIS kernel to provide mutual exclusion to level 1 process resources. For performance reasons, the EC and ML sources exist within the PROC1 manager source code. This chapter describes level 1 and 2 eventcount data structures and operations and provides information about mutual exclusion at the kernel level.

17.1. Level 1 Eventcounts

The sole method of synchronization in the AEGIS system is through the level 1 eventcount, which is the AEGIS system blocking primitive that causes a process to relinquish control of the processor. The only way a process can block itself is by using an eventcount.

Processes create eventcounts by allocating wired storage from supervisor global address space and initializing it as an eventcount data structure. A level 1 eventcount is composed of a wait value and links to the list of processes waiting on that eventcount; Figure 17-1 illustrates its structure.

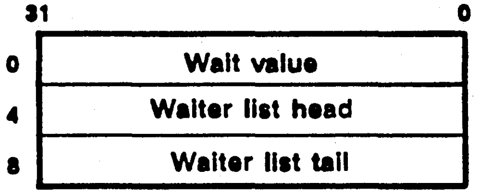


Figure 17-1. Level One Eventcount

Processes can perform the following operations on level 1 eventcounts:

- Initialize an eventcount (ec_$init)
- Wait for one to three eventcounts to reach one to three trigger values (ec_$wait)
- Wait for a list of eventcounts to reach a list of trigger values (ec_$waitn)
- Read an eventcount (ec_$read)
- Advance an eventcount (ec_$advance)

17.1.1. Waiting on a Level 1 Eventcount

When a process calls the EC manager to wait on an eventcount, the EC manager does the following:

- Places a waiter entry on the on the process's stack
- Takes the process off the ready list
- Calls the dispatcher

A waiter entry contains the following fields:

- The value for which the process is waiting
- A pointer to the process's PCB
- Pointers to other processes waitinq on the same eventcount

When the EC manager completes the wait operation, the process's stack will contain the number of waiter entries that correspond to the number of eventcounts on which the process is waiting, and a dispatch frame. Should another process subsequently wait on the same eventcount, the EC manager links the two waiter entries together. Figure 17-2 illustrates the linked list level 1 eventcount data structures when several processes are waiting on a number of eventcounts.

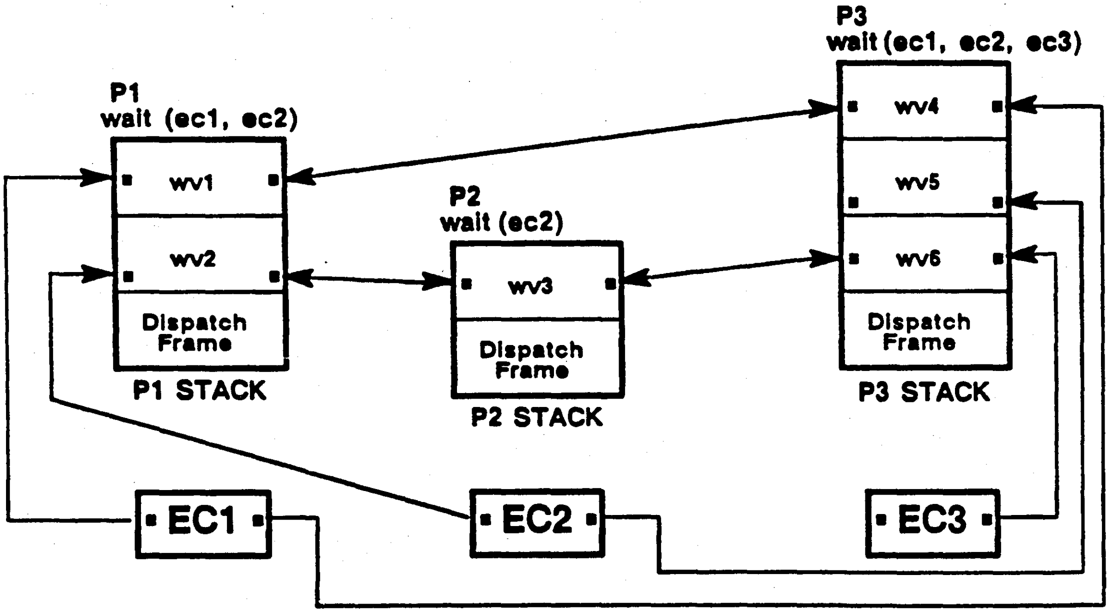


Figure 17-2. Processes Waiting on Level 1 Eventcounts

17.1.2. Advancing a Level 1 Eventcount

When a process calls the EC manager to advance an eventcount, the EC manager advances the eventcount, then chains through all the processes waiting on that eventcount to determine whether they are waiting for the value that the eventcount has reached and are thus ready to run. The EC manager then awakens any processes waiting for that value by threading them onto the ready list. The processes, in turn, call the EC manager to unlink their waiter entries from the waiters list.

If a process is waiting on multiple eventcounts, it will become ready to run when any one of those eventcounts reaches the process's specified wait value. However, a suspended process awakened by an eventcount advance will not be placed back on the ready list until it is explicitly resumed.

17.2. Level 2 Eventcounts

User-mode processes use level 2 eventcounts to synchronize their operations. A level 2 eventcount is similar to a level 1 eventcount: it consists of a 32-bit value plus a 6-bit index to the head of a list of processes waiting on the eventcount, as shown in Figure 17-3.

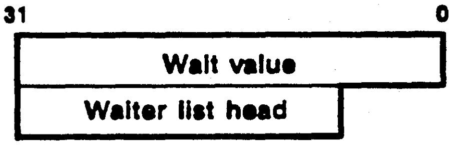


Figure 17-3. Level Two Eventcount

While the structure of a level 2 eventcount is similar to that of a level 1 eventcount, the level 2 eventcount is not wired and exists in the per-process address space of the process that created it, rather than in global address space.

The level 2 eventcount manager contains operations to:

- Create a level 2 eventcount (ec2_$init, ec2_$init_s, ec2_$register_ec1)

- Wait on an eventcount (ec2_$wait)

- Advance an eventcount (ec2_$advance)

Some of these operations are exported to user space; see the manual *Programming with System Calls for IPC* for more information.

17.2.1. Creating a Level 2 Eventcount

A process creates a level 2 eventcount by reserving space for it in its address space, then calling ec2_$init to initialize it as an eventcount. Because an eventcount is a shared object, any processes that map the eventcount can concurrently observe it. However, because the level 2 eventcount mechanism is based on shared memory, processes on different nodes cannot wait on the same eventcount.

Level 2 processes can also wait on level 1 eventcounts that certain AEGIS managers export to user space. In order to protect against level 2 processes passing in bad pointers to these system-level eventcounts, managers that export their eventcounts first **register** them with the EC2 manager. Registration proceeds as follows:

1. The level 2 process calls the manager to obtain a pointer to its exported eventcount.
2. The user-space manager passes a pointer to its eventcount to the EC2 manager.
3. The EC2 manager stores this pointer in its database of registered eventcounts and passes an index into this table, or *ticket*, back to the user-space manager.
4. The user-space manager passes the ticket to the level 2 process as a return argument.

When the level 2 process decides to wait on the level 1 eventcount, it calls ec2_$wait and specifies the ticket that identifies the eventcount. The EC2 manager uses the ticket to index into its array, and then sets up the wait on the corresponding eventcount. Figure 17-4 illustrates this sequence.

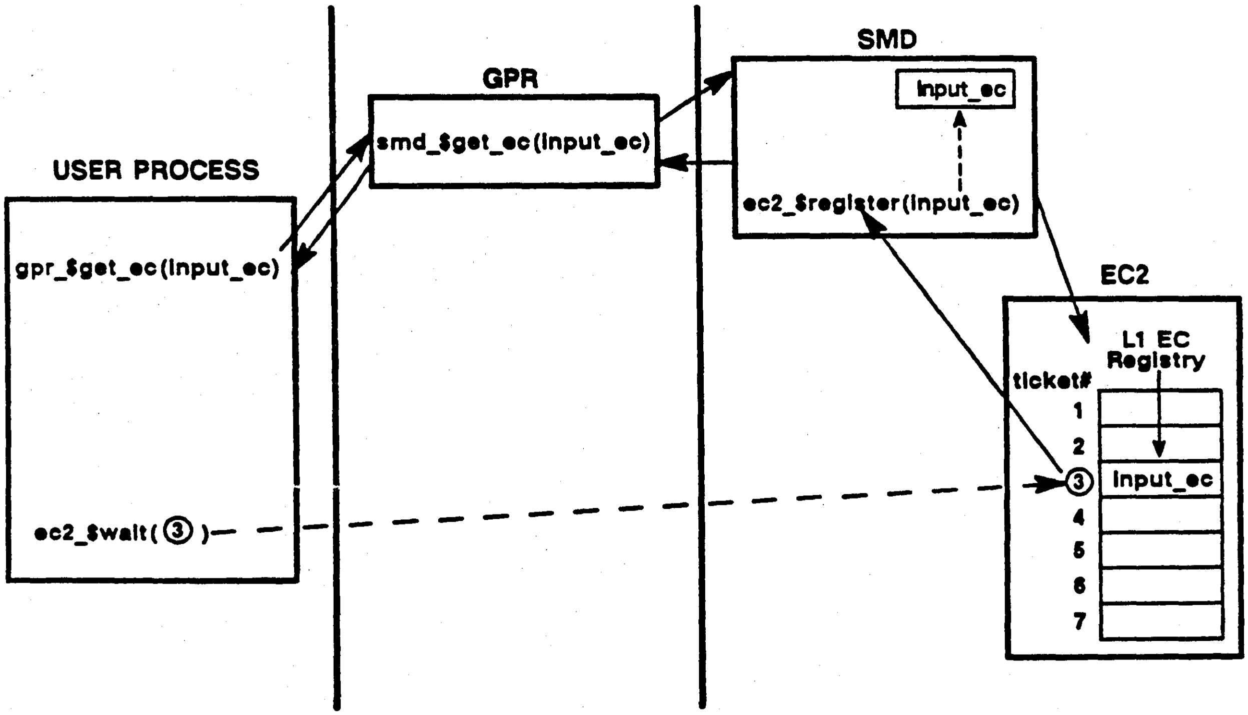


Figure 17-4. Registering a Level 1 Eventcount

In Figure 17-4, a level 2 process gains access to the user-space graphics primitives (GPR) eventcount by calling the GPR manager (gpr_$get_ec). The GPR manager in turn calls the kernel-space screen manager driver (SMD)(smd_$get_ec) to get a pointer and pass it to the EC2 manager for registration. The EC2 manager assigns a ticket to the level 1 eventcount and returns to the SMD manager, which returns the ticket value to the GPR routine, which returns the ticket to the calling level 2 process.

When the process wants to wait on this registered level 1 eventcount, it passes the ticket back to the EC2 wait routine (ec2_$wait). The wait routine reads its table of registered level 1 eventcounts to find the eventcount that corresponds to the ticket value passed to it.

17.2.2. Waiting on Eventcounts

Processes call EC2 wait operations to wait on:

- One or more level 2 eventcounts

- One or more level 1 eventcounts exported by AEGIS managers

To wait, the process passes a wait value and a pointer to the target eventcount. The EC2 manager carries out the following operations for a level 2 eventcount:

- Allocates a waiter entry from its global storage and chains it to its level 2 eventcount database

- Copies to the waiter entry the process's trigger wait value, its process ID, and a link to the next process waiting on that eventcount.

- Passes back a pointer that identifies the eventcount to the waiting process

If the process asks to wait on a registered level 1 eventcount, the EC2 manager passes the eventcount itself to the ec_$waitn call.

The EC2 eventcount service is layered on top of EC1 so that all level 2 wait calls that a process makes actually wait on one per-process level 1 eventcount. The process waits on this one level 1 eventcount regardless of how many level 2 eventcounts it waits on. The layering of multiple EC2 eventcounts on a single EC1 eventcount works because the EC2 manager awakens a process when *any one* of its trigger values is satisifed. The only time a process waits on more than one level 1 eventcount is when it waits on a registered level 1 eventcount.

Consequently, when the EC2 wait call completes, the process is waiting on:

- All registered level 1 eventcounts

- The per-process level one eventcount that represents all the level 2 eventcounts for which the process is waiting

- A quit eventcount, because the process must exit from the kernel if it gets a quit fault. Consequently, all processes that plan to wait on an eventcount for a substantial amount of time wait on a quit eventcount as well; see Chapter 18 for more information.

17.2.3. Advancing an Eventcount

A process that wants to advance an eventcount passes a pointer to the eventcount (which it obtains via shared memory) to the EC2 manager's eventcount advance operation. Figure 17-5 illustrates the data structures involved.

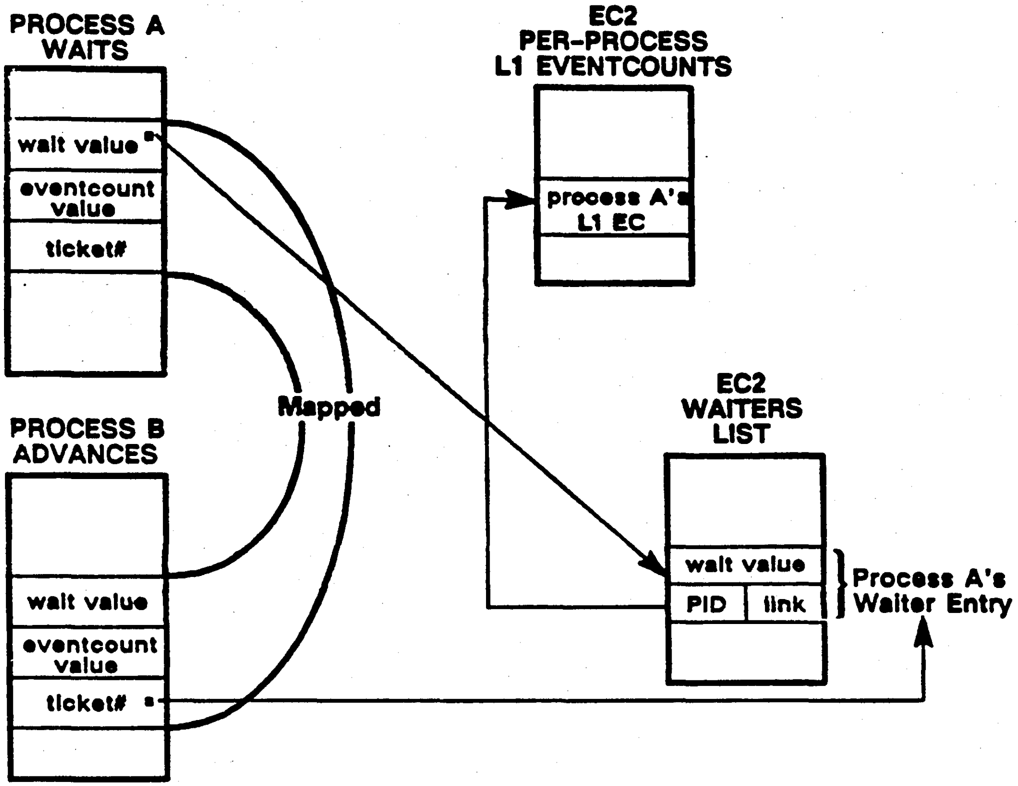


Figure 17-5. EC2 Wait and Advance Operations

The EC2 eventcount advance operation is implemented in two parts: the eventcount advance runs in user mode, while the operation that awakens waiting processes (ec2_$wakeup) runs in supervisor mode. The EC2 advance operation is split into user mode and supervisor mode to take advantage of the user-level mutual exclusion facility and to avoid the expense of unnecessary calls to supervisor mode.

When it is called to advance an eventcount, ec2_$advance increments the eventcount value by one, and then determines whether there are any processes waiting on the eventcount by checking to see if the eventcount has anything in the waiters list head field. If it does, the routine calls the supervisor wake operation (ec2_$wakeup), which does the following:

- Locates the waiter nodes of all processes that are waiting on the advanced eventcount

- Compares each process's trigger wait value with the current eventcount value.

- Calls ec_$advance on any processes whose trigger values match the current eventcount value; ec_$advance increments the per-process level 1 eventcounts, which awakens the processes.

17.3. Mutual Exclusion on Resource Locks

The mutex lock (ML) module is the kernel-level lock manager; it guarantees first-in/first-out request and grant of process requests for level 1 resource locks. The ML manager consists of a sequencer that orders multiple requests for a single resource lock by issuing "tickets" to the requesting processes; the sequencer increments each ticket value by one, and never gives the same ticket value to two different processes.

Consequently, if five processes want the disk lock, the ML sequencer orders, via ticket values, which process gets the lock first. Once the resource is released, it will go to the next process in the sequence. Currently, the sequence is ordered by the order in which the processes tried for the lock.

Although a process that calls ML to obtain mutual exclusion on a resource lock will be the only process authorized to proceed once ML returns, it will not be the only process that has the requested lock bit set in its PCB.

ML calls the PROC1 manager's set and clear lock operations to get and release the locks. The proc1_$set_lock routine attempts to set the lock bit, then returns to the ML manager, which issues a ticket for the lock to the requesting process.

Chapter 18
Fault Handling in the AEGIS Kernel

The AEGIS system recognizes two kinds of faults: synchronous faults and asynchronous faults. **Asynchronous faults** are software-defined fault conditions that occur independently of program execution. **Synchronous faults** are processor-defined fault conditions. Unlike asynchronous faults, they occur as a direct result of program execution; the set of synchronous fault conditions depends on the processor architecture.

18.1. Processor Fault Handling

When a fault occurs, the M680x0 processor performs the following actions:

1. Saves the current status register (SR) and program counter (PC).

2. Enters supervisor state.

3. Generates an exception vector number that corresponds to the fault; for example, an illegal instruction generates the vector number 010.

4. Creates a fault frame that includes the faulting SR and PC and pushes it onto the supervisor stack. The contents of the fault frame varies according to the kind of fault that occurred; the fault frame format differs depending on the processor in use. See the appropriate processor manual for more information.

5. Resumes execution at the location pointed to by the appropriate exception vector, which points to the entry point of an AEGIS fault handler.

18.2. AEGIS Fault Handling

The AEGIS system provides a fault handler for each fault condition; these handlers, along with some common fault handling routines, are grouped into the fault interceptor module (FIM). The fault interceptor module is actually two modules: fim_wired and fim_unwired. The fim_wired module contains those handlers that must be able to run without taking a page fault; in particular, the page fault handler itself. The fim_unwired module contains all the other handlers, such as the fault handlers for illegal instruction, zero_divide, and address errors.

In general, all of the AEGIS fault handlers follow this sequence of operations when invoked to handle a fault:

1. Determine whether the fault occurred in user mode or supervisor mode.

   - If the fault occurred in supervisor mode, they next determine whether or not the fault was "legitimate". If not, they call the routine fault_$crash to display the fault information.

   - If the fault occurred in user mode, they push fault status and registers on the stack.

2. Build a diagnostic fault frame that describes the fault.

3. Reflect the fault to user space.

This sequence represents the common fault handling followed by the majority of all fault handlers. The exceptions to this sequence are faults that involve virtual memory management (region, segment, and page faults). The AEGIS kernel normally handles memory management faults exclusively; as a result, memory management faults are generally invisible to user-mode programs. The fault handling sequence described above is invoked only if the manager involved in satisfying the fault, for example, mst_$touch, reports an error.

18.2.1. Determining Where the Fault Occurred

Fault handlers call the routine fim_chk_com to determine where the fault took place. The fim_chk_com routine determines whether the fault occurred in user mode or in supervisor mode by examining the supervisor-mode bit in the status register that the processor saved in the fault frame.

18.2.1.1. Handling Supervisor-Mode Faults

If the fault occurred in supervisor mode, fim_chk_com calls the fault_$crash routine, which displays the fault frame and exits through the crash_system routine. For example:

```
FAULT IN AEGIS:
03D77B88: SR:2008 PC:3D12CCC FF:B008 (B) FA:3FFFFFF SW:0135

CRASH_STATUS 00040004 ECB 00000000 PID 0002
S ...
```

The routine displays the fault address (FA) and special status word (SW) only on bus/address errors. (The frame format word (FF) is followed by the letter error identification from the PROM.) All registers except the stack pointer (SP) remain as they were at when the fault occurred. The G,G *+2 command sequence will return control to the point of the fault using any registers that the mnemonic debugger (MD) has reloaded. If you want to change the return SR or PC or otherwise modify the fault frame, you must patch the actual frame on the stack (using the address displayed above).

18.2.1.2. Handling User-Mode Faults

If the fault occurred in user mode, fim_chk_com pushes the following information onto the stack:

- The fault status code, for example, fault_$uii. The fim_chk_com routine generally obtains the fault status code from the processor. However, the fim_$soft_fault (TRAP E) passes fault status in from user space; on trace faults, fim_chk_com obtains the actual fault status from the fim_$trace_sts[asid] array.

- A word that indicates the kind of fault that occurred. A value of zero indicates that the fault was not the result of a bus or address error; a nonzero value indicates one of the bus or address errors (address, parity, bus).

- Processor address registers A0-A6, data registers D0-D7

At this point, the fault handlers all join the common fault handling path (fim_$com), which builds the diagnostic frame and reflects the fault to user space.

18.2.2. Handling Privileged Instruction Violations

The handler for privileged instruction violations carries out an additional operation to the normal fault handling sequence -- it checks for a Move from SR instruction. This instruction was not privileged on the 68000, but became privileged on the 68010 and 68020. If the handler finds that the instruction that incurred the fault was a Move from SR, it ignores the fault (the instruction is No-oped).

18.2.3. Handling MMU-Related Errors

Unlike most of the other fault handlers, which join the common fault handling path at fim_$com, address, parity, and bus errors (the latter being reported by fim_wired) join at fim_$abcom. If the fault occurred in user mode, fim_$abcom takes no special actions; it simply transfers control to the common fault handling path. Supervisor-mode address and bus errors, on the other hand, require special handling, because the fault may have been caused by a reference to a user-mode argument that was incorrectly specified.

If a supervisor-mode error has occurred, the fim_$abcom routine first checks to see if the faulting process holds any mutex locks. Two locks -- ec2_$lock and pbu_$lock -- are special-cased: if either is held, the routine calls the kernel-level ML manager (ml_$unlock) to release the lock.

Next, the routine obtains the faulting address from the fault frame. If the faulting address is above the supervisor global boundary; that is, the supervisor-mode fault occurred while trying to reference supervisor data, the routine then calls the fault_$crash routine.

18.2.4. Common Fault Handling

All of the fault handlers join at the common fault handling routine fim_$com. (If you are having trouble catching a fault or determining why a process is dying, this is a good place to put a breakpoint.) The stack at this point contains the following:

- saved registers
- A flag that indicates the kind of fault; for example, address/bus error, parity, fim_$generate
- The fault status (status_$t)
- The fault frame

Figure 18-1 shows the contents of the stack upon entry into fim_$com.

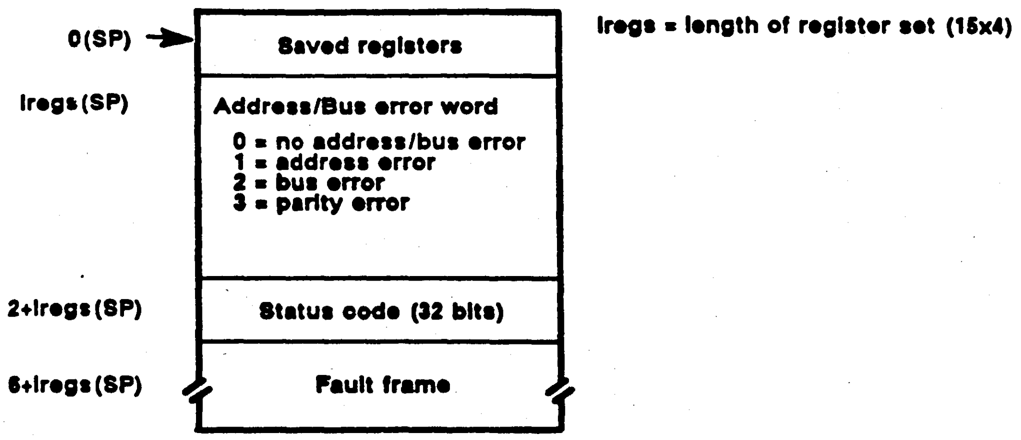


Figure 18-1. Stack at Entry to The Common Fault Handler

The common fault handler carries out the following steps:

1. Checks for faults within GPIO interrupt routines
2. Checks for fault on fault
3. Locates the user fault handler
4. Validates the user stack pointer (USP)
5. Creates a diagnostic frame
6. Dispatches to the user fault handler

18.2.4.1. Checking for GPIO Faults

The fim_$com routine first determines if a user-mode GPIO interrupt handler was in control when the fault occurred; faults in GPIO interrupt handlers must be treated specially because of the non-standard environment in which these interrupt routines run.

When the processor receives an interrupt from a user GPIO device, it saves the ASID of the currently running process and replaces it with the ASID of the process that owns the device. This step is necessary because GPIO interrupt routines are installed in per-process private address space and must have access to their own code and data. However, the system does not perform a full context switch to the process that owns the device, as interrupts should be handled as quickly as possible so that a context switch is not necessary. The interrupt routine is not allowed to generate any sort of a fault (even a sharing fault) while it runs, but if it does, the currently running process should not be disturbed (that is, faulted) since it most likely has no relationship to the process that owns the GPIO device.

Therefore, if a GPIO interrupt routine incurs a fault, fim_$com calls pbu_$get_diag_ptr; this routine returns a pointer that indicates where the fim_$com should build the diagnostic frame. The fim_$com routine uses this pointer as the location at which to build the frame rather than using the user stack pointer (USP), which is where it usually builds the diagnostic fault frame.

18.2.4.2. Checking for Fault on Fault

The common fault handler next checks a per-process flag (fim_$in_fim[asid]) to see if the fault handler has been re-entered while in the process of handling a previous fault. If fim_$com detects a prior fault, it deletes the process (proc2_$delete). The process is deleted while it is running in supervisor mode; it never returns to user-mode operation. As a result, any resources that the process was using are not released. Although the system releases any supervisor mode resources used, such as writing modified pages out to disk, it does not release any user-mode resources, such as open files and transcript pads.

If it is process 1 (DM or SPM) that has incurred a fault on a fault, fim_$com crashes the system.

18.2.4.3. Locating the User Fault Handler

Next, fim_$com determines whether or not the faulting process has installed a user-mode fault handler. The fault interceptor module database contains an array that lists the entry points to each process's user fault handler (fim_$user_fim_addr); the process manager (PM) installs (via fim_$install) a user FIM's address into this list and deletes it (via fim_$free_asid) when the process is deleted. The fim_$com routine attempts to locate the user FIM that corresponds to the faulting process using the process's ASID as the index into the table. If there is no user-mode FIM, fim_$com deletes the process or crashes the system if the process is 1.

18.2.4.4. Validating the User Stack Pointer

The fim_$com routine next validates the user stack pointer (USP), which is the location at which it will build the diagnostic fault frame. The USP must be:

- Below the protection boundary
- Above the bottom of the stack (as_$stack_low)
- On an even-address boundary

If any of these tests fail, fim_$com forceably resets the USP to a "known" valid location a short distance from the bottom of the user stack.

If a GPIO interrupt routine faulted, fim_$com builds the diagnostic frame at the bottom of the stack reserved for the interrupt routine.

18.2.4.5. Building the Diagnostic Fault Frame

Once fim_$com has validated the USP, it decrements its value by the size of the diagnostic fault frame and uses this pointer to build the frame.

The **diagnostic frame** contains information that describes the fault to the user fault handler. Figure 18-2 illustrates the information contained within the diagnostic frame.

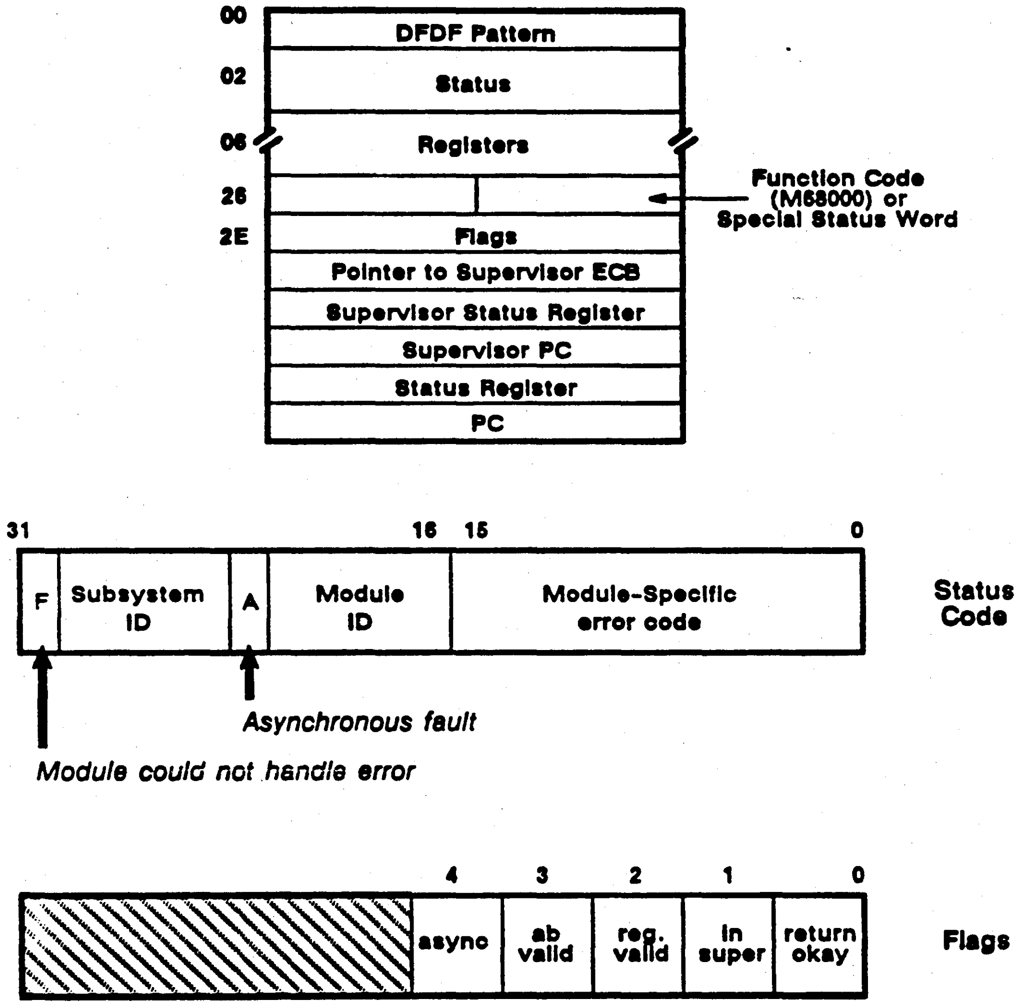


Figure 18-2. Diagnostic Frame

To build the diagnostic frame, the common fault handler:

1. Writes a debugging pattern (DFDF) that flags the frame as a diagnostic frame; this pattern makes it easy to identify diagnostic frames within stack dumps.

2. Writes the status code passed to the common fault handler by its caller; this is the normal status_$t code illustrated in Figure 18-2.

3. Saves the USP and processor registers (A0-A6, D0-D7).

4. Determines whether the fault is asynchronous; if so, it sets the asynchronous bit in the diagnostic frame and clears the asynchronous bit in the status code; asynchronous faults are described in detail in Section 18.4.

5. If the fault was a user-mode bus or address error, it obtains the function code (FC) or the special status word (SSW) and the faulting address from the fault frame.

6. Sets the return-permitted bit in the diagnostic frame if the fault was not a bus/address error.

7. Determines whether the fault occured in user mode or supervisor mode. If the fault occurred in supervisor mode, the handler sets the in supervisor bit in the diagnostic frame and saves (in the diagnostic frame) the supervisor status register, supervisor PC, and the ECB of the supervisor routine in which the fault occurred. If the fault occurred in user mode, the common fault handler saves the user SR and PC.

After building the diagnostic fault frame, fim_$com ensures that the only data left on the supervisor stack is a short (4-word) fault frame. In this way, an exit through the fault frame leaves the SSP pointing exactly to the top of the stack. The fim_$com routine may be required to make two adjustments to the stack. If the fault was a user-mode bus or address error, it pops the extra bus/address information in the long fault frame; if the fault occurred in the supervisor, it unwinds the entire stack; that is, it sets the SP to *os_stack_base[asid]* minus 8.

18.2.4.6. Reflecting the Fault to User Mode

Now fim_$com makes another check to see if the fault occurred in a user-mode GPIO interrupt routine. If so, the fault handler does not pass the fault to the current process; instead, it exits back to the GPIO logic (pbu_$int_fault_com), which:

- Restores the state of the interrupted process

- Passes an asynchronous fault to the process that owns the GPIO device

- Exits and advances the eventcount associated with the device, in case the owning process was waiting for the interrupt

If the fault did not occur in a GPIO interrupt routine, the fault handler clears the fault-in-fault flag and moves the user-mode FIM address from the user FIM array into the fault frame. If the fault occurred in supervisor mode, fim_$com calls cleanup routines for those managers that are susceptible to faults. (Currently, these managers are ACL, EC2 and NAME. They can be viewed as static cleanup handlers for the kernel.)

Finally, fim_$com checks to see if the faulting process holds any mutex or exclusion locks (via procl_$inhibit_check). If this check fails, the system crashes with fault_$while_lock_set status, since there are *no* circumstances in which the AEGIS kernel should exit to user mode with a kernel lock held. If no locks are held, the fault handler exits by jumping to the common exit routine fim_$exit; this routine normally just executes an RTE instruction through the fault frame, thus passing control to the user-mode fault handler. Section 18.4 provides more information about fim_$exit.

18.3. Handling SVC Faults

The SVC trap handlers, which handle traps 0 through 5 (and pbu_asm.asm, which handles trap 6) call a special FIM routine when they detect an illegal argument or an invalid SVC code. This routine, called fim_$generate, uses the common fault handling path described in the previous sections; the only difference is a hardware fault condition does not initiate its actions. For example, if the SVC catcher finds an argument pointer whose value is above the protection boundary, it calls fim_$generate with fault_$prot_violation status. The fim_$generate routine performs the following operations:

1. Stores status on the stack (just above the frame from the trap instruction)

2. Stores a flag that indicates to fim_$com that the fault is the result of a fim_$generate

3. Joins the common fault path at fim_$com.

The only effect that the fim_$generate flag has on fim_$com is that fim_$com does not mark the registers in the diagnostic fault frame as valid, as fim_$generate's caller has presumably destroyed the user-mode registers already.

18.4. Asynchronous Fault Handling

Asynchronous faults are faults that result from actions unrelated to the execution of a process; they can arrive at any time while a process runs and have no relationship to that process's operation. For example, if a process takes a page fault or divide-by-zero, these faults are synchronous because they were the direct result of code execution in the process. A quit fault, on the other hand, is generated by a user (with the help of the display manager process); the quit fault has nothing to do with what a process happens to be doing when the quit is generated.

Asynchronous fault handling in the kernel is split into two related operations: posting and delivery. User processes must call the supervisor to generate asynchronous faults; specifically, they call the PROC2 manager to post the fault. The PROC2 manager, in turn, uses the fault interceptor module to deliver the fault.

18.4.1. Posting an Asynchronous Fault

A process **posts** an asynchronous fault by calling the PROC2 manager's trace fault routine (proc2_$trace_fault) and specifying the target process's UID and the fault code (status_$t) to be sent. User-mode programs frequently post asynchronous faults; a display manager request for a quit fault is the most common instance. AEGIS kernel managers also post asynchronous faults, although they do so less frequently than user-mode programs; SIO line quits and floating point (PEB) faults are examples. Interrupt routines cannot generate asynchronous faults, (nor can cpu-B-eligible code) because the user-mode process's supervisor stack may not be valid and proc2_$trace_fault is unwired.

There are several variations on posting faults with proc2_$trace_fault:

- Calling proc2_$quit; this routine simply calls proc2_$trace_fault with fault_$quit status

- Calling proc2_$trace_fault_enq; this routine operates exactly like proc2_$trace_fault, but if a previous fault has yet to be acknowleged (see below), the fault is queued to a small pending fault queue.

- Calling proc2_$trace_fault_pgroup; this routine operates just like proc2_$trace_fault, but the fault is sent to each process in a process group.

- Calling proc2_$trace_fault_pgroup_enq; this routine operates like proc2_$trace_fault_enq, but the fault is enqueued for any processes within a group that have an outstanding unacknowleged fault.

A process that has received an asynchronous fault must acknowledge the fault before proc2_$trace_fault can accept any further asynchronous faults for posting; it does so by directing its user-mode FIM to call fim_$acknowledge on its behalf.

18.4.2. Structures for Asynchronous Fault Handling

The fault interceptor module (FIM) maintains several data structures that the asynchronous fault and FIM routines use during asynchronous fault handling and delivery; the routines use the target process's ASID as an index into these structures. The asynchronous fault database associated with each process includes:

- A trace bit flag

- An area for trace status code

- A quit inhibit flag

- A quit eventcount and quit value

- A fault delivery eventcount

(Note that the term *quit* or *quit fault* appears in the variable names in the source code. These terms are anachronistic references to the days when the model of asynchronous faults was simpler. When you encounter the term *quit*, read *asynchronous*.)

18.4.2.1. Trace Bit Flag

The per-process trace bit flag (fim_$trace_bit) indicates whether the process should have its trace bit set on its next exit from supervisor mode. The routine fim_$deliver_trace_fault sets the flag; it is cleared by fim_$exit and fim_$clear_trace_fault.

18.4.2.2. Trace Status

The per-process status code (fim_$trace_sts) indicates the status code (status_t) to be delivered to the process when a trace fault occurs; the PROC2 posting routine (proc2_$deliver_trace_fault) puts the status code into this field.

18.4.2.3. Quit Inhibit Flag

The quit inhibit flag (fim_$quit_inh) indicates the state of asynchronous fault handling. A false (00) value indicates that the process can accept an asynchronous fault; a true value (FF) indicates that the process has an unacknowledged asynchronous fault and thus cannot accept another fault. The proc2_$deliver_trace_fault routine sets this flag; The fim_$acknowlege and fim_$free_asid routines clear it.

18.4.2.4. Quit Eventcount

The FIM provides each process with a level one eventcount (fim_$quit_ec) that can be used to awaken the process should an asynchronous fault occur. AEGIS kernel modules that need to be awakened on an asynchronous fault include this eventcount in the ec_$wait call. The proc2_$deliver_trace_fault routine will advance it when an asynchronous fault occurs.

The field fim_$quit_value stores the fim_$quit_ec value for the last acknowledged asynchronous fault. AEGIS kernel modules that wait on the fim_$quit_ec eventcount use [fim_$quit_value + 1] as their wakeup trigger value.

18.4.2.5. Fault Delivery Eventcount

A posting process can also wait on an eventcount for a target process's acknowledgement (fim_$deliv_ec). User-mode processes call the PROC2 manager (proc2_$get_ec) to get a delivery eventcount; the fim_$acknowledge routine will advance it when the user FIM calls it to acknowledge the fault.

18.4.3. Asynchronous Fault Delivery

Asynchronous fault delivery is a three-stage process. The first stage occurs in the process that is delivering the fault and the second two in the process to be faulted (the "target" process).

18.4.3.1. Delivering the Asynchronous Fault

All of the trace fault posting routines use the proc2_$deliver_trace_fault routine to deliver the fault. This routine operates with the PROC2 mutex lock held, thereby avoiding problems when two processes try to post a fault to the same target process at the same time. (It also avoids posting a fault to a target process that deletes itself before the post is complete.)

The proc2_$deliver_trace_fault routine determines whether an asynchronous fault is outstanding for the target process by examining its quit inhibit flag. If the inhibit flag is not set *or* if the status is fault_$blast or fault_$fault_lost, then proc2_$deliver_trace_fault does the following:

1. Saves the status code in the target process's fim_$trace_sts field and sets the asynchronous bit.
2. Sets the inhibit flag.
3. Advances the target process's fim_$quit_ec.
4. Calls fim_$deliver_trace_fault.

If the process is inhibited, then proc2_$deliver_trace_fault checks to see if the caller wants the fault enqueued. If not, it returns status of proc2_$fault_pending. Otherwise, if there is room in the queued fault pool, the routine saves the fault for future delivery by the proc2_$deliver_queued_faults routine. If there is no room in the pool, a fault of fault_$lost_fault is forceably delivered to the process.

The fim_$deliver_trace_fault routine is responsible for seeing that the target process receives a trace fault the next time it leaves the kernel. (Remember that since the delivering process is running, all other processes must be in the kernel, either because of a trap, fault, or interrupt.) The fim_$deliver_trace_fault module performs the following actions:

- Sets the flag fim_$trace_bit[asid]. If the bit was not previously set, it increments the variable pending_trace_faults.
- Changes the instruction at fim_$exit from a return from exception (RTE) instruction to a no operation (NOP) instruction.
- Calls cache_$clear just in case the instruction was cached.

This completes the actions performed by the delivering process.

18.4.3.2. Processing the Asynchronous Fault

The second stage of fault delivery occurs in the target process, starting when that process next leaves the kernel via fim_$exit. The fim_$exit routine is the common exit point to user space for all faults, interrupts, and traps. If no faults are pending for any process, fim_$exit consists of a single RTE instruction. As soon as a fault is pending, the RTE is changed to a NOP, and the following actions are taken:

- The fim_$exit routine examines the supervisor bit in the fault frame through when the RTE is to return. If the status register indicates that return is being made to supervisor mode, it executes an RTE instruction; in other words, the trace fault is retained until user mode is to be entered.
- If fim_$exit is returning to user mode, it checks fim_$trace_bit[asid] to see if a trace fault is pending for this process. If not, it RTEs.

- If there is a pending trace fault, fim_$exit clears the fim_$trace_bit flag and sets the trace bit in the SR of the fault frame.

- The count of pending trace faults is decremented. If it becomes zero, the NOP at fim_$exit is turned back into an RTE (and the cache is cleared).

Note that the proc2_$deliver_trace_fault cannot simply set the trace bit in the appropriate fault frame on the supervisor stack; this is so for the following reasons. On 68000-based machines (DN400, 600), the user-mode status register *always* resides in a "known" place (os_stack_base[asid]-6), and early versions of the trace fault logic did in fact just set the trace bit without the fim_$deliver/fim_$exit logic. However, other processor architectures (68010, 68020) have many different fault frame formats, and there is no fool-proof way to start at the bottom of the stack (highest address) and determine where the user-mode SR resides. You might think that you could assume a 4-word frame (traps, interrupts) and have other fault routines leave a flag around for the trace fault logic. This doesn't work because it is possible to get interrupted **after** a fault frame (for example, a page fault) is pushed onto the supervisor stack, and **before** the first instruction of the fault handler (a disable, presumably). The current solution is to let the target process set its own trace bit, since it always knows where the appropriate SR resides. An alternative is to make all the fault handlers follow standard stack base (SB) conventions. If this were implemented, proc2_$deliver_trace_fault could trace back the target process's stack until it found the oldest fault frame.

18.4.3.3. Taking a Trace Fault Trap

The third stage of asynchronous fault delivery takes place when the target process takes a trace fault trap. When the target process returns to user mode, the trace fault occurs after one user-mode instruction is executed. The trace fault causes entry to the fault interceptor module's trace fault handler. The trace fault code is distinguished from the common fault code (fim_$com) only in that the status code placed in the diagnostic frame is the status that was stored in fim_$trace_sts for the target process.

Although fim_$exit could be designed to obtain the trace fault status and simply jump to fim_$com, it is not implemented this way because to do so would defeat single-stepping, which is one of the uses of the trace fault logic. In addition, the user-mode program may be in the middle of an instruction (pre-fetch or write-behind page fault, for example). Such an implementation would cause all sorts of difficulties by allowing asynchronous faults to occur at arbitrary points during the execution of an instruction.

Running in the kernel FIM does not cause the fault to be acknowledged. This means that proc2_$deliver_trace_fault will not yet allow another asynchronous fault to be posted for the target process. Also, the fim_$quit_value is not set to the fim_$quit_ec.value; this allows process-blocking calls such as ec2_$wait_svc to return with a fault-while-waiting status instead of blocking.

The user fault handler is responsible for acknowledging the fault when it is capable of accepting another. The process manager does this when the fault is dispatched. (Dispatching occurs immediately if not inhibited via pfm_$inhibit, or when the PM's asynchronous inhibit counter reaches zero.)

When a user fault handler calls fim_$acknowledge, the routine does the following:

- Sets the fim_$quit_value to the fim_$quit_ec.value

- Clears the way for another asynchronous fault by setting fim_$quit_inh to false

- Advances the fim_$deliv_ec

18.4.4. Using Quit Eventcounts

Various parts of the AEGIS kernel use fim_$quit_ec to allow eventcount wait operations -- also called blocking operations -- to awaken on an asynchronous fault. Code that wakes up on the fim_$quit_ec must set the fim_$quit_value to the fim_$quit_ec.value to prevent spurious wakeups that could occur between the time the fault is posted (eventcount is advanced) and the time the fault is acknowledged.

Blocking operations within the kernel that want to wake up on an asynchronous fault cannot just wait for fim_$quit_ec.value+1, because an asynchronous fault may have been posted *between* the time that the target process entered the kernel and the time that the eventcount wait is performed. In such cases, a wait for fim_$quit_ec.value+1 will never be satisfied because subsequent asynchronous faults are now inhibited, and the user fault handler will never get to call fim_$acknowledge to re-enable them. Since, however, fim_$quit_value reflects only the previous acknowledged fault, a wait for fim_$quit_value+1 will be immediately satisfied, and the blocking operation has a chance to return the appropriate status (for example, quit_while_waiting).

The user fault handler may have inhibited asynchronous faults itself for its own reasons, and therefore other kernel operations may occur before fim_$acknowledge updates fim_$quit_value to fim_$quit_ec.value. In particular, there may be subsequent blocking operations that also want to wait for asynchronous faults. These operations do not, however, want to terminate immediately just because one asynchronous fault was received but fim_$acknowledge has not yet been called. Thus, the system requires that any eventcount wait (blocking operation) that is "satisfied" by an asynchronous fault should immediately set fim_$quit_value to fim_$quit_ec.value.

In short, fim_$quit_value can be thought of as a partial acknowledgement mechanism that guarantees that an asynchronous fault will satisfy at most a single blocking operation in the kernel.

Chapter 19
SVC Dispatching

The collection of AEGIS kernel modules that user mode programs can run is referred to as the **supervisor**. User-mode code gains access to these modules through the SVC trap instruction, which causes the hardware to execute a trap instruction to the privileged AEGIS service. Thus, user code is said to be *running in supervisor mode* when it takes an SVC trap.

19.1. Changing Mode to Supervisor

The trap page consists of entry points to routines that the processor hardware uses to handle exceptions and interrupts. While most of the trap page contains hardware exception vectors, five vectors in this page are reserved for the AEGIS system to field user space calls to the AEGIS supervisor. These five vectors are the entry points to the code of six trap handlers known collectively as the **SVC catcher**. The trap handler used corresponds to the number of arguments passed in the call. For example, a call to supervisor mode that specifies three arguments generates a trap instruction to SVC handler number three.

The system assigns an SVC code to each supervisor subroutine that user-mode programs can call; these codes are stored in the SVC library, or svclib. For example, the naming server module invoked in response to a WD command (name_$get_wdir_uid) has an SVC code of 34. Because this routine takes one argument, SVC handler number one will process the trap.

When a user-mode program wishes to run a supervisor-mode routine, it places the SVC code number into processor data register D0 (part of the process's processor state) and initiates, via a trap instruction, a trap exception to the trap handler that corresponds to the number of arguments in the call. The generic code sequence is:

```
move.l  #code,d0
trap    nargs
rts
```

where *code* is the SVC code and *nargs* is SVC handler zero through four (for 0 through 4 arguments) or five (for five or more arguments). The example below illustrates the code sequence when user space code calls the naming server's working directory inquiry module.

```
svclib entry for name_$get_wdir_uid module

        entry.p name_$get_wdir_uid
name_$get_wdir_uid equ *
        move.l  #34,d0
        trap    #1
        rts
```

The user-mode caller places the SVC code 34 into D0 and executes a trap instruction to trap handler number one. The hardware handles the trap exception by setting the supervisor bit in the status register and then transfers control to the specified trap handler.

Each trap handler has a table of entry points to the supervisor subroutines, called the **SVC dispatch table.** The SVC number passed in D0 is the index into the handler's dispatch table. The example below shows the format of the dispatch table for handler #1, and identifies the offset to the naming server routine name_$get_wdir_uid.

```
SVC dispatch table for trap handler #1

            extern.p  fim_$install
            ac        fim_$install                   2
            extern.p  network_$read_service
            ac        network_$read_service          3

                        .
                        .
                        .

            extern.p  name_$get_wdir_uid
            ac        name_$get_wdir_uid             34
```

The SVC trap handler does the following:

1. Takes the SVC code from D0 and ensures that it is within the range of defined SVC codes

2. Calculates the offset into the table at which the target supervisor module resides, thereby obtaining the module's entry point address

3. Verifies that the user-mode arguments are below the supervisor global address space protection boundary; that is, that they refer to entities within the caller's private address space. User-mode code cannot change the contents of supervisor global space; thus, the SVC handler checks to make sure the user call is not attempting to make a change to global space.

4. Copies the pointers to the user space arguments (if any) to the per-process supervisor stack and saves the information needed for the return to user mode after the supervisor module completes

5. Calls the target supervisor module

When the supervisor module completes its operation, the SVC trap handler removes the saved information from the supervisor stack and returns control to the user mode caller through a return from exception (RTE) instruction.

Note that the trap handler generates no output to its caller; it simply passes the addresses of the user space arguments. If the supervisor module returns any information to its caller, this information is returned strictly through the caller's arguments.

19.2. User and Supervisor Modes and ASID

Running in supervisor mode is not identical to using ASID 0. (For purposes of this discussion, ASID 0 refers to the supervisor global portion of ASID 0.) Although the operating system -- the code that runs in supervisor mode -- resides in shared supervisor address space, which happens to be identified by ASID 0, user-mode code does not request that ASID 0 carry out some operation on its behalf. User code asks a ***supervisor module*** to do the operation for it. Thus, ASID 0 is an address space, not a procedure.

All processes, whether level 2 or level 1, can access locations in ASID 0. Level 1 processes can ***only*** use ASID 0, while level 2 processes can reference locations in ASID 0 as well as locations in their own private address spaces. However, in order to touch protected ASID 0 locations, a level 2 process must be running in supervisor mode. Thus, level 2 processes gain access to ASID 0 by asking the supervisor, via an SVC trap, to reference the protected addresses.

When a level 2 process runs supervisor-mode code via the SVC mechanism, the process is ***running*** in ASID 0, because it has a supervisor program counter and its active stack pointer points to a per-process supervisor stack. However, the ASID portion of its process context is still the ASID assigned to it at process creation; ASIDs 1 through 25. Thus, the process can continue to refer to addresses within its per-process space as well as to addresses within ASID 0.

Because global addresses are common to all processes, level 2 processes do not have to switch context to ASID 0 when they reference an address within it. So, if a process with ASID 3 is running in the supervisor and takes a page fault in the operating system, the process handles the page fault in ASID 3, but installs the page in ASID 0. During this sequence, the hardware register that tracks ASID will indicate ASID 3, not ASID 0.

Chapter 20
Network Overview

Networking facilities can be separated into three levels:

- The physical network, which provides the medium to transmit messages from one node to another

- AEGIS low-level interprocess communication (IPC) software, which provides the medium for delivering messages to their destination within a particular node

- AEGIS higher-level network support software, which offers a set of remote services to its clients, such as remote access to objects

Figure 20-1 shows the relationship between the managers that handle AEGIS networking.

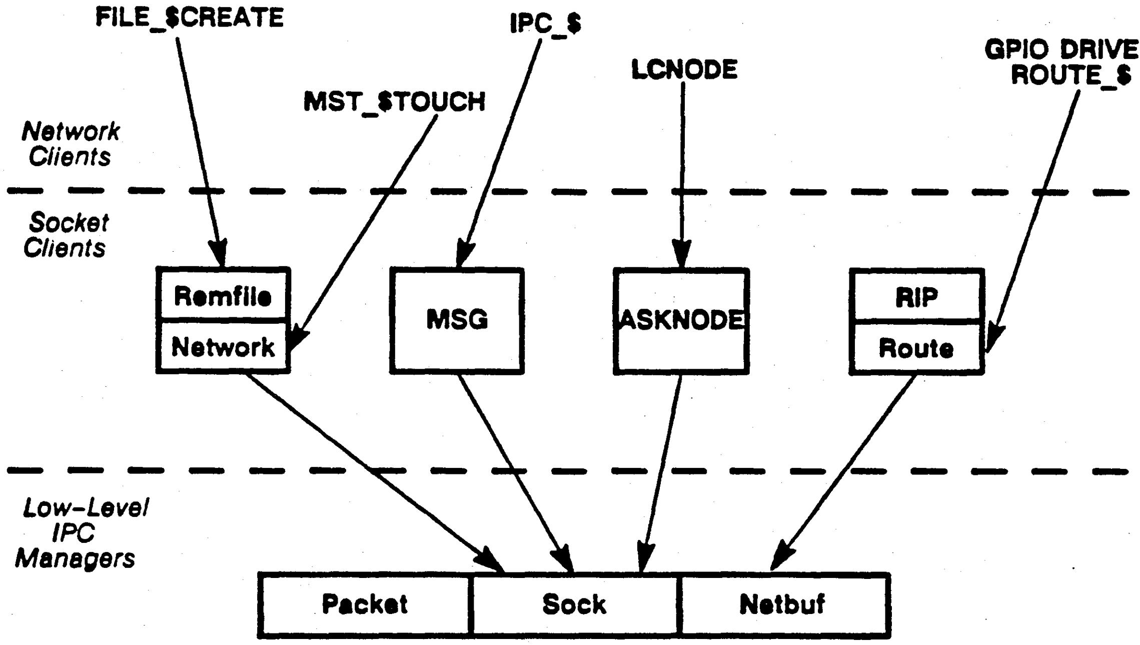


Figure 20-1. AEGIS Network Components

20.1. The Physical Network

The proprietary DOMAIN network consists of a token-passing ring; that is, the communications cable connects the nodes in a circle. In a token-passing ring, a special bit pattern, called a **token**, circulates around the ring, passing through each node. In order to transmit a message, a node must gain control of this token. The node's ring transmitter generates the message, which is received at each successive node and re-transmitted to keep the message going around the ring;

this process is called **transceiving**; the ring hardware carries out the receive/transceive sequence without intervention from the central processor.

Although messages sent by a transmitting node pass through each node on the ring, only the target node actually processes the message; it sends, via direct memory access, (DMA) a copy of the message directly into memory, and records its receipt of the message in a field within the message, called the ACK byte (the ACK byte is part of the low-level IPC message-passing protocol discussed in Chapter 22.) If the node decides to receive a message passing through it, it also awakens the processor by signaling an interrupt.

Because each node transceives the message, it eventually returns to the node that generated it; this node checks the ACK byte in the message for evidence of the target node's receipt and removes the message from the ring.

The token-passing ring design promotes the following features:

- Distributed control of communications hardware
- Graceful degradation under heavy bursts of network traffic
- Automatic acknowledgement of successful transmission
- Support of wiring technologies such as fiber optics and microwave transmission as well as the conventional cable medium

Chapter 21 describes the ring hardware in more detail.

DOMAIN nodes support other types of communications lines in addition to the proprietary ring; for example. a node can support the communications hardware necessary to connect two or more ring networks, and has the facilities to support customer-supplied network hardware that connects to a MULTIBUS controller. For more information on the multiple network environment -- called an **internet** -- consult Chapter 24.

20.2. Low-level IPC Software

The components of low level IPC include:

- The packet mechanism, which is the message-passing protocol used on the network
- The network buffer pool, which provides a pool of pages to use for passing packets around the network
- The socket mechanism, which provides the means for packet delivery to processes within a single node

20.2.1. Packets

Messages are sent around the ring as **packets**. Each packet has a uniform structure, or protocol. All messages on the network use this protocol.

A packet is divided into two variable length pieces called the **header section** and the **data section**. The data section contains the message itself; the packet header contains five distinct types of information:

- Information that directs the packet from one node to another
- Information that the network hardware on a node uses to determine whether or not to accept the packet
- Information that directs the packet from one network to another
- Information that directs the packet to its destination within a node once the ring hardware has accepted it
- Information specific to the AEGIS network support software component that is using the low-level IPC software, called the socket client

The ring transmitter hardware inserts a message separator between the header and data sections, and inserts two more fields after the data page: the cyclic redundancy check (CRC) field and the acknowledge (ACK) byte. Figure 20-2 illustrates the structure of a packet.

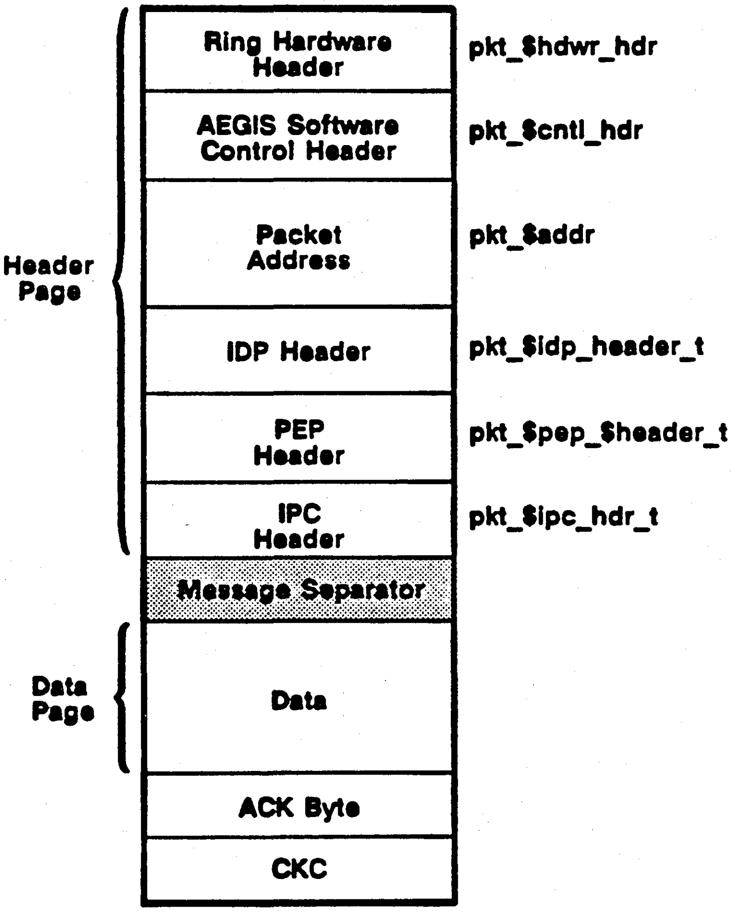


Figure 20-2. Packet Protocol

The division into header and data sections becomes significant when a node receives a packet. The ring hardware sets up two DMA channels to receive a single packet; one channel receives the packet header, while the other receives the packet data. When a packet arrives, the ring receive hardware "DMAs" the packet into memory; when it encounters the message separator, it ends the transmission into the first channel and transfers DMA operation to the second channel. As a result, a packet's header and data pages are scattered into different pages of a node's memory; the socket mechanism "gathers" these pages back together using pointers.

Header and data packet sections exist specifically to support the remote paging protocol described in Chapter 13. A paging request packet must carry a description of the page being requested as well as carrying the actual page itself. The hardware-supported header/data separation allows variable-length protocol fields and page descriptors but still permits the transfer of virtual memory pages to and from page-aligned physical memory buffers. As a result, the network paging protocol never copies page data; instead, it transmits pages from their location in the sending node's physical memory and receives the page data into page-aligned buffers in the requesting node. Consequently, processes can install the pages directly into virtual address space without having to first copy them to page-aligned areas.

The packet manager (PKT) implements the packet transmit protocol and includes calls to build a header, disassemble a header, and to send and receive a packet. Chapter 22 describes the packet protocol in more detail.

20.2.2. Sockets

The socket mechanism is the lowest level of network software. While the ring hardware supports packet addressing to the node level, the socket mechanism exists to get the message to its intended recipient within a node. A **socket** is a numbered message queue. Each socket contains a list of incoming packets, and each socket is identified by a small integer value that is unique within a node but is not unique across nodes in the network. All packets are addressed to both a node and to a socket within that node; as a packet is received at a node, the ring receive interrupt handler removes it from the ring and demultiplexes it to the socket number specified in the "to socket" field in the packet's software control header.

There are three classes of sockets: well-known, reply, and user. **Well-known sockets** are the set of sockets that the system allocates at initialization to the higher-level network support software available in every node; the socket ID for a given network service is the same on all nodes and is known to all potential clients of the service. For example, the remote paging server receives requests into socket number one regardless of the node on which it is running.

The socket mechanism provides the only way to deliver a packet to its intended destination; a packet sent from one node to another always goes through a socket. Thus, a process that needs to receive a message must have a socket into which that message can be queued. Clients allocate **reply sockets** dynamically from a pool of sockets to receive replies to requests for remote services. The reply socket provides a "return address" to send along with the request packet. When the client process's transaction with the server is complete, it returns the reply socket to the pool. The most common means of socket-level IPC is a client reply socket using a well-known server socket. For example, a request to read in a remote object's pages involves allocating a reply socket and sending a page-in message to the paging server, which runs at its well-known socket on the object's home node.

User processes can also gain control of a socket through the message (MSG) interface. The process calls the MSG manager to allocate a socket for its exclusive use; the MSG manager obtains the socket from a pool of **user sockets**. When the process finishes with the user socket, it returns it to the pool.

The AEGIS socket mechanism is based on the datagram model; there are no connections and no set-up. In a socket, the message simply goes out or comes in. The socket mechanism is an unreliable message delivery service; sockets are deemed unreliable because the following conditions can occur:

- Lower level errors can cause packets to be lost

- A full socket queue can cause packets to be discarded

- Packets can get out of sequence or be duplicated

It is the higher level network software that is responsible for ensuring reliable message delivery.

The socket manager (SOCK) maintains the socket pool and contains operations to allocate and free sockets and to insert and retrieve packet entries from a given socket. Chapter 22 describes socket structure in more detail.

20.2.3. The Network Buffer Pool

The network buffer is a pool of virtual pages available for use by clients of the low-level IPC; the pool is divided into a header page pool and a data page pool. Clients use the header and data page pools to hold incoming or outgoing packet header and data information; the pool provides clients with a quick and easy way to get wired pages for low-level IPC transfers. The network buffer pool initially contains enough wired pages behind the virtual pages to handle minimal network traffic; clients can subsequently allocate additional wired pages to the pool.

However, because the network buffer pool size must remain in a steady state, clients that expect to handle network traffic must observe the following rules when using the network buffer pool:

- Clients must allocate as many wired pages **to** the network buffer pool as they intend to take **from** it; they must give the buffer pool these wired pages before they remove any pages from the pool

- When their network operations are complete, the clients must remove all the pages they put into the pool and call the memory map (MMAP) manager to free them

Thus, all low-level IPC clients that request delivery of messages must first allocate storage for those messages in the network buffer pool. Note, however, that the ring receive hardware will not necessarily return the packet to the actual physical page number that the client specifies. Instead, it copies the data into the buffer and returns the client a pointer to that data (via the socket manager).

The network buffer manager (NETBUF) keeps track of the availability of buffer pool virtual pages, makes the virtual-to-physical page associations by calling the MMU manager, and and provides operations to remove and replace header and data pool pages. Chapter 22 describes the network buffer manager in more detail.

20.3. AEGIS Network Support Software

The next level in the network hierarchy consists of kernel and user space software that offers a variety of network-related functions to its clients. AEGIS kernel software provides the following network-related services:

- A remote file system, which permits remote object creation and deletion and provides the ability to gain access to remote objects
- A node status inquiry service that collects statistics and general information about the characteristics of remote nodes
- A user-space interface to the low-level IPC software that allows unprotected AEGIS software and user programs to pass socket-level messages
- A routing service that provides the ability to send a packet to a node on a remote network and provides routing information to the nodes on multiple networks
- A remote naming service that maintains the consistency of multiple naming server network root directories and resolves pathnames over the internet
- The ability to define the kinds of packets that a node will accept

The managers that comprise the higher-level network support software have the following characteristics:

- They communicate with other nodes using the low-level IPC software; that is, they send and receive packets into sockets
- They use a request-response protocol to implement any remote services that they offer to their clients

20.3.1. Request-Response Protocol

AEGIS network software modules that field requests for remote services (and any user space software that offers remote services) implement their remote operations using a request-response protocol that separates the operation into two sides:

- The **client side**, whose primary purpose is to send out requests chosen from its "menu" of remote services
- The **server side**, whose primary purpose is to receive the remote requests, handle them, and pass back the requested information to the client

A process that calls a network service is also known as its client; it calls the service's client side to carry out a particular remote operation on its behalf. The client side uses the low-level IPC to send the service request to the target node where the corresponding server side performs the request. The server then passes back the result as ε reply to the client side, which delivers it to the calling process. Figure 20-3 illustrates the client and server sides and how they operate.

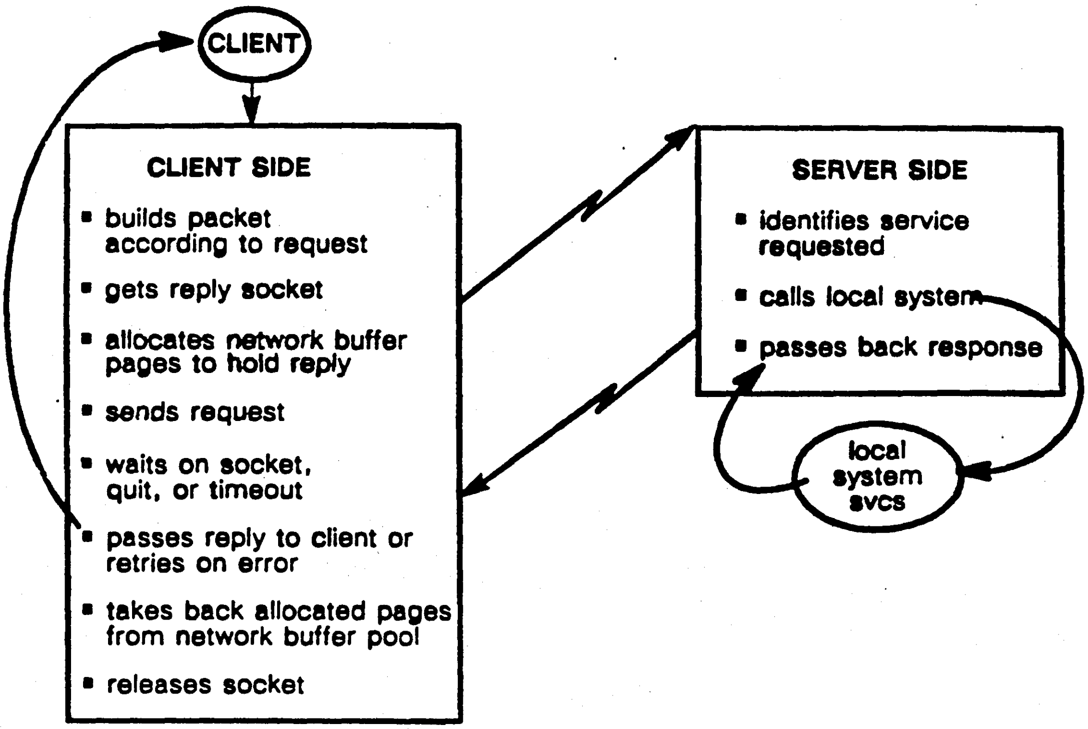

CLIENT
CLIENT SIDE
builds packet according to request
gets reply socket
allocates network buffer pages to hold reply
sends request
waits on socket, quit, or timeout
passes reply to client or retries on error
takes back allocated pages from network buffer pool
releases socket
SERVER SIDE
identifies service requested
calls local system
passes back response
local system svcs

Figure 20-3. Client/Server Operation

20.3.2. Clients of Sockets

Each AEGIS manager that provides remote services uses the socket mechanism and the packet protocol to send and receive replies to requests for remote services. Consequently, these high-level network software managers are known as "clients" of the low-level IPC software. The AEGIS kernel managers that are clients of sockets include:

- The NETWORK and REMFILE managers, which use the low-level IPC and request-response protocol to to implement the remote file system. The remote file manager handles remote requests for object creation, deletion, and attribute manipulation; the NETWORK manager fields requests to read and write a remote object's pages. Chapter 23 describes these managers in more detail; Chapters 5 and 13 describe how they interact with the local file system to carry out remote object management.

- The MSG interface, which exports the low-level IPC facility to user space processes so that they can implement their own network services. The user-space AEGIS network services such as MBX call the MSG manager to gain access to the socket mechanism; user processes can use the MSG interface as well. Chapter 23 describes the MSG interface.

- The ASKNODE service, which sends and receives replies to inquires about a remote node's characteristics; Chapter 23 describes the ASKNODE service.

- The internet routing software, which passes messages to other networks through special routing nodes. The internet routing software consists of a routing process and a module that maintains a table of routing information. The routing process runs only on a routing node; it uses the low-level IPC to receive and forward packets to their destinations in the distributed system. The routing information protocol module runs on every node; for each node, it maintains a table of routing information that routing clients consult when they need to send packets to nodes on other networks. Chapter 24 describes these modules in detail.

AEGIS user-space socket clients include the mailbox facility, the remote server portion of the remote naming service (NS_HELPER), and the diskless node boot server (NETMAN).

Chapter 21
Ring Hardware

Ring hardware components consist of input and output ports, a bypass relay, digital logic and memory. Figure 21-1 illustrates these hardware components.

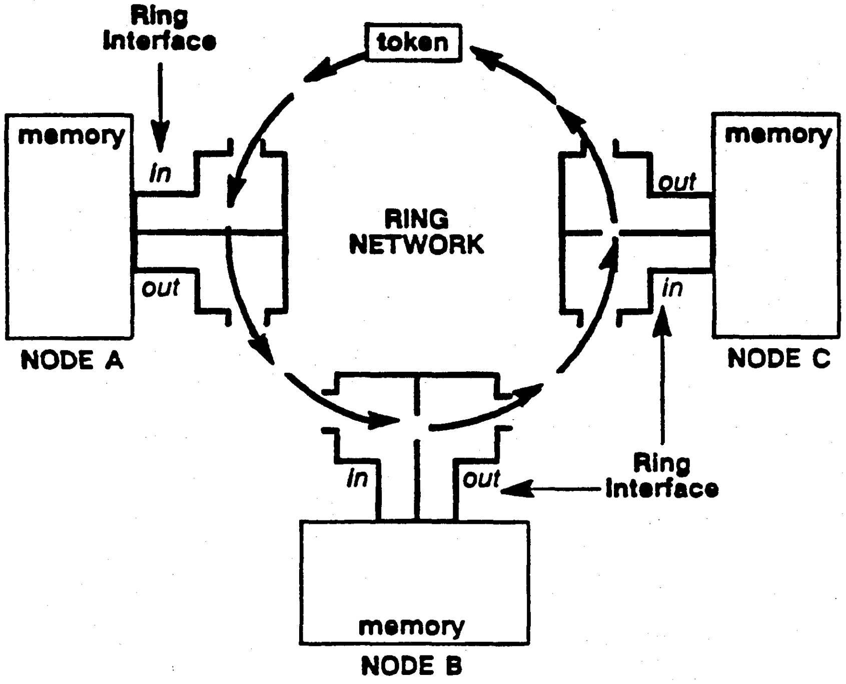


Figure 21-1. Ring Network Hardware

21.1. Ring States

The ring can be in one of three states:

- Disconnected from a node; this state occurs when a node is physically taken out of the ring, either through the NETSVC -N command or a power down. Because the bypass relay from the input to output ports is disconnected, the node cannot "see" the network, nor can the network "see" the node.

- Idle; this state occurs when all nodes are connected and transceiving the token, but no node is transmitting a message.

- Passing a message; this state occurs when a node gains control of the token and enters transmit mode; Figure 21-2 illustrates a message transmission from node B to node C.

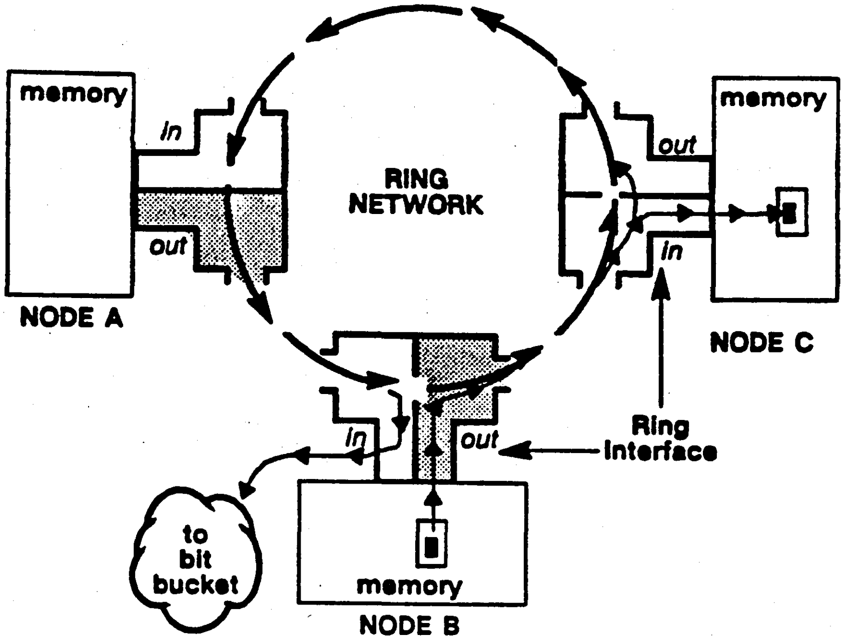


Figure 21-2. Message Transmission on the Ring

21.1.1. Message Transmission

The transmission sequence shown in Figure 21-2 proceeds as follows:

1. Node B enters transmit mode by changing the token's SYNCH character.

2. Node B breaks the ring for a moment and starts streaming the message out onto the ring.

3. Each node in the ring receives the message, decodes it to digital, and examines it. If the message is not destined for that node and the message is not a broadcast (all nodes receive broadcast messages), the node transceives the message.

4. As the message is received and transceived around the ring, Node C recognizes the message, makes a copy of it, sends it into its memory, and acknowledges receipt of the message by modifying the ACK byte.

5. The bits of the message stream around to Node B; this node's transmitter places these bits in a hardware bit bucket, puts the token back out on the ring, and returns the network to the idle state. It then reads the ACK byte and determines that Node C has successully received the message.

A node is allowed to transmit only one message at a time and then place the token back onto the network. The node must then wait for the first message to be correctly transmitted for the token to come around again before it can transmit a second message. Thus, in one circulation of the token around the network, each node has the opportunity to put ONE message on the network.

Note that messages are not recirculated; once the transmitting gets the message back, it proceeds without delay to the receive/transceive logic.

21.1.2. Lost Tokens and Multiple Tokens

During normal message transmission, the transmitting node places the token back out on the ring after its message revolves once around the ring; the transmitting node does not wait to read the ACK byte before it returns the token to the network.

A token can sometimes be lost or destroyed, or can fail to appear when the network is booted. The ring hardware handles these cases as follows. If a node wants to transmit and the token does not pass through it after a specified time, the node assumes that the token has been destroyed and carries out a **forced transmit**. In a forced transmit, the node streams a message out onto the network regardless of its state, and regenerates the token at the end of the message.

A node's transmitter also handles the occurrence of multiple tokens; in this case, it removes the extra token and reports the error.

21.1.3. Transmission Time

Although there is no limit to the number of nodes that can be attached to the network, there is a physical limit between the output port of one node and the input port of another node; that is, you can only send electrical signals so far. However, the time delay created by this limit is negligible -- it takes a message about 70 microseconds to circulate a 700-node network -- and thus does not impact the time it takes data to travel around the ring. The time it takes to place bits on the ring is also short; it takes one millisecond for each 1024-byte message packet.

There is one limit to message transmission, however. When a transmitting node is waiting for its packet to come full circle, it will only wait four milliseconds. If no packet arrives and the timeout expires, the transmitter hardware gives up and reports the error to the network software.

21.1.4. Retransmission on Error

The ring hardware does not perform retransmission on error; hardware retransmits are too expensive, because the DMA operation must be repeated on each transmit. Instead, the network software decides whether or not to retransmit the message. The software judges that certain errors are safe to retry, and does so on the following:

- Wait acknowledge (WACK) -- the node is aware that the message is intended for it, but it has no DMA channels set up to receive the message. In this case, the node sets the WACK bit in the ACK byte to indicate to the transmitter that the message could not be recieved, but to try again later. The AEGIS I/O drivers immediately retry on WACK errors.

- No acknowledge (NACK) -- the ACK byte remains unchanged, indicating that none of the nodes on the network received the message. A NACK error most commonly occurs when node hardware is so busy looking at its local disk that it fails to look at the messages passing through it, or when the node is disconnected from the network. The software also retries NACK errors.

21.1.5. Biphase and Elastic Store Buffer Errors

The signal that travels between one node's output port and the next node's input port contains *clock*. When a node looks at this signal, it can lose clock because the signal is corrupted. A **biphase error** indicates that the receiver, while looking at the signal, could not correctly regenerate the clock from the previous node. A biphase error means that the electrical signal from node A's cable through the modem to node B's cable has broken.

In addition, because a node's input and output ports are individually clocked, slight skews in transmission rate can occur. Consequently, each node has a 1- to 2-bit **elastic store buffer** that takes up the slack between data streaming in and data streaming out. However, it is possible for a node to become so badly synchronized that the elastic store buffer overflows.

Biphase errors are reported by the next node downstream that sees the error; if node A's cable breaks, node B will report the error in the ACK byte. Elastic store buffer errors, however, are detected by the first downstream node that has the most constricted (taxed by synchronization attempts) elastic store buffer; consequently, it is hard to pinpoint the node that caused an elastic store buffer error.

Chapter 22
Low-Level IPC Data Structures

This chapter describes the components that make up the AEGIS datagram service:

- Packets
- Sockets
- Network buffer pool

22.1. Packet Structure

Packets are divided into several header sections and a data section. The hardware inserts a message separator between the header and data sections, and follows the data section with an acknowledge byte and a cyclic redundancy check byte.

Packet headers include:

- The ring hardware header
- The software packet control header and packet address fields
- The internet datagram protocol (IDP) header
- The packet exchange protocol (PEP) header
- The interprocess communication (IPC) header
- The client header

The next sections describe these headers in more detail.

22.1.1. Ring Hardware Header

The information passed in the ring hardware header consists of:

- The node IDs of the source and destination nodes (the "from" node ID and the "to" node ID)
- A clock field which aids in performance-measurement gathering
- The packet type
- An early acknowledge (EACK) field

Figure 22-1 shows the layout of the ring hardware header and the fields within it.

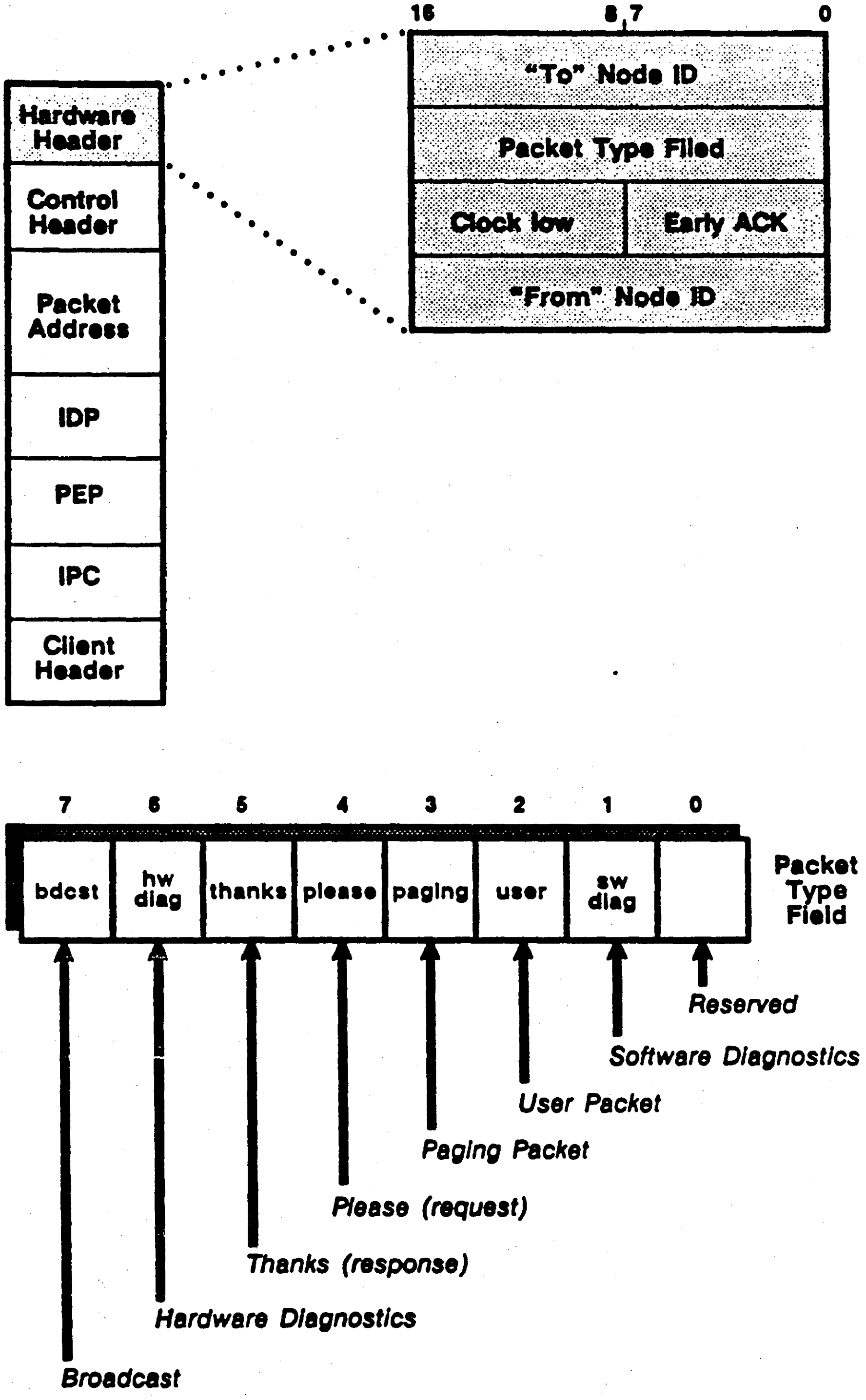


Figure 22-1. Ring Hardware Header

22.1.1.1. Packet Type

The packet type field contains bits that indicate the type of packet being sent around the ring. AEGIS software components assign a packet's type; the ring hardware ANDs the type bits with its hardware type mask register. When a packet arrives at a node, the ring hardware compares the packet's type field against its corresponding type mask register to decide whether or not to physically receive the packet that is passing through it and DMA a copy into memory. The packet type field in the ring hardware header is a 16-bit field; at present, the hardware uses only the first 8 bits. These bits are:

- Broadcast -- when this bit is set, all nodes in the network will receive and process the packet, regardless of the node ID specified in the "to" node ID field. All nodes that receive the packet will modify the ACK byte. Also, because the hardware ignores the "to" node ID field, AEGIS software uses this field by convention to report information on broadcasts.

- Hardware diagnostics -- this bit is reserved for hardware diagnostics; the AEGIS system does not use it.

- Please -- The system distinguishes between a packet that requests a service and a packet that contains a reply to a service request. The please bit, when set, indicates that the packet is a request; that is, the packet's originator is asking for a service from the target node, for example, a paging request.

- Thank you -- This bit, when set, indicates that the packet is a reply; that is, the packet is a response to a request made from another node.

- User -- This bit indicates that the packet is being used for interprocess communication between user processes rather than for communication between AEGIS systems on different nodes (all packets sent from the MSG interface have this bit set).

- Paging -- This bit indicates that the packet is part of the remote paging protocol; it is set on page-outs and attribute changes.

- Software diagnostics -- The system sometimes sends packets marked with this bit set to determine network health or to help announce a problem.

In order for a node to accept a packet and DMA it into memory, the following conditions must *both* be true:

1. The node ID in the "to" node ID field must match the node through which the packet is passing, unless the broadcast bit is set.

2. The node must be willing to accept packets of the type specified in the packet type field.

Various AEGIS components modify the packet type field to control the information flow to particular nodes. For example, the user space network service (NETSVC) program turns off the "please" bit in the ring hardware's type mask register when the -L option is specified. As a result, the node will not accept any request packets (which will have only the "please" bit set); all requestors are NACK'ed. . The node will continue to accept reply packets, but remains unavailable for requests until the the please bit is turned back on in the type mask register.

22.1.1.2. The Early Acknowledge (EACK) Byte

The early ACK byte contains information that the hardware uses during message transmission and acknowledgement. Figure 22-2 illustrates the bits within the early ACK byte.

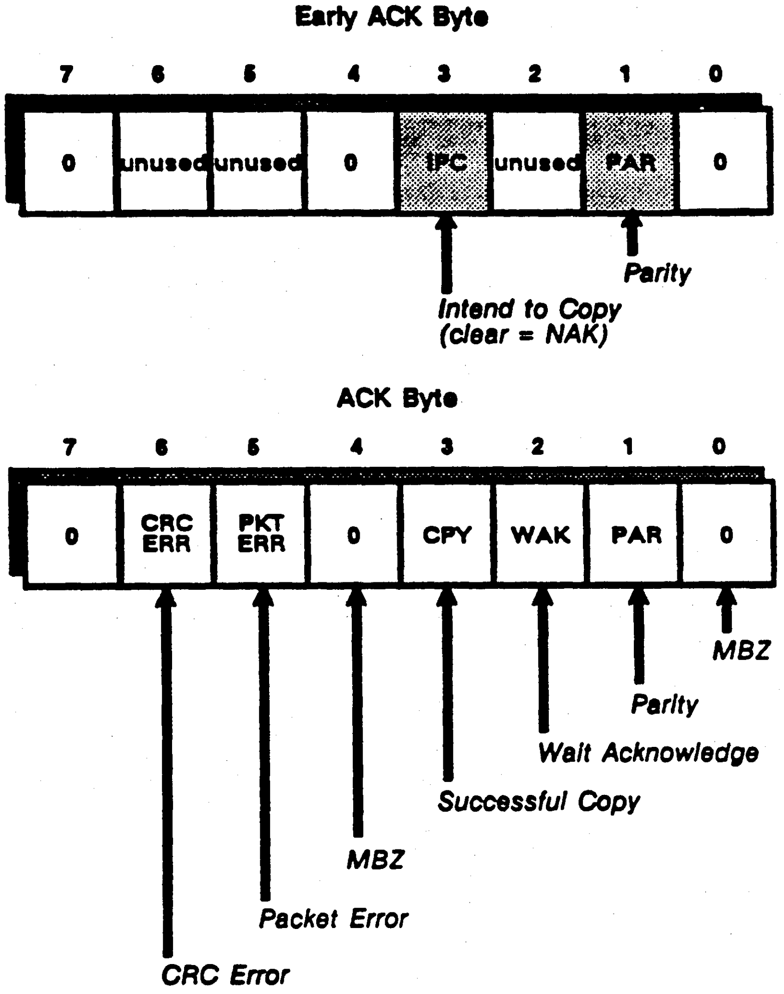


Figure 22-2. EACK and ACK Byte Fields

In particular, the EACK byte contains an *intend to copy* bit (ICP) which, when set, indicates that a receiver intends to try and copy the packet. Because the EACK byte arrives at the transmitting node before the ACK byte, the hardware can use this field to determine whether or not to continue a message transmit. If the EACK bit returns unmodified, it means that no node on the network wants to receive the message. In this case, the hardware aborts the message, taking the message off the ring when the targetted receiver is not there to accept the message. If the EACK byte is modified, the ACK byte that follows at message end will indicate whether or not the target node actually has copied it.

22.1.2. Software Control Header

Pre-internet AEGIS systems and a node's PROM use the software control header and packet address fields to direct the packet to its proper destination within a node once it has been received by the node's ring hardware. The control header contains:

- The socket number within the node
- A transaction ID
- Header, data, and socket queue lengths
- A canned source route, which does not take into account the adaptive routing provided by the AEGIS internet subsystem
- Compatibility information to coordinate message-passing between different versions of the AEGIS system

Figure 22-3 illustrates the fields within the software control header.

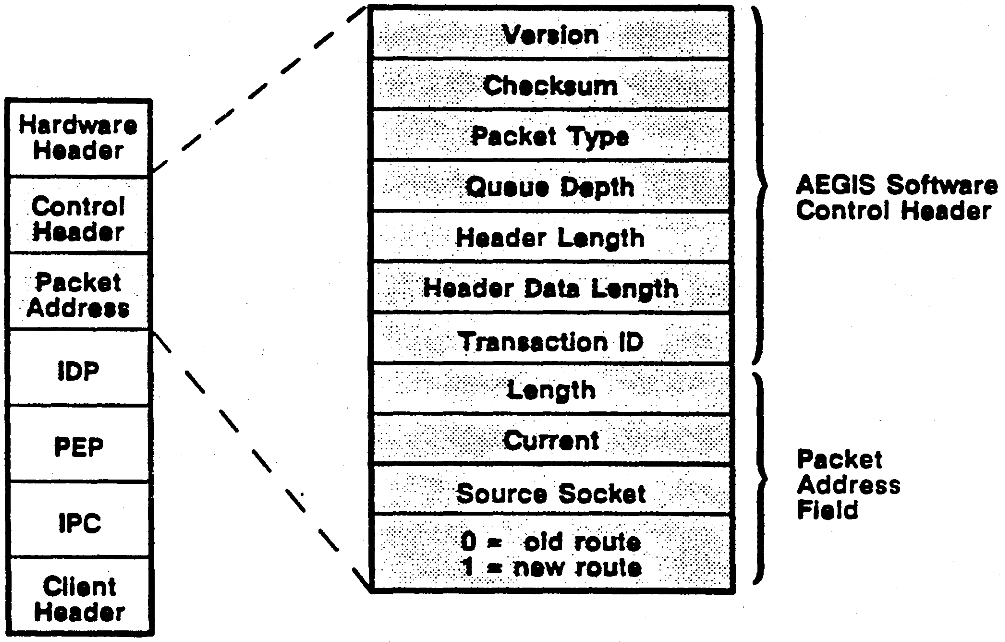


Figure 22-3. Packet Software Control Header

This packet format remains for compatiblity with earlier versions of AEGIS and for sending bootstrap messages to the PROM of diskless nodes.

22.1.3. The Internet Datagram Protocol Header

The internet datagram protocol (IDP) header replaces the software control header of earlier AEGIS systems. The internet routing software uses it to route the packet to a particular network; the system uses it to direct the packet to its proper destination within a node once it has been received by the node's ring hardware. The IDP header contains transport control information as well as source and destination information. Figure 22-4 illustrates its format.

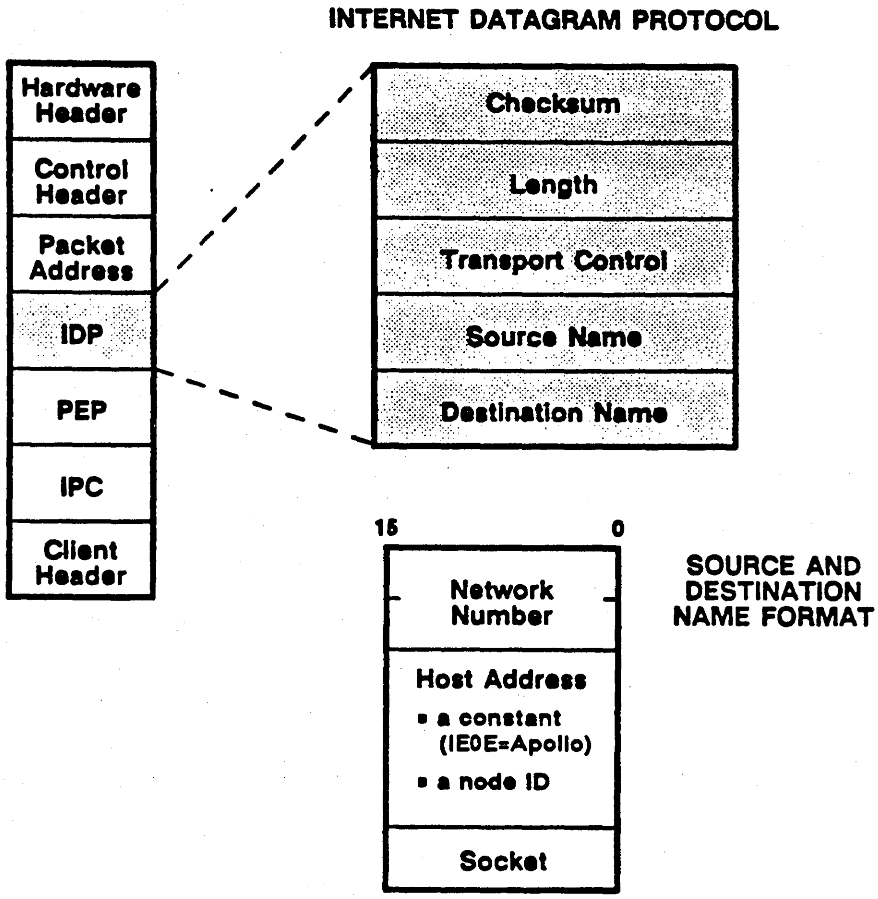


Figure 22-4. Internet Datagram Protocol Header

22.1.3.1. Transport Control

The transport control field counts the number of hops the packet makes in its journey through the internet. A hop consists of one passage through a routing node; the routing process increments this field each time it forwards a packet.

22.1.3.2. Source and Destination Names

A packet's source and destination names consist of:

- A network number, which identifies the communications medium to which the target node is connected.

- The node ID of the target node. While the node ID in the ring hardware header changes with each hop, the node ID in the IDP header remains the same; the packet has arrived at its destination or back to its source when the node ID in the ring hardware header matches the node ID in the source or destination names.

- The socket number within the node, which identifies the socket to which the packet should be queued.

22.1.4. Packet Exchange Protocol

AEGIS network services that implement client/server operations must build a PEP header in addition to the IDP header. Figure 22-5 illustrates the layout of the PEP (and IPC) header.

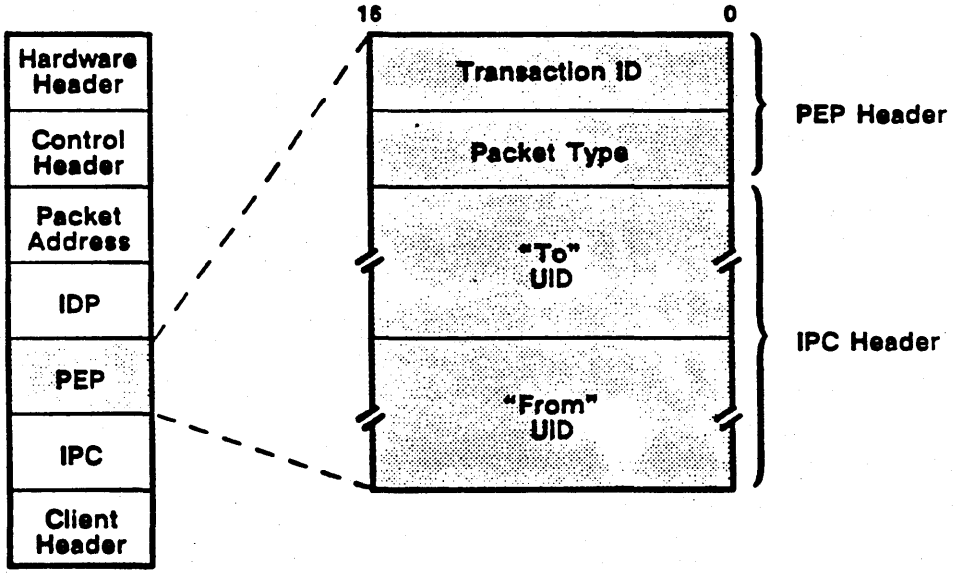


Figure 22-5. PEP and IPC Headers

The PEP header specifies the packet type and also contains a **transaction ID**, which client and server sides use to match up requests for service with the correct replies. Every time a client side issues a packet, it obtains a transaction ID from the packet manager. When the reply packet comes back, the client side checks the reply packet's transaction ID to make sure it matches the one it sent in its request; in this way, clients can synchronize request/reply packets should they arrive out of sequence.

Not all packets need the PEP header; a node that sends a broadcast packet, for example, does not expect a reply and so does not need to build a PEP header.

22.1.5. IPC Header

The user space IPC_$ system services create this header to pass the UIDs of source and destination sockets. (The MSG manager allocates the sockets on the IPC services' behalf; the IPC calls allow users to name sockets with pathnames, which the system translates to UIDs.) This header only exists if the packet is the result of an IPC call.

22.1.6. Client Header Information

A client header, or user header, contains information that is specific to the software component that generated the packet. For example, the paging server creates a client header as part of its remote paging operations. The AEGIS system limits the size of the client header to the remainder of the header page (after the hardware and software headers).

For example, the remote paging server uses the low-level IPC software to pass object pages through the network and inserts information specific to the paging operation into this part of the header. Consequently, packet format differs depending on which client is building the packet.

22.2. The Acknowledge (ACK) Byte

The ACK byte follows the packet data section; the bits within it indicate whether or not the the target node received the packet or if an error occurred during its circulation through the ring.

Figure 22-2 shows the fields within the ACK byte. Their use is as follows:

- A clear byte field (no bits set) indicates that no node has acknowledged the packet; this is a negative acknowledge (NACK) reply.

- The copy bit is set if the target node successfully copied the message.

- The wait acknowledge (WACK) bit is set if the node could not receive the message for some reason but wants the transmitter to try again.

- The packet error bit is set by any node that wants to indicate an error; for example, the ring transmitter sets this bit to indicate DMA underruns, overruns, clocking problems and bus errors; the error bit indicates to all receivers that the packet has been corrupted.

- The CRC bit, if set, indicates that a problem occurred with cyclic redundancy checking.

22.3. Sockets

The system maintains a pool of sockets; it identifies these sockets by small numbers (presently 1-30). Sockets are either allocated dynamically on a process's behalf (reply or user) or are preassigned by the system (well-known). Table 22-1 lists the socket numbers of the well-known sockets and the system function to which it is reserved and identifies reply and user socket pools.

Table 22-1. AEGIS Socket Allocation

Socket No.	System Function
1	Paging socket
2	File socket
3	NETMAN socket
4	Information socket
5	Receives internet asknode "who" replies
6	File server overflow socket
7	Software diagnostic socket
8	routing information protocol (RIP) socket
9	Mailbox socket
10	NS Helper socket
11	TCP/IP
12-16	Reply socket pool
17-30	User socket pool
65536	Bit bucket socket or router socket

If a node is configured as a routing node, the system allocates the bit bucket socket (65535) to receive packets to be forwarded to other networks.

22.3.1. Socket Structure

A socket's structure consists of:

- A level 1 eventcount
- The maximum number of entries the socket can contain (max_depth); currently, a socket can contain up to 16 packet queue entries, but routing nodes and user-allocated sockets can accept more than 16 packets
- The current number of entries queued in the socket
- Forward and backward links the other sockets in the pool
- An array of socket queue entries that point to the incoming packets

When a packet arrives, the ring receive software places two pointers into the socket queue entry: a pointer to the packet's header page and a pointer to the packet's data page. The header page is associated with a virtual address inside the operating system, so that the socket client (the AEGIS network support services) can refer to it. The data page, however, does not intially have a virtual address; it gains the virtual-to-physical address association when it is installed into a process's address space. Figure 22-6 illustrates the layout of a socket.

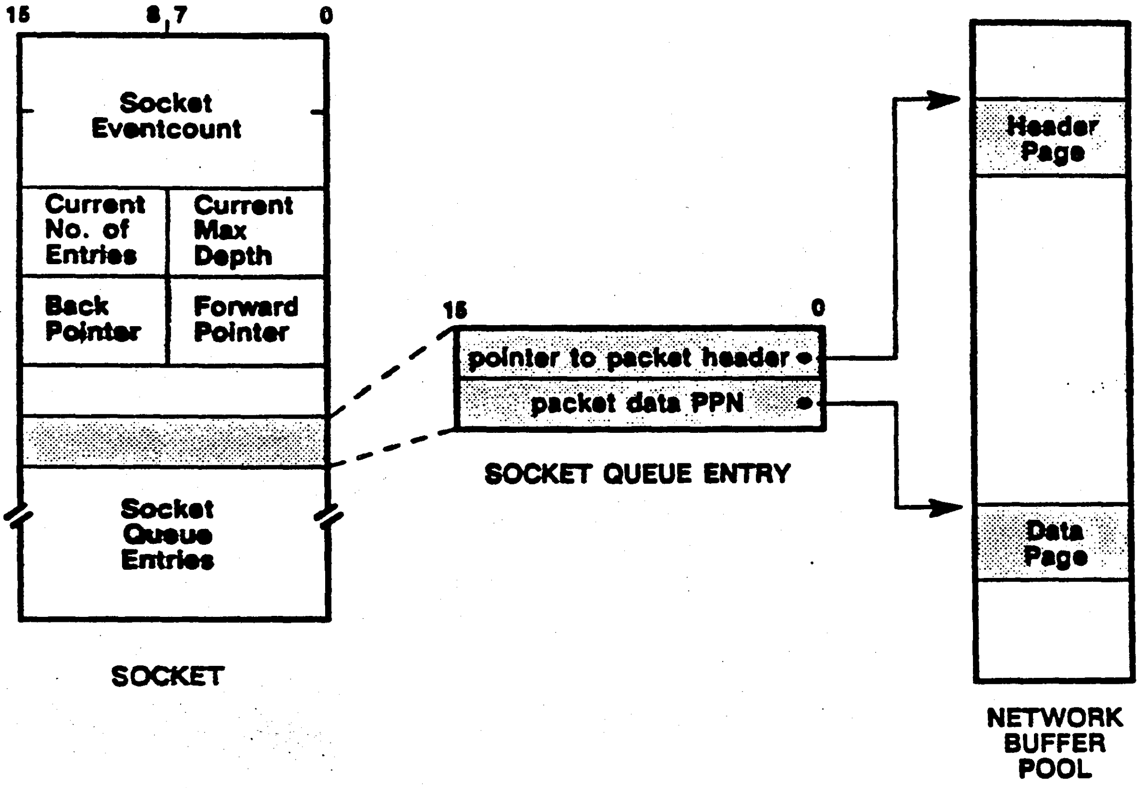


Figure 22-6. Socket and Socket Queue Entry Structure

22.4. The Network Buffer Pool

The network buffer is a pool of virtual pages (192 at present); the network manager's initialization routine reserves the virtual address space for the network buffer pool at system initialization, and allocates to it enough wired pages to handle minimal network traffic. The table is subsequently managed by the NETBUF manager (which the NETWORK manager initializes); this manager tracks the availability of network buffer pages using a table (BUFTAB). Each entry in the table correponds to a virtual page; if the page is in use, the entry contains its physical page number (PPN); otherwise, the page is available (available pages are indicated by a value of -1.)

Clients that expect incoming packets must allocate network buffer pages to receive the transactions and must free those pages when they are finished with them.

22.4.1. Allocating Pages to the Pool

First, the client must obtain wired pages to pass to the network buffer manager. The PMAP manager contains a routine, wp_$calloc, to conditionally allocate a page by calling mmap_$alloc. Nearly all the network clients call this routine to get as many wired pages as they need to handle the number of packets they expect to receive (the PMAP manager calls its own internal allocation routine instead of calling wp_$calloc.)

Next, clients call the netbuf_$getva routine to associate virtual pages in the pool with these newly acquired physical pages. The netbuf_$getva routine pairs the given physical page number(s) with an available virtual address, notes the pairing in the buffer table, and returns the virtual address to the client. It also calls mmu_$install to make the virtual-to-physical association (the access to a network buffer page is supervisor read/write).

The network buffer pool is actually two pools: a buffer pool for header pages and one for data pages. After they receive the virtual addresses, clients next call the netbuf_$rtn_hdr or netbuf_$rtn_dat to add the physical pages to the header or data section in the network buffer pool. If the NETBUF manager doesn't need these PPNs, it will free them up itself rather than returning to the client.

Finally, the clients turn around and call the NETBUF manager (via netbuf_$get_hdr or netbuf_$get_dat) to get the same number of pages from the preallocated header or data page pools as they have previously put in; this operation keeps the pool in a steady state.

22.4.2. Removing Pages from the Pool

By convention, all clients of the low-level IPC service must remove any pages they allocate to the network buffer pool when they complete their IPC operations. Routines call netbuf_$get_hdr and netbuf_$get_dat to remove physical pages from the network buffer header and data pools. The routines then call the MMAP manager (mmap_$free) to put these pages on the free list.

Chapter 23
AEGIS Network Support Software

This chapter describes the managers that comprise the higher-level AEGIS network support software and the functions they perform. It describes:

- The NETWORK manager
- The remote file (REMFILE) manager
- The message (MSG) manager
- The ASKNODE service

23.1. The NETWORK Manager

The NETWORK manager carries out the following operations:

- Initializes network-related databases and servers during system initialization
- Exports the ring hardware type mask to AEGIS software; programs such as NETSVC call the NETWORK manager to set and clear bits in a ring hardware type mask to control the kind of packets a node will accept
- Provides paging request services portion of the remote file system to its clients

The NETWORK manager includes the following network support servers:

- The paging server
- The request server

23.1.1. System Initialization Functions

The NETWORK manager is responsible for initializing the network buffer pool and the MSG data structures when the system is bootstrapped. In addition, it invokes the network support servers needed for full higher-level network operation; these servers are:

- The ring receive server process, which handles incoming packets from the ring if the ring interrupt handler encounters a problem
- The remote paging server
- The remote request server, which is the "shell" process that dynamically calls the remote file server, the asknode server, and the routing information protocol (RIP) server should packets arrive at their sockets

23.1.2. Packet Type Export

The NETWORK manager provides routines that other AEGIS components can call to define the set of service requests that a node will handle. The service types correspond to the bits in the ring hardware's type mask register and the packet type bits. The NETWORK manager's add service and drop service routines set and clear these bits to allow or prevent the node from accepting packets of a certain type.

Service Type	Bit in Type Mask
Add/drop local requests	Sets/clears the thank you bit
Add/drop remote requests	Sets/clears the please bit
Add/drop paging requests	Sets/clears the paging bit
Add/drop user requests	Sets/clears the user bit
Add/drop broadcast requests	Sets/clears the broadcast bit
Add/drop diagnostic requests	Sets/clears the diagnostics bit

The NETSVC command calls this portion of the NETWORK manager to set the network services that a node will perform; for example, NETSVC -L clears the please bit, so that requests for service that originate from other nodes will be rejected.

23.1.3. Paging Services

The NETWORK manager provides the following menu of *paging* requests to its clients:

- Page-in requests, which read in object pages from another node in the network (network_$read_ahead)

- Page-out requests, which send out remotely modified pages back to the home node (network_$write)

- Attribute requests, which modify the attributes of a remote object (network_$set_attributes)

- Information requests, which retrieve a remote object's attributes (network_$ast_get_info) from the local OSS (vtoce)

- Ring information requests, which get ring status from another node in the network. The ring device driver on each node keeps statistics on the number of sends and receives that encountered biphase errors, elastic store buffer errors, and other information.

- Echo requests, which test to see whether a given paging server is running or not

When a client requests one of these services, the NETWORK manager sends the request to the paging socket on the target node, where the remote paging server will handle it.

23.1.4. The Remote Paging Server

The remote paging server performs the following functions:

- Handles paging requests
- Flushes the NETLOG buffer
- Handles sticky biphase errors
- Handles overflows in file server socket

The paging server handles these operations because they are services that must be handled in a timely fashion by a wired process.

23.1.4.1. Paging Request Handling

The paging server listens on the paging socket to detect paging requests. When a packet arrives in the socket, the paging server breaks it down so that it can determine the kind of request. It handles client requests as follows:

- For page-in requests, the paging server calls ast_$touch.
- For page-out requests, the paging server calls ast_$touch.
- For attribute requests, the paging server calls ast_$set_attribute to execute the call. When reply packet is sent back successfully, remote paging server modifies the object's DTM.
- For information requests, the client header specifies the type of information requested about the object. The paging server passes the request arguments to ast_$get_info, which fetches the object's VTOCE. The paging server then sends back the contents of the VTOCE in the reply packet.
- For ring information requests, the paging server builds a ring diagnostic record to send back ring status information.

Chapter 13 describes the remote paging requests in more detail.

23.1.4.2. File Socket Overflow

Normally, if a socket overflows, the low-level IPC facility drops the packet; meanwhile, the client who sent it waits for the reply until his timeout expires, then retries the send. However, the ring receive interrupt handler special-cases packet delivery to the file server socket. If the file socket is full and cannot accept any more packets, the ring receive code sends the packet to the file overflow socket instead. The paging server waits for incoming packets on this overflow socket. The paging server handles file socket overflows because the remote file server cannot do so; it is either busy with a page fault or with some long operation.

When an overflow packet comes in, the paging server changes it to indicate overflow, then sends the packet back to the REMFILE client side, who then retries the operation. Because the paging server sends back the overflow reply quickly, this method of overflow handling allows the client to retry more quickly than if it waited for its timeout value to expire.

23.1.4.3. Flushing the NETLOG Buffer

The paging server waits on the netlog eventcount to detect writes to the network logger (NETLOG) buffer. The paging server, rather than the file server, flushes netlog pages because the file server is usually logging events into the NETLOG page, thus contributing to the need to write it out. The paging server calls the user-mode NETLOG manager to actually flush the buffer.

23.1.4.4. Sticky Biphase Errors

The system monitors biphase errors on the ring using the "sticky biphase" bit in the ring hardware register; the bit is set if a biphase error occurs. In the past, sticky biphase errors were detected only if the node was receiving or transceiving. Currently, the paging server checks the hardware bit once every 5 seconds to see if it's set, and reports the condition if so. This operation provides continuous monitoring of network errors.

23.1.5. The Remote Request Server

The remote request server listens on the file socket, information socket, and RIP socket; when a packet arrives in one of the sockets, it calls the appropriate server. The request server handles "asknode who" (LCNODE) requests specially. The servicing of an asknode request does not prevent the request server from handling incoming packets in other sockets. Instead, the asknode server asks for a delay; at its end, the request is propagated and normal asknode requests are re-enabled.

23.2. The Remote File Manager

The remote file (REMFILE) manager handles remote requests for location-independent object management, directory maintenance, and lock management. The manager maintains a database of remote request/response pairs that provide a menu of services from which remote nodes can choose a request.

The menu of services includes:

- Concurrency lock, unlock, and verification requests (file_$priv_lock/unlock, file_$local_verify)

- Directory search (dir_$root_get_entryu) requests from naming servers on remote nodes

- Requests to add or drop hard links from the local directory structure (dir_$add/drop_hard_linku)

- Requests to create and delete objects (file_$create_type) (ast_$truncate, ast_$invalidate)

- Requests to purify or invalidate an object (ast_$purify, ast_$invalidate)

- Requests to set an object's attributes (ast_$set_attribute)

- Echo requests, where the client sends a test message to the target remote file server to determine whether or not it is running

The REMFILE client side passes the arguments in a packet to the remote file server on the target node; the remote file server handles the request by calling the appropriate local AEGIS manager (the FILE, AST, or directory managers), then passes back the answer to the client in a reply packet.

Figure 23-1 illustrates the manager's operation.

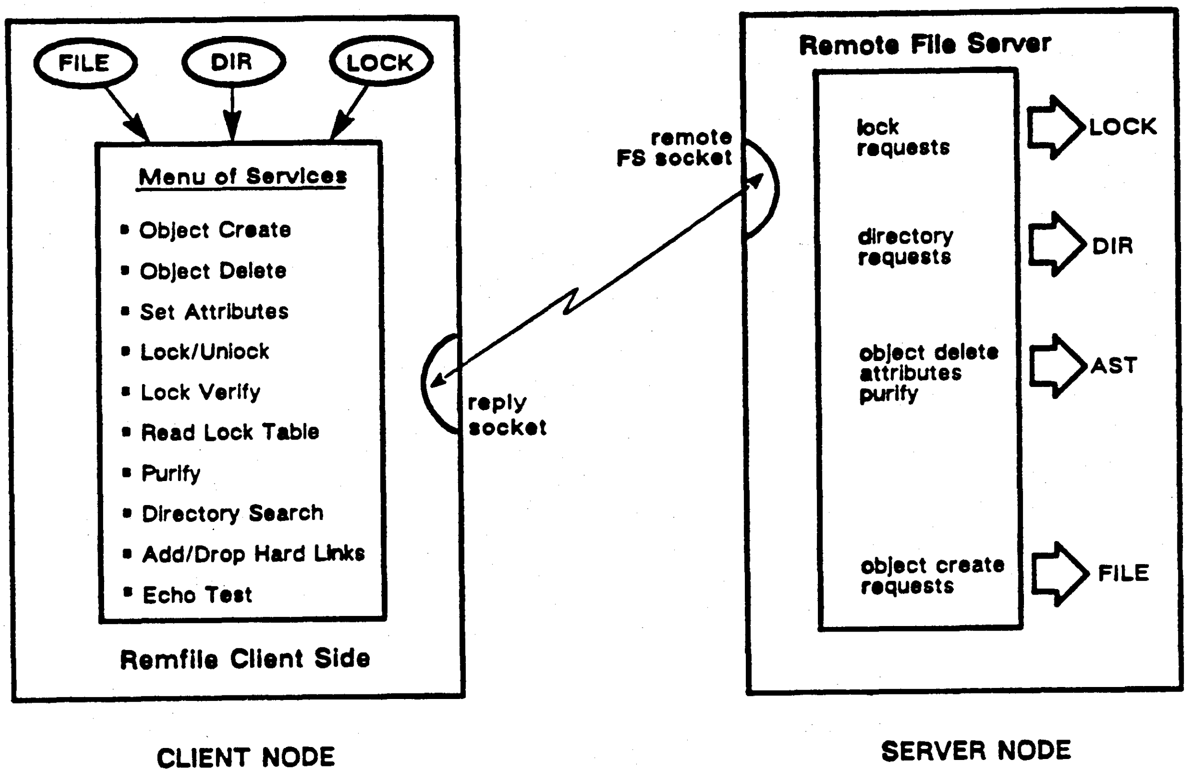


Figure 23-1. REMFILE Operation

23.2.1. REMFILE's Client Side

The REMFILE client side is a very "patient" service; it will retry a failed request many times before it sends a "communications problem with remote node" message.

The REMFILE client side proceeds as follows:

- Allocates a reply socket
- Gives the network buffer pool manager (NETBUF) a page to hold the eventual reply
- Sends the message, and waits on the following events:
    - The reply socket eventcount -- if this eventcount advances, the REMFILE client side tries to get the reply.
    - The quit eventcount -- quits terminate the request.
    - Its own timeout value -- if the reply wait times out, the REMFILE client tries again (but decrements its retry count by a large margin).

If the client is awakened by the reply socket eventcount, it retrieves the reply packet and checks the transaction ID within it to make sure the response matches the request so that it has obtained the correct response. If the transaction IDs do not match, it resends the message.

If the reply packet indicates that the request didn't get through because the file socket has overflowed, the client resends the message.

In the event that the client side has received the data its client requested, it completes by:

- Closing the socket
- Removing network buffer pages from pool and freeing them
- Sending the transaction ID back to caller

23.2.2. Remote File Server Operation

Although the remote file server actually handles the remote request, it is the request server that "listens" to the remote file server socket and invokes the remote file server when a packet comes in. The remote file server in turn calls the appropriate local manager to execute the request, then packages up the reply to send back to the client. Before it sends the reply, however, the remote file server must check the kind of request it has processed. Certain requests should not be duplicated, because processing the request twice in the event that the client retries will cause problems to occur.

For example, if a REMFILE client sends a request to create an object, and the remote file server is unable to send back the reply before the client's timeout value expires, the client will simply issue a duplicate request, not knowing that the object has already been created. If the remote file server were to process this duplicate request, it would create two objects to satisfy the one request.

Consequently, the remote file server checks to make sure the operation is idempotent; if it isn't, the server then compares the request packet's arrival time against the time that it finishes processing the request. If there is too great a discrepancy and a client retry is thus imminent, the server will NOT send back a reply packet. Instead, it "backs out" of the operation it has performed by undoing the operation it has performed on the remote client's behalf; for example, it deletes the object it has just created. Currently, object creation and concurrency lock requests are the only non-idempotent system operations.

23.3. The Message Interface

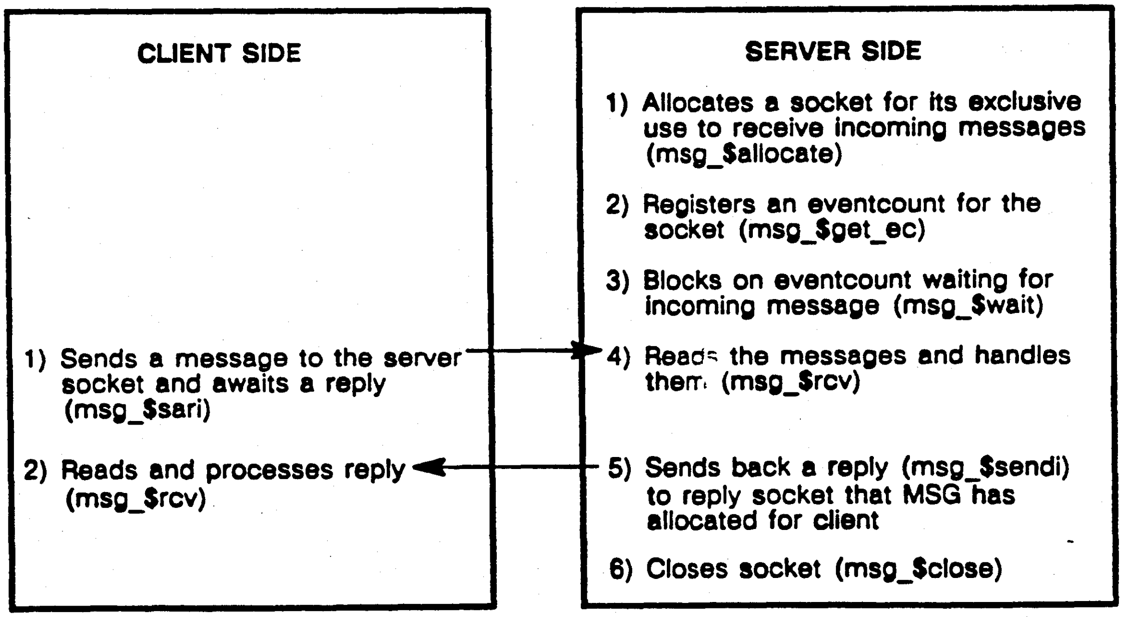
The message (MSG) interface provides user-mode AEGIS services and individual user programs with access to the socket datagram service. Programs that want to use the socket datagram service to pass messages implement a user-written client/server application program, then use the MSG interface to send and receive socket-level messages on the application's behalf.

The "application" can be an AEGIS network service that runs in user mode, for example, the mailbox facility (MBX); these applications call the MSG manager directly. User-written applications make calls to the IPC services documented in the Programming With System Calls for Interprocess Communication manual to gain access to the socket mechanism.

The MSG manager allows the application's server side to allocate a socket for its exclusive use: no other processes can receive incoming packets into the socket while the process is using it. The server waits on a level 2 eventcount associated with the socket (a registered level 1 eventcount; see Chapter 17) for any incoming messages, then reads them, handles them, and sends a reply back using the calls that the MSG manager provides.

Clients of the application's socket use the MSG manager to allocate a reply socket, send the remote request to the server at the user-allocated socket, and wait for the server's eventual reply.

Figure 23-2 illustrates the MSG calls that the application's client and server sides use.



CLIENT SIDE

1) Sends a message to the server socket and awaits a reply (msg_$sari)

2) Reads and processes reply (msg_$rcv)

SERVER SIDE

1) Allocates a socket for its exclusive use to receive incoming messages (msg_$allocate)

2) Registers an eventcount for the socket (msg_$get_ec)

3) Blocks on eventcount waiting for incoming message (msg_$wait)

4) Reads the messages and handles them (msg_$rcv)

5) Sends back a reply (msg_$sendi) to reply socket that MSG has allocated for client

6) Closes socket (msg_$close)

Figure 23-2. Request-Response Protocol Using MSG

23.4. The ASKNODE Service

The ASKNODE service allows clients to request and receive miscellaneous information about other nodes on the network. In particular, the ASKNODE client side sends requests to:

- Return the list of nodes that are currently responding
- Inquire about system software and hardware performance, such as the frequency of socket overflows and the number
- Obtain network statistics, such as number of page-ins, page-outs, read and write calls, and violations
- Obtain network and node root directory information
- Obtain AEGIS system build times
- Obtain process information so that the client can send a trace fault to a remote processes
- Inquire about ring status, such as number of biphase or elastic store buffer errors
- Obtain a node's hardware configuration, such as whether or not it has a PEB
- Obtain information about the network ports on a given node

When a packet arrives at the information socket, the request server invokes the ASKNODE server to handle it. The server gets the packet, processes it, sends the reply back to the requestor, and returns to the request server.

If the request is to find out "who's there" in the network, the ASKNODE client side sends out a broadcast message that specifies the *who* socket as the reply socket. The first node to receive the broadcast sets the ACK byte, which indicates to the other nodes that the packet is not for them. The node then sends two messages: it returns a response to the originating node, and rebroadcasts or **propagates** the "who's there" packet. In this way, each successive node in the network receives the broadcast and sends back a response. From the time the node receives the broadcast packet until the time it sends the reply and propagates the packet, no other ASKNODE requests will be serviced. This procedure prevents a socket overflow on the originating node or on a routing node if the "who" is being broadcast over the internet.

When it determines that the broadcast has completed, the client side returns a list of the nodes that are currently responding to the process that called it. Currently, only one process at a time can make an ASKNODE "who" request.

Chapter 24
The Internet Subsystem

A **network** can be defined as any communications medium that allows the AEGIS system to page through it; that is, it allows packets to be sent through it. An **internet** is a group of two or more connected networks. An internet has several advantages over a single network:

- It permits DOMAIN nodes to be connected to different types of local area networks

- It allows rings of varying speeds to be connected together

- It improves network management by using many small rings, so that a ring can be isolated without impact on the rest of the network

Packet transmission on a ring network has been described in Section 22.1. In an internet, a packet can be destined for a local network or a remote network. Packets destined for remote networks must pass through one or more interim networks before they reach the target network; these interim networks can be other ring networks, T1 bridges, or user-supplied MULTIBUS-compabitible network devices. The process of sending a packet from one network to another is known as **routing**. The node that carries out the routing procedure is called the **routing node**.

Figure 24-1 shows a theoretical internet.

24.1. Identification in an Internet

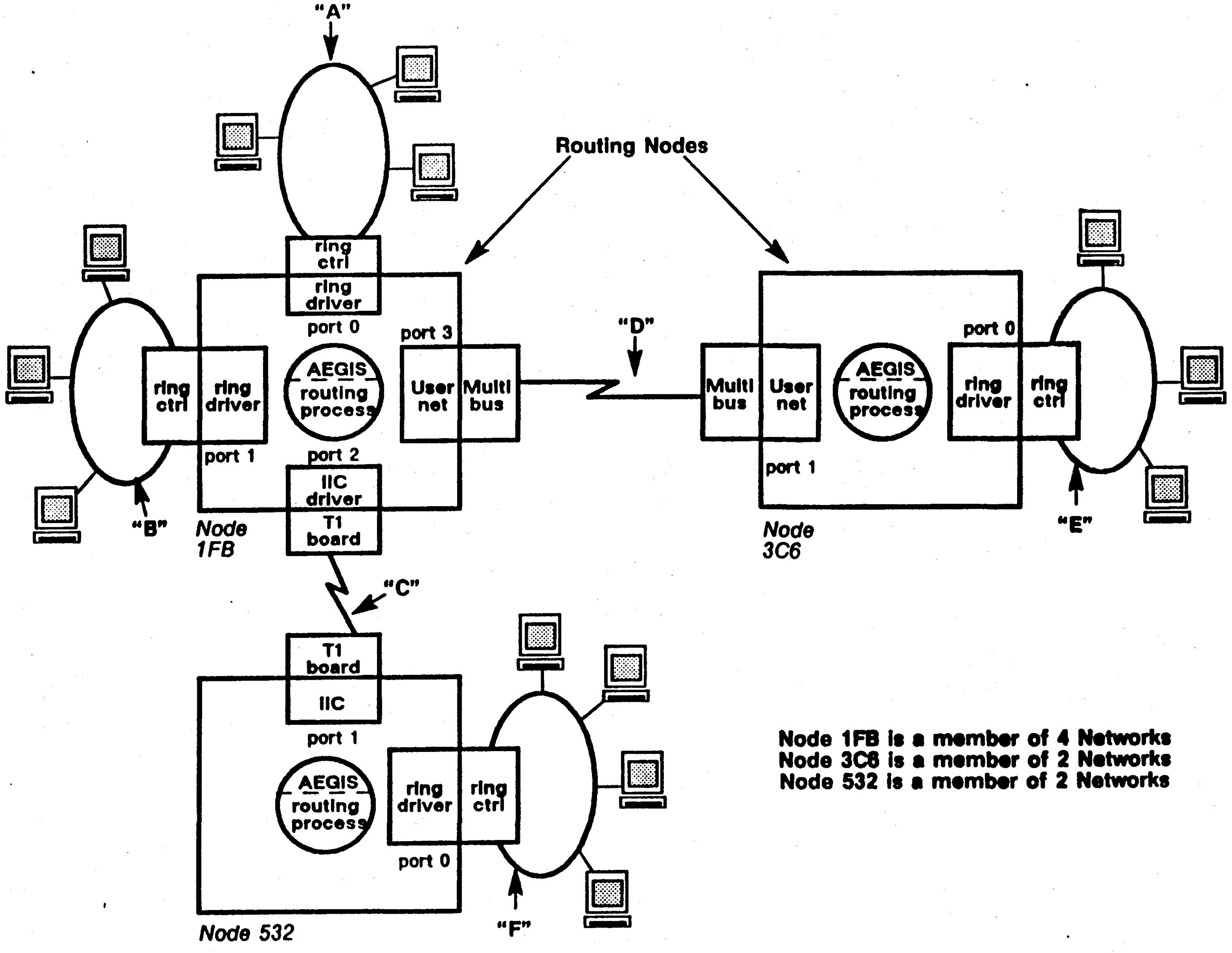
There are three types of identification in an internet:

- The network number, which identifies the communications line

- An internet address, which identifies a node in an internet

- A network port descriptor, which identifies a network in a device- independent way

The network number and internet addresses are readily visible to other nodes in the internet; the network port descriptor is not.



Node 1FB is a member of 4 Networks
Node 3C6 is a member of 2 Networks
Node 532 is a member of 2 Networks

Figure 24-1. An Internet

24.1.1. Network Number

Each network in the internet (there can be a maximum of 64) is assigned a 32-bit network number that identifies the communications line. Each node stores these network numbers in a table that the NETWORK manager maintains.

A network number of zero can mean two things:

- The network is being brought into the internet and has not yet been assigned a network number

- The network is local; that is, a network ID of zero is used as a "you are here" pointer in the internet topology, where "here" differs depending on the location in the internet.

Network numbers can also be nil; that is, there's no network at this number. Nil networks usually indicate an empty MULTIBUS slot.

24.1.2. Internet Address

System components that want to communicate with another node in the internet need to identify the network on which it resides as well as identifying the node itself. Consequently, system components that send packets to other nodes specify the node's **internet address**, which consists of the 20-bit node ID plus the network number on which the node resides. Figure 24-2 illustrates the internet address.

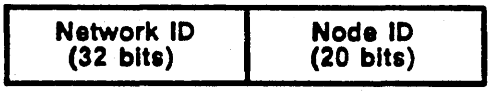


Figure 24-2. Internet Address Format

The system needs an internet address to gain access to an object; consequently, the object locating services (MST, AST and HINT) and the lock manager cache information about an object's internet address in their data structures. In particular, the VTOC indexes in mapped and active segment table entries store an index into the network number table; when the system wants to access an object on the internet, it passes that index to the NETWORK manager, which returns the network number that corresponds to the index (network_$get_net). The system then has an internet address to place into the packet.

Because routing nodes are members of more than one network, they have several internet addresses; that is, the network number portion of the internet address may be the network number of any network connected to the routing node.

24.1.3. Network Ports

A network is linked to the AEGIS system through its port. All nodes have at least one port, which identifies the ring to which they are connected. Routing nodes have several ports; consequently, the network's port descriptor provides the routing software with a device-independent way to refer to the different kinds of networks attached to it.

A network port descriptor is a small integer that the routing software assigns when it connects the network to the internet and opens it for AEGIS paging operations. The port descriptor is the node's way to refer to the network hardware connected to it, and is important only to the node that owns the hardware.

Currently, a node can support seven ports. The first port (0) is reserved to the first ring network; the other ports can identify other rings, T1 (bridge) lines, or MULTIBUS network devices (4 ports are reserved for user-added network devices).

The routing software uses a network's port descriptor as an index into the network port table, which describes the network devices connected to the routing node. Each entry in the table gives the following network-specific information:

- The network device's network number
- The kind of service the network provides; for example, whether or not the port supports full routing
- The kind of network device that exists at that port; for example, ring, T1 or user network

24.2. Internet Software Components

Both routing nodes and non-routing nodes run internet software that supports packet routing; this software consists of the following components:

- A routing information protocol (RIP) table, which exists on all nodes. The routing table provides routing information that processes use when they send packets to destinations in the internet.
- A routing process, which runs only on nodes dedicated to the performance of internet routing functions. The routing process forwards packets to their internet destinations and periodically sends out the latest routing information to other nodes in the internet.
- A routing information protocol (RIP) handler, which receives and stores the routing information packets sent out by routing nodes.
- A device-independent network I/O manager that fields packet sends and receives to the various networks connected to a routing node.
- Device-dependent network drivers that manipulate the network hardware attached to a given routing node.

24.2.1. The Routing Table

The routing information protocol (RIP) table is a per-node data structure that stores routes from the node to other networks in the internet. The routing information in the table depends on the node's location in the network. Each entry in the RIP table corresponds to a network in the internet. The routing information stored includes:

- The network number of the target network
- The number of hops it takes to get to that network; that is, the number of routing nodes the message must pass through before it reaches its destination network
- The next hop towards that network
- The next network port towards that network

The RIP table entry also indicates how reliable the information within the entry is; it specifies a time after which the entry is considered unreliable, and has a status field that indicates whether the routing information is fresh, old, or has expired.

Routing nodes must wire their RIP table (the table consumes approximately 1 page).

24.2.2. The RIP Handler

The RIP handler is a request-response service. The client side allows processes to request routing information from local RIP tables and from the RIP tables of other nodes in the internet. The server side, called the RIP server, has its own well-known socket to receive incoming routing information packets sent by routing nodes in the internet.

A node that wants to obtain a route through the internet consults the RIP handler's client side (rip_$find_nexthop). The RIP handler then looks at its table to find the optimum path, given the destination internet address, and returns the ID of the routing node that should forward the packet and the port through which the packet should pass. The RIP manager always chooses the most expedient path to the target, regardless of amount of traffic that path is presently incurring. The most expedient path is the path with the smallest number of hops.

The RIP handler also provides a broadcast routine that the routing processes call to send out the latest routing information to all the other nodes in the internet (rip_$broadcast). A RIP broadcast packet entry contains the network number of the target network and the number of hops away it is. The RIP servers on each node receive these broadcasts and modify their routing tables accordingly.

24.2.3. The Routing Process

Every routing node runs a routing process, also called a **router**. Whenever two or more network ports are open for routing service, the routing software (route_$set_service) starts a routing process. The routing process has two functions:

- It forwards packets through network ports
- It supplies nonrouting nodes with the information they need to maintain the most current routing information in their routing tables

The network I/O manager (net_io_$put_socket) checks packets that arrive at a routing node to see whether they are intended for the routing node itself or are "through traffic" to another network. Packets for the routing node are placed into the socket specified in the packet header; for example, a request for the routing node's build time goes to the information socket. Packets that should be forwarded go to the routing node's routing process socket regardless of the socket specified in the header.

The routing process never retries a packet forward; in addition, it will drop packets that have been forwarded too many times and packets that have been misrouted.

24.2.4. Device-Independent Network I/O

The network I/O manager is the system's network-device independent send and receive component. It permits the AEGIS system to handle I/O to and from multiple network devices. The network I/O manager fields all higher-level transmits to the appropriate driver using the port number specified in the call. It also sends packets that network drivers have received to the appropriate socket.

24.2.5. Network Device Drivers

The internet supports a variety of network hardware devices: DOMAIN ring controllers, T1 hardware, and MULTIBUS-compatible network devices. The AEGIS system provides drivers for the ring and T1 controllers; customers that wish to add their own network devices to the MULTIBUS backplane use the user-space general purpose I/O (GPIO) facility to create the network device driver, then make SVC calls to the routing software.

24.3. Sending a Packet on the Internet

A node in an internet sends a packet as follows:

- If the packet is addressed to a node on the local network the sender simply transmits the packet to the target node on the local network.

- If the packet is addressed to a node on a remote network, the sending node transmits the packet through the local network to a routing node. The sending node consults its routing table to determine the path to the destination node; it then sends the packet to the first routing node on that path.

- When the routing node gets the packet, it DMAs the packet into memory and transcieves the packet to the next node. The packet continues around the ring until it returns to the node that originated it; this node removes the packet from the ring and thus completes its role in the packet's transmission.

- Meanwhile, the routing node sends the packet to the routing node on the other side. This routing node examines the packet and either sends it to the target node, if it resides on this second network, or forwards it to another routing node, if the destination network is still remote.

24.3.1. Determining the Routing Node

A packet sent out on the internet has two target addresses:

- "Where next", which is a routing node ID

- "Where eventually", which is the destination node ID

The to node ID field in the packet's ring hardware header stores the node ID of the "next" node, while the IDP header stores the destination node ID and the node ID that originated it. The packet has reached its destination when the node ID in the packet hardware header matches the node ID in the software header (when "next" equals "eventual").

The process determines the next node to send to by consulting its RIP table (rip_$find_nexthop); the RIP handler returns the node ID to send the packet to and the port through which to send it (at this point, the port is the ring controller).

24.3.2. Sending a Packet through a Network Port

When it has obtained the routing node to send to, the process sends the packet through the network port returned by the RIP handler by calling net_$io_send. The network I/O manager uses the port descriptor to index into its network device table; it then reads the port device field to determine which network driver to call:

- If the network is a ring, it calls the ring device driver's send operation (ring_$sendp)

- If the network is an IIC bridge, it calls the IIC driver (iic_$sendp)

- If the network is a user-written network device, it queues the packet to that network port (which is a buffer queue) and increments the eventcount that corresponds to the port. The eventcount advance awakens the user network's transmit procedure to send the packet through the port.

24.3.3. Handling Incoming Packets on A Routing Node

The network I/O manager (net_io_$put_sock) on the routing node looks at the destination node ID of all incoming packets to determine whether the sender is sending a packet to be routed or is requesting information from the routing node itself. If the destination node ID matches the routing node's ID, then the packet is a request for some AEGIS service. In this case, the manager places the packet into the appropriate socket; for example, if the sender is requesting the routing node's build time, the manager places the packet into the information socket, which invokes the ASKNODE server.

If the packet's destination is some other node ID, the packet is a routing request and is consequently placed in the routing node's "through-traffic" queue, which is the routing socket.

24.3.4. Forwarding The Packet

When it gets a packet to route, the routing process increments the hop count in the packet header. The process then determines where to send the packet to next by consulting the RIP handler to find out what the next hop is, given the packet's destination network number and node ID (rip_$find_nexthop).

The RIP handler, by examining the target network ID, determines whether the packet should pass to another routing node, or to one of the networks to which the routing node is directly connected. (If the destination is not local or directly connected, the handler must lock the RIP table with a mutex lock to prevent any updates to the table while it is reading it.) The handler passes back the node ID and the port number to send the packet through to the routing process, which sends the packet out through the appropriate port (via net_io_$send). Packets destined for directly connected networks go to the network; packets destined for remote networks go to the next router.

If the routing process cannot find the next hop, it decides that the packet has been misrouted and simply drops it.

24.3.5. Maintaining Current Routing Information

The routing processes on routing nodes wait on a timer eventcount for RIP broadcast intervals; every 30 seconds, this eventcount reaches the broadcast value, and the routers call the RIP handler to issue RIP broadcast packets.

Each RIP broadcast packet contains:

- The standard packet header format
- The network numbers of target networks and the distances to them via this router, where distance equals the number of passages through routing nodes (hops)
- The node ID of the router broadcasting the RIP packet
- The network number of the port through which the RIP packet is going

The routing node builds the broadcast packet (rip_$broadcast), then calls the RIP handler to send it out through all of its routing ports (rip_$send_everywhere).

The RIP servers on the other nodes receive the RIP broadcast packets sent by routing nodes; RIP broadcast packets arrive in the node's RIP socket, where the RIP server unpackages them and adds the information to the routing table.

If the RIP handler does not hear from a routing node (via a RIP broadcast packet) within 90 seconds, it updates its RIP table to show that that router is unavailable.

The RIP handler also contains a module that periodically purges the the RIP tables of stale RIP data (rip_$age).

24.4. Internet Support for User Network Devices

Customers can connect their own network devices to the MULTIBUS backplane, but they must write their own drivers to run the device. Customers create user-written network drivers as follows:

- Write a user-space GPIO driver that contains a transmit procedure and a receive procedure to send and receive packets through the device

- Declare the driver a network by registering it in the network I/O manager's network port table; currently the manager reserves four port descriptors for user-written networks

- Call the routing software to transmit messages messages through the user network port

A port for a user network consists of a buffer queue that stores outgoing packets and an eventcount that advances when packets arrive at the port, activating the transmit portion of the network driver. Although the buffer queue is actually a socket, it does *not* accept incoming packets from the network; it only contains outgoing packets.

The routing process exports two functions to user-written network drivers:

- A route_$outgoing procedure, which sends packets to the user port buffer queue and awakens the transmit procedure by advancing the buffer queue's eventcount

- A route_$incoming procedure, which the driver's receive side calls to forward incoming packets to their destinations

Chapter 25
Introduction to System Initialization

System initialization is the collection of procedures that bring a node from power-on or reset to the display manager's login prompt. System initialization begins with the central processor's bootstrap PROM hardware and extends through a variety of software programs and system routines. In general, initialization proceeds as follows:

1. You turn the node on or press the RESET button

2. The bootstrap PROM initialization code begins to run.

3. If the node is a DNx60 series device, the PROM loads the microcode files, which include the writeable control store (WCS) file, the instruction decode file, and the scratchpad file.

4. The PROM loads and calls the system bootstrap utility SYSBOOT if there is a local disk, or calls the network bootstrap utility NETBOOT if booting over the network, to load and run the AEGIS system.

5. The AEGIS cold start routine sizes memory, loads the initial memory management unit (MMU) configuration, and calls the software system initialization procedure os_$init.

6. The os_$init routine completes the operating system initialization, including the initialization of the AEGIS managers, device drivers, other protected supervisor-mode system software. It then loads and passes control to the initial user-mode program, the bootshell (SHELL).

7. The bootshell displays the Apollo logo, then loads and runs the user environment initialization (ENV) program.

8. The ENV program initializes the first user process environment and loads and runs one of the following processes:

   - The display manager (DM)

   - The single-user shell (SH)

   - The server process panager (SPM)

Chapters in this section expand upon the following system initialization procedures:

- Bootstrap PROM initialization
- AEGIS bootstrapping from a disk, the network, or a cartridge tape
- AEGIS system initialization by the cold start and os_$init routines
- Establishment of the user environment by the ENV program

The chapters in this section do not describe display manager, server process manager, or the single-process shell initialization.

Chapter 26
The Bootstrap PROM

The bootstrap PROM, also called the mnemonic debugger (MD), serves several purposes. It contains the code that bootstraps a node, and also contains code to run a low-level debugger. In addition, it contains several low-level services that are available to external routines.

The PROM type varies according to node model. This chapter provides a generalized view of the PROM's system initialization functions and points out any significant differences among PROM types.

26.1. PROM Overview

System initialization begins when someone turns on a node's power or pushes the reset button. Either of these actions gives the PROM control from the M680x0 processor's power-on/reset vector.

The PROM operates in one of two modes: **normal mode** or **service mode**. The node service switch determines the mode in which the PROM will operate when it gains control. PROM system initialization differs significantly between normal mode and service mode. In normal mode, initialization is a *load and go* operation that proceeds without human intervention up to the point of login. In service mode, the PROM enters the mnemonic debugger command interpreter (MD), which enables the use of the mnemonic debugger to control further operations.

26.1.1. RAM Memory Use

The PROM uses two to three pages of RAM memory. On DNx60 models, these pages are located at addresses 200000 through 200BFF; they are located at addresses 100000 - 1007FF on the other node models.

The PROM's static data and supervisor stack area occupy the first one or two pages of this region. On DNx60 models, both the CPU and CPIO stacks also occupy this area. The last RAM page is reserved for the mapped-mode trap page, which is initially a copy of the PROM trap page (0-3FF).

26.1.2. Physical and Mapped Modes

The PROM runs in either physical or mapped mode. In **physical mode**, the memory management unit (MMU) is disabled. The PROM initially runs in physical mode. In **mapped mode**, the MMU is enabled and loaded with the PROM's mapping of memory and I/O devices. (the cold start routine enables the MMU; see Chapter 28.) When it begins to run, the PROM determines whether the MMU is enabled by examining a flag at location 0. In physical mode, this byte has the value 00. In mapped mode, the byte's value is FF.

For node models other than the DNx60, physical address space in mapped mode appears as follows:

- The PROM initial trap page (0-3FF physical) becomes inaccessible.

- The PROM static procedure and data section (400-3FFF) is mapped one-to-one so that mapped addresses correspond directly to physical addresses.

- I/O objects are mapped to their AEGIS locations (starting at FA0000).

- The PROM stack and variable data (100000 - 1003FF) is mapped to E00000.

- The mapped-mode trap page at 100400-1007FF is mapped to address 0.

- The rest of physical memory is mapped one-to-one to maintain direct correspondence between mapped and physical addresses.

The PROM can run in either physical or mapped mode because all references use variables; a PROM routine converts references to physical or virtual addresses. The service-mode MD commands P and M switch the PROM between physical and mapped mode.

The ability to run in physical or mapped mode is most useful when using the PROM's mnemonic debugging logic while the system is running. If you enter the PROM from AEGIS, for example through a crash, you should be able to use the PROM without disturbing the contents of the MMU. Therefore, the PROM code and the AEGIS system must use the same mapped locations for all addresses that the PROM will access.

PROMs on nodes with reverse-mapped MMUs must enter mapped mode in order to access the disk or the network. These PROMS perform disk and network I/O in mapped mode because all DMA goes through the I/O map (IOMAP), and, on older node models, the I/O map is inaccessible unless the MMU is enabled. The PROMS on forward-mapped MMUs perform disk and network I/O in physical mode, and do not enter mapped mode unless explicitly directed to do so.

26.1.3. PROM Functions

The PROM's main functional units are:

- System initialization logic for hardware that the PROM references, including serial input/output (SIO) lines, VME and MULTIBUS devices, memory, (clearing parity in addresses 100000 - 1007FF), and I/O devices.

- A set of minimal device drivers for the display, SIO devices, disks, the ring, cartridge tape, and the diagnostic LED display.

- Boot logic, including the logic to get SYSBOOT or NETBOOT into memory

- Power-up diagnostics to perform minimal hardware verification

- Logic to enter mapped or physical mode

- Mnemonic debugger (MD) commands

- Assembler/disassembler

26.2. PROM Structure

The PROM occupies physical address space 0-3FFF on DN400, DN420, DN600, DN300, and DN320 node models. It occupies physical addresses 0-7FFF on DNx60s. On DN550s, DSP80s, and machines using Motorola 68020 CPUs, there is additional PROM space at physical addresses 14000 - 17FFF. These addresses are mapped to 4000 - 7FFF on DN550s and 68020-based systems. This space is unused on DSP80s, and is mapped one-to-one on those units.

26.2.1. Initial Trap Page

Page 1 (0-3FF) of all PROMs is the initial trap page, which contains M680x0 exception vectors. This page is only used in physical mode. In mapped mode, the PROM replaces it with the copy in RAM memory, which enables the vector addresses to be changed. The mapped vector page is located at physical address 100400 on most machines. (It is located at address 200800 on DNx60 nodes.) . The processor hardware assigns the actual vectors to addresses 0-FF.

26.2.2. Machine ID

A machine ID exists at address 100 to identify the node model. Valid machine IDs are:

```
0 = DN400 old DN420
1 = DN420, DN600
2 = DN300, DN320, DN330
3 = DSP80, DSP90
4 = DN460, DN660, DSP160
5 = DN550, DN560
```

26.2.3. Auxiliary Information

PROM address 102 indicates auxiliary information. Currently, two bits within this address are used. The first bit (1), if set, indicates that log error and crash entry points exist. Bit 2, if set, indicates that the node uses an M68020-based processor board (DSP90, DN330, DN560).

26.2.4. Externally-Callable PROM Routines

Addresses 104 through 13F or 20F contain the addresses of several subroutines that can be called from outside the PROM. SYSBOOT, NETBOOT, and the AEGIS system crash and dump routines use subroutines. They are also available to any other stand-alone routine that does I/O to the network, disks, or SIO lines.

26.3. PROM Initialization Procedure

When the node is turned on or reset, the processor loads the stack pointer (SP) and program counter (PC) registers from addresses 0 and 4, respectively. The initial PC points to second page of the PROM. The SP points to the PROM's stack at 100200 (200400 on DNx60 models).

The PROM turns off the LEDs and tests the normal/service switch. to determine whether it should proceed in normal mode or service mode. Figure 26-1 illustrates the normal- and service-mode PROM initialization activities, and shows their interrelationships.

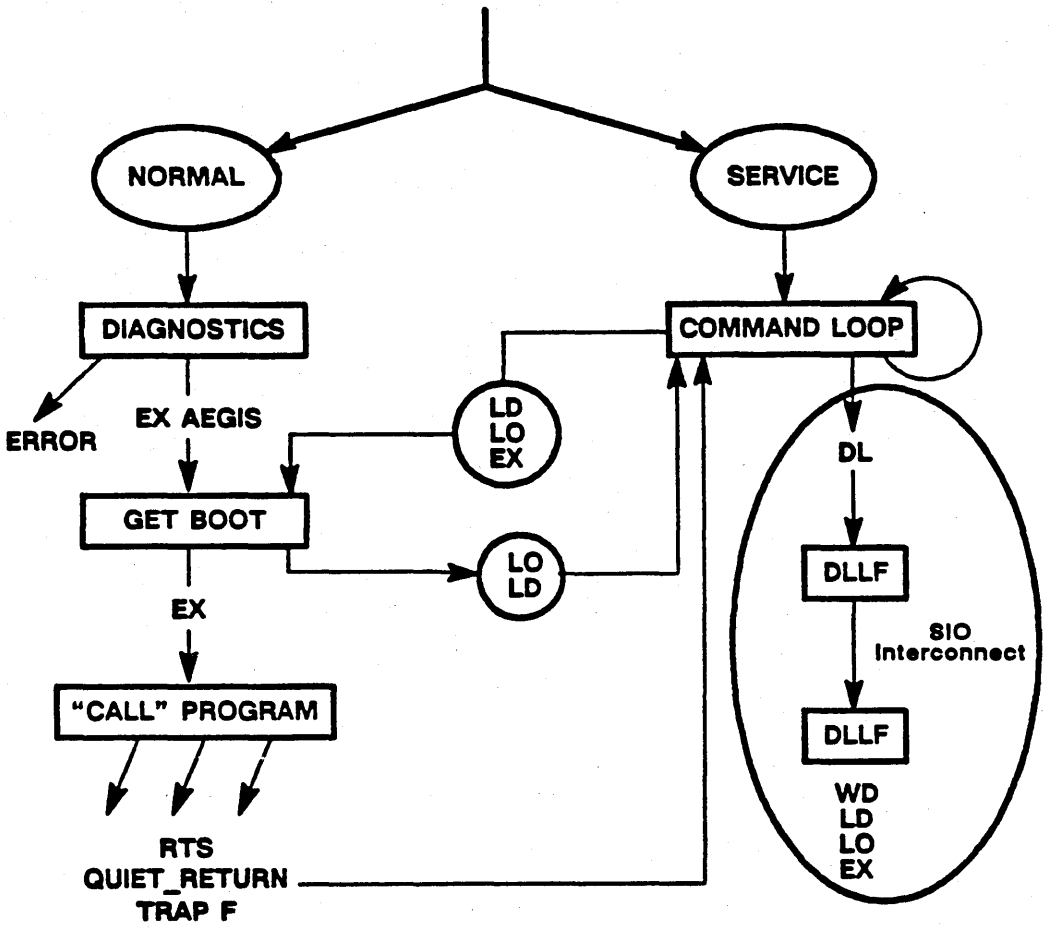


Figure 26-1. PROM Startup Activities in Normal and Service Modes

26.3.1. Normal-Mode Initialization

When the node is in normal mode, the PROM executes the following at startup:

1. Runs a series of diagnostic tests.

2. Loads the DNx60 microcode if it is initializing a DNx60 node.

3. Determines which system bootstap program is required (SYSBOOT, NETBOOT, or CTBOOT) and loads the appropriate program.

4. Checks the execution-requested flag to determine whether an EX command has been issued.

26.3.1.1. Diagnostic Testing

The PROM first runs a series of diagnostic tests. It indicates which diagnostic it is running by a message on the display (or SIO device) screen. All nodes except the DN550 use the message format:

```
Diagnostics: n
```

where n indicates the test completed. The DN550 uses the format.

```
Self Test Started n
```

The PROM carries out the following diagnostic tests (an asterisk indicates that the PROM does not perform the test on DNx60 nodes):

- Checks the PROM checksum
- Checks the PFT (pattern write and verify)(*)
- Checks the PTT (pattern write and verify)(*)
- Checks the I/O map (pattern write and verify)(*)
- Tests the PFT, PTT and I/O map interaction (*)
- Tests the RAM memory (pattern write and verify)(*)
- Tests the RAM memory in mapped mode (pattern write and verify)(*)
- Returns to physical mode and verifies the results of the mapped mode test (*)
- Test the DSP80 CPU extension board functions (DSP80 or DN550)
- Performs a single node transmit test on the ring board
- Verifies the integrity of the VME interface board (DN550 only)

26.3.1.2. Loading DNX60 Microcode

If the node is a DNx60, the PROM loads the the following microcode files:

- The writeable control store microcode file (/SAU4/WCS.UC)
- Instruction decode RAM contents (/DCODE.UC)

- Scratchpad constants and temporary varibles (/SAU4/SPAD.UC)

- A program that loads the above file into micro storage (/SAU4/ULOAD)

The PROM loads each microcode file by first loading SYSBOOT, NETBOOT, or CTBOOT and then using the boot program to load the file into physical memory. This procedure is identical to the procedure used to load the AEGIS system (which is detailed in the following steps). Once the microcode files exist in memory, ULOAD runs to load the microcode into micro storage, and then the microexec is started.

26.3.1.3. Determining the Bootstrap Program

The PROM next determines whether SYSBOOT, NETBOOT or CTBOOT is required by checking the controllers on the node. If a winchester or storage module device controller exists, the PROM will load SYSBOOT from disk. If only a ring controller exists, or if the attempt to load SYSBOOT failed, The PROM attempts to load NETBOOT from a partner node. The PROM will only attempt to load CTBOOT if you have set the disk type to C using the MD DI command

If SYSBOOT is required, the PROM reads the physical volume label to get the disk parameters (blocks per track, and so on) that describe the size and shape of the boot volume. (The PROM initially assumes one block per track and one track per cylinder.) The PROM then loads SYSBOOT from the first ten blocks on the disk (2 through B).

If NETBOOT is required, the PROM determines if it has a known partner node to boot from; if no known partner exists, the PROM broadcasts a packet requesting the NETMAN program on other nodes to respond if it is this node's partner node. When they receive the broadcast, all nodes in the network check their diskless lists to see if they are valid partners. The partner node's NETMAN sends a positive response. The PROM then requests the NETBOOT program from the partner node and loads it into local memory.

If CTBOOT is required, the PROM loads CTBOOT from the first file on the cartridge tape.

Once it has obtained and loaded the appropriate bootstrap program, the PROM calls it to load the AEGIS system. See Chapter 27 for more details about AEGIS initialization.

26.3.1.4. Checking the Execution Flag

When the boot program returns to the PROM, it determines if EX (execution requested) flag is true. If execution was not requested, the PROM goes into the MD command loop. If execution was requested (by an EX command) the PROM passes control to the loaded program. In normal mode, the EX command for AEGIS is internally generated, and AEGIS initialization proceeds automatically.

26.3.2. Service Mode Initialization

When the PROM determines that the node is in service mode, it branches to a command interpreter called the **mnemonic debugger** (MD). (Note that the entire PROM is often called the MD.) The mnemonic debugger provides a set of low-level commands and an assembler and disassembler. Whenever the system crashes it enters the MD. In service mode, you can enter the MD from AEGIS by holding down the CTRL key and pressing the RETURN key.

In service-mode system initialization, the MD tests the standard keyboard and SIO1 and SIO2 line inputs for a RETURN character. The first device to provide such a character becomes the input/output device for the debugger. The MD will then respond to any commands from the I/O device (including EX AEGIS to initialize the operating system).

You should not use SIO2 for the MD if you will be running AEGIS or any other stand-alone utility (SAU) from the mnemonic debugger. These programs only scan the standard display input and SIO1 lines, not SIO2. For example, if you put a dumb terminal on SIO2 and use it in service mode, and then enter the EX AEGIS command, the terminal will display the boot sequence through the HI: LOW: START: address messages. No further output is displayed, and AEGIS will not accept input from the terminal.

Chapter 27
SYSBOOT, NETBOOT, and CTBOOT

The system bootstrap program (SYSBOOT), the diskless node bootstrap program (NETBOOT), and the cartridge tape bootstrap program (CTBOOT) are bootstrapping mechanisms that the PROM loads and invokes directly to load stand-alone programs into memory. Stand-alone programs include diagnostics, stand-alone utilities such as SALVOL and INVOL, and the AEGIS system itself. All stand-alone programs are machine type-dependent; the boot programs load them into fixed physical locations in memory. Each stand-alone program to be loaded must reside in a SAUn directory (where "n" identifies the machine type) and must start with three integer values:

- A low address at which the program is loaded
- The start address, which is the initial execution entry point
- The machine type identifier of the target machine, or 0 if the program can run on any machine type.

The SYSBOOT and NETBOOT programs can access a disk SAUn directory and either list the directory contents or load a specified file into memory. SYSBOOT accesses the local boot disk, while NETBOOT accesses a partner node's disk across the network. CTBOOT can find and load a specified program from a cartridge tape that was written with the WBAK program, and can list the names of the files in a SAUn directory saved on the tape.

The PROM invokes the boot program and passes it three input parameters:

- The PROM's command buffer
- The boot logical volume number or network partner
- A set of option flags that provide informatio such as whether the service switch is in normal mode.

When the PROM is running initialization in normal mode, the boot program loads AEGIS into the node's memory in response to an EX AEGIS command that the PROM issues. When the PROM invokes the boot program in response to an MD command, the boot program either loads the specified file or lists the SAU directory.

The boot program returns three output parameters to the PROM:

- The starting address of the loaded program
- An execute or "go" flag. The boot program sets the go flag to true if it loads a program in response to an EX command; otherwise it is false. If the go flag is true, the PROM will invoke the loaded program.
- For NETBOOT only, a set of UIDs from the partner node. These UIDs are the OS paging file UID, the disk entry directory (/) UID. and the network root directory (//) UID.

27.1. The System Bootstrap Program (SYSBOOT)

SYSBOOT runs if the node has a local disk. It loads a file from the SAUn disk directory into memory, or lists the contents of the SAUn directory. SYSBOOT can also automatically execute SALVOL if the boot disk requires salvaging. However, SYSBOOT will only run SALVOL if it is operating in response to an EX AEGIS command in normal-mode initialization.

SYSBOOT is always located in physical disk blocks 2-B of a bootable disk. These are logical blocks 1 through A of the *first* logical volume on the disk. The initialize volume (INVOL) program reserves these blocks for SYSBOOT. The copy boot (CPBOOT) program puts SYSBOOT on the disk.

SYSBOOT performs the following operations:

1. Reads the disk physical volume label (using the PROM read disk routine) to get the disk address of the logical volume.

2. Reads the logical volume label for the logical volume specified by the PROM (This is logical volume 1 for normal mode startup).

3. Determines if SALVOL must be run. It checks two places in the logical volume label: 1) the BAT header volume trouble bit, and 2) the shutdown state word. If the volume trouble bit is set or the shutdown state word indicates that the disk is mounted (that is, the volume was never properly dismounted), then salvaging is required and SYSBOOT checks if the following are true:

   - It did not just attempt to salvage the disk

   - It is responding to an EX command for a file beginning with AE

   - The node is in normal mode.

   If all three conditions are true then SYSBOOT saves the name of the original file to be loaded, and changes the name of the file to be loaded to SALVOL.

4. Reads the logical volume's root directory (/).

5. Gets the UID of the SAUn directory from the root directory, where:

   - n = machine id if the machine id does not = 0

   - n = 1 if the machine id = 0

   - n = null if the machine ID is 0 or 1 and the SAU1 directory is not found.

6. Gets the VTOCX for the SAUn directory.

7. Reads the /SAUn directory.

8. If SYSBOOT is running in response to an EX or LO command:

   a. Gets the UID of the file to be loaded from the SAUn directory .

   b. Gets the file's VTOC index (VTOCX).

c. Uses the VTOCX to read the file's VTOCE, and uses the VTOCE to read the first record of the file into memory.

d. If SYSBOOT is running in response to an EX command, SYSBOOT checks to see if the program has the correct machine ID.

e. Reads the entire file into memory, starting at the location specified by the low_address value in the first word of the first record.

f. If salvaging is required as determined in step 3, does the following:

- executes SALVOL
- Changes the file name back to the original name of the file to be loaded.
- Returns to Step 1.

g. Writes to the output device the following addresses for the loaded file:

- Low: lowest physical address of the file
- High: highest physical address of the file
- Start: starting entry point of the program.

If the SYSBOOT is running in response to an MD LD command:

a. Writes "//LOCAL/SAUn x" (where x is the logical volume number) to the output device.

b. Lists the objects in the directory, 60 characters to a line.

9. Returns to the PROM.

27.2. The Diskless Node Bootstrap Program (NETBOOT)

NETBOOT loads a file or lists a directory from a disked partner node over the network. NETBOOT communicates with the user-mode AEGIS diskless node bootstrap procedure NETMAN on the partner node, and then requests services through NETMAN. This section describes NETBOOT operation on the local and diskless nodes and NETMAN and other process support on the partner node.

27.2.1. NETBOOT Functions

When NETBOOT runs, it first requests a response from a partner node. In response to a LO or EX command (normal-mode initialization automatically generates an EX AEGIS command) NETBOOT does the following:

1. In response to an EX command, sets the execute flag to true; otherwise it is false.
2. Sends a boot request to the partner node that requests it to return the file to be loaded.

3. Reads the reply packets that constitute the file and loads them into memory, checking to make sure that they will not overlay the boot program.

   As it loads each packet (a page), it writes a period (.) to the display.

   Each time it loads eight packets, it writes the total number of bytes loaded to the display.

4. When the it has finished loading the file, NETBOOT writes the following addresses for the loaded file to the output device:

   - Low address

   - High address

   - Start address

   If NETBOOT has run in response to an LO command, or the file is too small to be AEGIS, NETBOOT returns to the MD.

   Otherwise, it transmits a request for the UIDs of the following:

   - The OS paging file

   - The network root directory (//)

   - The disk entry directory (/)

In response to an LD command, NETBOOT first transmits a list directory request to the partner node. It then receives the returned messages, which contain a list of the files in the SAUn directory, and writes them to the terminal.

27.2.2. Partner Node Support of Diskless Nodes

NETMAN runs in the partner node and services various requests from the diskless node during startup. It supports both the PROM's loading of NETBOOT and NETBOOT's loading of AEGIS. In general, it services requests made through NETBOOT for the MD LO, LD, and EX commands.

NETMAN is the user-level diskless node bootstrap server. It runs in user mode and is accessed through a well-known socket. NETMAN provides three types of service: dump, echo, and boot. This document only covers the boot service support of system initialization and MD operations.

NETMAN responds to a boot service request by passing the request to the boot request processor, boot_$serv. Table 27-1 lists the kinds of boot_$serv procedure services. In addition, the following paragraphs describe the boot_$get_uids service in detail.

Table 27-1. BOOT_$SERV Services

Service	Description	Requesting Software/Hardware
boot_$echo	Not used for initialization	
boot_$ld	Old list directory request, returns only a single packet	NETBOOT, in response to the MD PL command.
boot_$mult_ld	List directory, sends multiple packets to list SAUn directory	NETBOOT, in response to the MD LD command.
boot_$load	Sends the requested file from the SAUn directory	NETBOOT, in response to the MD EX and LO commands, including normal AEGIS initialization.
boot_$sysboot	Sends the NETBOOT file from the SAUn directory	Diskless partner's PROM before it can process any MD LO, EX, or LD request, including normal mode startup.
boot_$volun	Checks DISKLESS_LIST file to determine if this node is a partner to the requesting node and sends a confirmation, including the node-id, if true	Broadcast by diskless node's PROM as part of normal mode startup and in response to any LO, EX or LD command if the partner is not yet known.
boot_$get_uids	Sends the UIDs for the OS paging file, network root, and the node entry directory; it also creates any required supporting files	NETBOOT, following a boot_$load request, in response to an MD MD EX command for any file long enough to be AEGIS, including during normal mode start-up).

27.2.3. Get UIDs Service

In addition to sending the requested UIDs to the diskless node, NETMAN's boot service routine boot_$get_uids carries out all the initialization and housekeeping required to enable the node to support the diskless node. The get UIDs routine:

1. Resolves network root UID.

2. Resolves the node entry directory UID.

3. Resolves the /sys/node_data.node_id directory, where node_id is the ID of the diskless node, and unlocks it.

   If the directory does not exist:

   - Creates the directory.

   - Resolves the new directory's UID.

   - Sets the default ACLs.

4. Does the following to prepare the OS paging file (/sys/node_data.node_id/os_paging_file):

   - Resolves the file UID.

   - If the file does not exist, creates it.

   - Checks the file length, and if it is less than 320 blocks, extends it.

5. Does the following to prepare the bootshell file (/sys/node_data.node_id/shell): (note that the boot shell is an impure object, mapped for read and write, and therefore the diskless node needs its own copy.)

   - Resolves the file UID.

   - If the file exists, unlocks it and deletes it.

   - Creates the file.

   - Copies /sys/node_data/shell to the file.

6. Copies the /sys/sysdev directory to /sys/node_data.node_id/sysdev to give the remote node its own system devices.

7. Copies the startup templates as follows:

   - If the remote node is a DN300 or DN320, it copies the DM startup file templates to /sys/node_data.node_id/startup.191 and makes sure the file contains a KBD 2 command.

   - If the remote node is a DSP80 (and therefore does not have a display), it copies the server process manager (SPM) startup file template directory to /sys/node_data.node_id/startup_templates.

   - Otherwise, it copies the DM startup template directory to /sys/node_data.node_id/startup_templates.

8. Sends a reply to the diskless node with the required UIDs.

27.3. The Cartridge Tape Bootstrap Program (CTBOOT)

The CTBOOT boot program enables a node to boot from a cartridge tape. For example, it enables a partnerless or stand-alone DN550 to boot and load software on a new Winchester disk. Like SYSBOOT and NETBOOT, CTBOOT enables the PROM to load and run stand-alone programs and to list the contents of the SAUn directory on the tape. CTBOOT will only run in response to an EX, LO, or LD command passed to it by the PROM.

The PROM can access a tape's contents only if CTBOOT is at the start of the tape. The CPBOOT utility puts the CTBOOT at the start of a cartridge tape. The remaining files on the tape must be written in read_backup (RBAK) format, for example, using the write_backup utility (WBAK).

To run CTBOOT, you must first specify the cartridge tape as the boot device. To do so, enter the MD and enter the following command line:

DI C

You can then enter an MD command to list the SAUn directory from tape or to load or run any stand-alone program on the WBAKed tape, including AEGIS or INVOL. When the PROM requires the boot program, for example in order to invoke the EX AEGIS command, it reads the first file from tape. The PROM then invokes this file identically to SYSBOOT to load the requested file or list the tape contents.

CTBOOT performs the following operations:

1. Sets up the name of the SAU directory as SAUn, where n is the machine ID.

2. Parses and saves the command line passed to it by the PROM.

3. Scans the first logical file on the tape to find the directory name //.../SAUn. Note that that any directory names to the left of SAUn are ignored, so the SAUn directory can be located at any depth in the directory tree structure.

4. If CTBOOT is running in response to an EX or LO command:

   a. If CTBOOT is running in response to an EX command, checks to see if the program has the correct machine ID.

   b. Reads the entire file into memory, starting at the location specified by the low_address value in the first word of the first record.

   c. Writes to the output device the following addresses for the loaded file:

      - Low: lowest physical address of the file

      - high: highest physical address of the file

      - Start: starting entry point of the program.

   If the CTBOOT is running in response to an MD LD command:

   a. Writes the directory pathname to the output device.

   b. Lists the objects in the directory, 60 characters to a line.

5. Rewinds the tape.

6. Returns to the PROM.

Chapter 28
AEGIS Initialization

After SYSBOOT, NETBOOT or CTBOOT loads AEGIS, the PROM passes control to the newly loaded program, and OS initialization begins. Two procedures within the AEGIS kernel -- the **cold start routine** (COLD_START) and the os_$init routine -- are responsible for AEGIS initialization. The cold start routine contains the initial entry point for AEGIS and is given control directly from the PROM. When it completes its operations the cold start routine calls the os_$init routine.

28.1. The Cold Start Routine

The cold start routine occupies a single page of memory starting at physical address 100800. The data space for the cold start routine and its associated data occupies one page starting at physical address 101000; these pages are mapped one-to-one in virtual memory. The os_$init routine frees this memory after cold initialization completes and AEGIS is running.

The cold start routine carries out the following operations:

1. Reads the following machine definition arguments passed by the PROM:

   - The boot device type (network, winchester, etc.), unit number, and logical volume number

   - The node ID

   - If the node is running diskless: the UIDs of the OS paging file, network root directory, and node directory.

2. If the node is a DN300, DN320, DSP80, or DN550, determines whether the processor is a 68010 or 68020. If it is a 68020, enables the on-chip instruction cache and relocates the following in virtual memory:

   - AEGIS from E00000 to 3D00000

   - The page translation table from 700000 to 400000

   - The i/o definitions from FA0000 to 3FA0000.

3. Copies the trap page from physical address 0 - 400 to physical addresses 100400 - 100800.

4. Initializes the MMU in mapped mode with memory mapped one-to-one. If the node is a DNx60, the routine does not initialize the MMU, but instead relocates the trap vectors using the vector base register (VBR).

5. Scans (reads and rewrites) physical memory to initialize the memory map (MMAP) (by marking free and missing pages) and to eliminate ECCC or parity errors.

6. If the node has a VME bus, sets up the VME memory size register. This register informs the VME interface which addresses refer to memory and which should be passed to the VME bus.

7. If the node is a DN550 or DSP80 with a MULTIBUS, then enables the MULTIBUS. Enables the second second half of the MULTIBUS only if physical memory does not exist at address 380000.

8. Enables the MMU with the initial virtual to physical address mappings. (The virtual and physical address spaces are defined in detail in Appendix B.)

9. If this node has a 68020 with 68221 coprocessor, enables the coprocessor; if the node has a 68020 but no coprocessor, enables the floating point emulator.

10. Turns on memory ECCC or parity operations.

11. Calls the os_$init routine to complete the AEGIS initialization process.

28.2. The OS_$INIT Routine

The os_$init routine completes the supervisor-level initialization of AEGIS. When it completes its operations, it calls the bootshell. The os_$init routine does the following:

1. If the node has a 68020 processor, calls the routine mmu_$init to adjust the MMU operations for the larger PTT and as_$init to adjust the address space.

2. Initializes the trap page with the addresses of the appropriate fault interceptor module entry points.

3. Removes the virtual-to-physical association between the cold start routine and associated data pages from the MMU. This frees the physical memory so that the the memory map manager, when initialized, can record these pages as available.

4. Initializes the managers listed in table 28-1, as required by the node configuration. The table lists the initialization call, its function and additional information that describes the operation. To track the structure of the os_$init procedure, this table also describes some operations that do not strictly initialize managers but are included within this section of the code.

Table 28-1. Managers Initialised by OS_$INIT

Procedure	Manager	Comments
MMAP_$INIT	memory map	Required by dbuf, pbu_$init, makes a list of present but unused pages.
PBU_$INIT	peripheral bus unit	Must precede IO_$INIT
IO_$INIT	Winchester, Floppy, Ring xmit, Ring rcv, Cart tape, Stor mod	For storage module, only reserves I/O map pages, does not wire the procedure.
LPR_$INIT	Parallel line printer	
PEB_$INIT	PEB	(Floating point accelerator module)
MEM_$INIT	Memory ECCC logic	Must be done inhibited
TERM_$INIT	Dumb terminal driver	Initializes all four SIO lines
TIME_$INIT	Timer	Crashes if should have calendar on disk controller and do not
UID_$INIT	UID Generator	Needs the node number and time
none		Sets up the null process stack
PROC1_$INIT	level 1 process mgr.	Caller becomes process 1
SMD_$INIT	screen mgr. driver	Inits the display driver
TPAD_$INIT	touchpad data mgr.	Inits. touchpad mode settings/mouse
DTTY_$INIT	dumb TTY handler	Can be initialized for display or for a real dumb terminal
EC2_$INIT_S	level 2 eventcounts	Inits. freelist of wait list entries
PRINT_BUILD_TIME		prints the system build time
none	Parity	Sets parity error trap to its fim
none	DNx60 floating pt.	Sets floating point traps to fims
DBUF_$INIT	Disk buffer manager	Inits. disk buffers used by bat/vtoc
SM_$CINIT	Storage module controller	Only if SM is boot device; do not wire. Call is in an internal proc.

5. If the node is not diskless, mounts the boot volume as follows:

   a. Mounts the boot physical volume, initializing the volume table entry with information from the physical volume label.

   b. Gets the UID of the first logical volume on the drive.

   c. Mounts this logical volume.

   d. Verifies the clock time against the boot volume shut_down time. If the shutdown time is later than the clock time or is over three days before the clock time, asks if you wish to update clock. This is done by the cal_$verify procedure of the os_cal_unwired module.

   e. Mounts the VTOC. If salvaging is required, issues a message to the display.

      If the node is in normal mode, crashes the system.

      If the node is in service mode, it displays the message: " Proceed to bring up OS (and risk volume)? If the response is yes, it mounts the VTOC; otherwise, it crashes the system.

6. Initializes the virtual memory managers. This is done by the mapped segment table initialization (mst_$init) procedure, which operates as follows:

   a. Initializes the access control list manager (ACL) using acl_$init.

   b. Initializes the active segment table manager and AST (ast_$init) using ast_$init.

   c. Initializes the address space ID (ASID) allocation list (for ASIDS 1 -25).

   d. Allocates ASID 0, which maps global address space.

   e. Sets up the MST for ASID 0.

   f. On Dnx60 nodes, sets up the segment map tables (SMAPS) and region registers.

   g. Sets the MST wired page limits, the maximum allowable number of wired MST pages. This is determined by a ratio of one page of MST entries per ten pages of pageable physical memory, so that approximately 200 megabytes of virtual memory can be mapped in one megabyte of physical memory.

7. Establishes the OS mappings to the OS paging file and initializes virtual memory mechanisms as follows:

   - Maps the segments required for AEGIS by running the mst_$maps procedure for each of the following areas, in turn. Each area is mapped to offset 0 in the OS paging file:

     a. The non-pageable OS procedures and data

     b. The pageable OS procedures anc' initialized data

     c. The non-pageable whole cloth pages to be used for page-aligned data such as the file lock table, ACL lock data tables, and PROC2 database.

     It is possible to map three different areas of virtual address space starting at the same location in the OS paging file because the two wired areas will never require space in the file. Thus, this technique reserves the virtual address space used by the wired procedures and wired pre-aligned data. It ensures that the mapped segment manager will not map the anything over them, yet it does not waste file space that will never be used.

   - Wires the AEGIS procedure section pages that must always remain in memory by setting their memory map entry .inuse bits as false (not in use). This eliminates the use of backing store.

   - Activates (adds to the AST) and associates the pageable AEGIS procedure pages and the initialized data areas.

   - Wires the non-pageable whole cloth uninitialized area pages by setting their memory map entry .inuse bits as false (not in use). This eliminates the use of backing store.

   - Frees any unused memory map pages, such as those that would be used for a nonexistent second display or MULTIBUS device and pages that mmap_$init has determined to be unused. This procedure both removes the page association from the MMU (mmu_$remove) and frees the page in the memory map (mmap_$free).

8. If the storage module is the boot volume, wires the SM manager.

9. Maps and activates the display memory and the memory for a second display, if it exists.

10. Creates the following special level 1 processes:

    - The clock process (PID 3)

    - The purifier process (PID 4)

    - The wired and unwired DXM process

11. Makes the current process (PID 1) a level 2 process by allocating an address space ID (ASID 1) and assigning it to the process, and by sets the process type to the initial system process.

12. Starts the network packet receive process (PID 6), using the network_$init procedure. This procedure also starts the first network paging server process (PID 7) and the first network request server (PID 8).

13. If this node does not have a clock, gets the current time from the partner node.

14. Initializes the lock manager database, using the file_$lock_init procedure.

15. Initialize the naming server, using the name_$init procedure.

16. Checks that the PROM node number and the boot volume node number are identical. If they are not the same and the node is in service mode, writes a message asking if you want to proceed. If the answer is not yes, or the IDs do not match and the node is in normal mode, initiates os shutdown.

17. Makes the disk entry directory (/) the working directory.

18. Initializes the PROC2 manager using proc2_$init, which does the following:

    - Initializes the PROC2 UID array to newly generated UIDs.

    - Initializes the PROC2 information record that describes process 1 (this process). The initialization operation creates and maps the 'node_data/stack object, which is the stack object for this process.

    - Checks if AEGIS is being booted from tape. If so, runs the tape_$boot procedure to find and load /bscom/rbak_shell (the cartridge tape bootshell) and returns.

      Otherwise, resolves the bootshell 'node_data/shell, locks it for cowriters, maps it, and returns with the bootshell start address.

19. Initializes the UID location hint manager using hint_$init.

20. Initializes the event logging manger using log_$init.

21. If the node has a color display, initializes the color debug module using color_$cold_init.

22. If the node is diskless, gets the calendar information from the partner.

23. Makes the trap page read-only.

24. Recalculates the maximum number of pages allowed for the mapped segment table to reflect the pages that have been freed.

25. If the node has a PEB (performance enhancement board), loads the PEB writable control store.

    If the node has a 68881 coprocessor, allocates and maps an area in which per-process 68881 state information will be saved.

26. If the node is diskless, initializes the MST so that it can print the PARTNER NOT RESPONDING message, if necessary.

27. Frees the pages used by the initialization code. the os_$init routine and all other initialization code called only during system initialization are located in separate named sections that are loaded at the the end ot the OS wired area. They have therefore been wired up to this point. However, they will never be needed again and can now be freed. As a result, the length of the paging file does not have to reflect the length of the initialization logic.

28. The os_$init routine exits by calling proc2_$startup using the bootshell address returned by the proc2_$init routine.

29. The proc2_$startup routine changes mode to user and invokes the bootshell.

Chapter 29
User Mode Initialization

The last task in system initialization is the initialization of the user interface. This process is completed by three modules:

- The bootshell (SHELL)
- The user environment initialization program (ENV)
- The normal user interface process; DM, SH, or SPM.

The bootshell is a transition point between AEGIS initialization and the definition of the user environment. The bootshell provides a variety of commands that are available when the node is in service mode, including debugging, virtual memory, and file system management commands.

The user environment initialization program (ENV) initializes the first process user environment and loads the display manager, the server process manager, or the shell program. It sets up the address space used by the first user-mode process, which is typically the DM.

The display manager (DM) is the normal user interface. The DM can only be used on nodes with a full bit-mapped display. It is the DM that presents the normal login message.

The server process manager (SPM) runs on server nodes, which are usually DSP80 and DSP160 systems without display interfaces. The SPM manages the node's response to create remote process (CRP) commands from other nodes. It also handles certain houskeeping functions, such as starting the MBX helper and monitoring the shutdown switch.

The shell program (SH) is the single-process shell. It can provide a user interface over an SIO line, where the full display capabilities are not available. This program is the same as the shell program invoked by the DM SH command.

29.1. The Bootshell

Like AEGIS, the bootshell is a program that has been processed by the run file converter ("an RFCed program"). The executable file is located in 'NODE_DATA/SHELL. A second version of the bootshell, RBAK_SHELL, is loaded and used when the node boots from a cartridge tape. This version can manipulate and read the tape, and provides an additional command, RBAK, which reloads the system software from the cartridge tape.

Except when running the DM, RBAK, SPM, SH or GO command, the bootshell runs as user.none.none, and therefore its access to objects can be limited by ACL restrictions. As a result, it is in some ways a vestigial resting point. It is useful for such operations as debugging library-level code.

29.1.1. Bootshell Initialization Operations

In a normal-mode system initialization, the bootshell carries out some preparatory operations and then loads and calls ENV to set up the initial user process environment. In service-mode initialization, the bootshell enters a command processor and displays the bootshell prompt. You can then enter any of the bootshell commands.

In a normal mode initialization, the bootshell:

- Displays the Apollo logo (/SYS/APOLLO_LOGO).

- Invokes the bootshell command file located at 'NODE_DATA/STARTUP_SHELL, if one exists. (This file is not required.)

- Determines which program should be invoked by ENV. If the node has a display, DM (/SYS/DM/DM) will be invoked; otherwise, SPM (/SYS/SPM/SPM) will be invoked. The single process shell (/SYS/BOOT) can only be invoked explicitly by entering the SH command when in the bootshell.

- Loads and runs ENV (/SYS/ENV)

If the node is being initialized from cartridge tape (that is, the bootshell is RBAK_SHELL), the bootshell automatically runs its RBAK procedure. This procedure prints a message that asks if you wish to replace the system software on your disk. If your answer is yes, it copies the first WBAK file from the cartridge tape to the system disk and returns to the bootshell command processor described below.

29.1.2. Bootshell Commands

The bootshell provides a command processor that appears automatically during initialization in service mode (that is, as a result of an MD EX AEGIS command). You can enter the bootshell command processor in normal mode by exiting the last user process as follows:

- Entering the EX command to the DM

- Pressing the CTRL/Z key combination when in the single process shell, but only if it was invoked by an SH command to the bootshell

- Entering the Quit command or pressing the CTRL/Z key combination to the SPM.

In these cases, exiting the program returns the process to ENV, which returns to the bootshell.

Bootshell commands can be divided into two categories: standard commands and BSCOM commands.

The standard bootshell commands are listed in Table 29-1. In addition, the bootshell commands include copies of the the MD debugging commands, including the assembler/disassembler. The debugging commands, like the other BS commands, are part of the bootshell code, and do not rely on the PROM.

The BSCOM commands are located in the /BSCOM directory. You invoke them by using the bootshell LO command. The BSCOM commands are:

- CPT.BS -- Copy Tree
- CPBOOT.BS -- Copy Boot
- DLT.BS -- Delete Tree
- LAS.BS -- List Address Space

These commands are similar, but not identical, in operation to their DM counterparts. For example, the CPT and DLT commands do not take arguments, but initiate an interpreter that requests input; they also provide online Help. Like most other bootshell commands, the BSCOM facilities are limited by the fact that the bootshell executes as USER.none.none.

The BSCOM directory has two additional files, LIB.BS and RBAK_SHELL. LIB.BS is the library used by the BSCOM commands. RBAK_SHELL is a copy of the bootshell that is loaded when AEGIS is booted from a cartridge tape. It is located in the BSCOM directory for distribution purposes only.

29.2. The User Environment Initialization Program (ENV)

The ENV program initializes the first process user environment, including the user global space that is shared by all user processes. It then loads the DM, SPM, or the single process shell (SH). ENV is bound to a private copy of the process manager (PM) and uses the manager's pm_$init_first procedure to carry out nearly all the initialization operations. The ENV program deletes the 'NODE_DATA/STREAM_$SFCBS files and calls the pm_$init_first routine.

The pm_$init_first routine performs the initializations that are unique to the first user-level process. It initializes user-level managers and user global space, including the global libraries and known global table, in addition to performing process-specific tasks. See Chapter 9 for an explanation of user-global and process-private address space.

Table 29-1. Bootshell Command Summary

CF	[<pathname> | -E]	run/end command file
CHN	<pathname> <compname>	change name
CRD	<pathname>	create directory
CRF	<pathname>	create file
CRL	<pathname> <linkname>	add link
CTNODE	<leaf> <node_id>	add node to local copy of root
CTOB	<pathname> <uidhi> <uidlo>	catalog name with specified uid
DEBUG	<value>	enable/disable debug mode
DLF	<pathname>	delete file
DLL	<linkname>	drop linkname
DM		load display manager
DMTVOL	{W|S|F} <lvno> [<pathname>]	dismount a logical volume
GLOB		list installed globals
GO		load as if in normal mode
H		prints help text
IN	<pathname> [-D] [-S | -NS]	invoke loader to install named file
LD	[<pathname>] [-A [-D]] [-U]	list directory
LI	<address>	set display lites address
LO	<pathname> [-D] [-S | -NS]	invoke the loader w/ the given RFC file
MA	<pathname> [<l> <sz>] [-E]	map file
MTVOL	{W|S|F} <lvno> [<pathname>]	mount a logical volume
ND	<pathname>	set naming directory
RBAK		Restore contents of a cartridge tape *
REL	[-A]	release proc-mgr assigned storage
SH		load single process shell
SHUT		shutdown the system
SPM		load the server process manager
STCODE	<status-code>	print textual definition of status code
TB		stack trace back
TI	{-ON | -OFF}	enable/disable the timer
TR	<pathname> <sz>	truncate raw data file to given size (hex)
UCTNODE	<leaf>	drop node from local copy of root
UCTOB	<pathname>	un-catalog pathname from namespace
UMA	{<pathname> | <l> <sz>}	unmap file by name or addr/size
WD	<pathname>	set working directory

Key:	<l>	:=	low address
	<h>	:=	high address
	<s>	:=	start address
	<sz>	:=	size
	<type>	:=	{ nil, rec, hdru, obj, dev, pad, undef, uasc, mt, boot }

Note * - The RBAK command is only provided by RBAK_SHELL, the cartridge tape bootshell, and can be executed only if AEGIS was booted from a cartridge tape.

The pm_$init_first routine does the following:

1. Initializes the user global space read/write storage (RWS) by creating a backing file for the storage ('node_data/global_data). If such a file already exists, it is deleted first. The new file is made permanent and locked for multiple readers or one writer (nr_xor_1w).

2. Maps and allocates the global space storage used by the global information record at the base of user global address space.

3. Initializes the user-private read/write space manager to use the private impure data area.

4. Allocates the space in the private read/write space for the process fault manager error mask.

5. Builds private read/write scratch space to be used in building the global space KGT to check for duplicate entries. In doing so, it resolves the process 1 stack object pathname ('node_data/stack) and initializes the read/write storage manager.

6. Initializes the global-space known global table (KGT) manager. This procedure allocates the space required by the pure (read only) KGT.

7. Defines entries in the pure KGT for the PFM and FPU masks and for various creation record variables.

8. Allocates global space for, and defines a pointer to, the first entry in the global library section information list. Defines an entry in the pure KGT that points to the list.

9. Builds a dummy impure KGT by determining the KGT's address and setting the number of entries, free entries, and maximum entries to 0.

10. Directs the AEGIS kernel to pool stack objects when processes are deleted.

11. Determines the node machine type ID and the type of floating point hardware, if any. (These are used by the library installation code, to determine whether the PEB library must be loaded.)

12. Installs the global libraries in global space. To do so it first sets the working directory to /lib. It then loads each of the libraries specified by pm_$libnames and adds their defined globals to the pure KGT. Note that each global library is either required or optional. If an optional library's name or UID can not be found, it is not installed and no error occurs.

    Also note that the syslib file in the /lib directory should match the type of PEB or floating point hardware that is installed on the node. There are five versions of syslib:

    ```
    syslib      --  for systems with no floating point hardware
    syslib.peb  --  for systems with a Performance Enhancement Board
    syslib.460  --  for DNx60 systems
    syslib.881  --  for systems with the Motorola 68881 coprocessor
    syslib.020  --  for systems with the M68020 processor
    ```

The pm_$init_first routine attempts to load the correct library, based upon the machine type ID and the PEB kind ID. If it fails to find the correct library then it writes an error message and tries to install the syslib library.

13. Completes the building of the global KGT, and truncates and unmaps the scratch area (in private Read/write space) that was used to build it.

14. Searches the KGT for the the addresses of externals required by the pm_$init routine.

15. Builds a creation record for the user-space boot program. To do so it:

    - Resolves, maps, and locks (for cowriters) the ACL cache ('node_data/acl_cache) that the DOMAIN/IX environment uses.

    - Resolves the user-level boot file pathname passed to it by the bootshell (/sys/dm/dm, /sys/spm/spm, or /sys/boot).

    - Assigns various creation record variables

16. Initializes the following by calling the appropriate PM initialization procedure:

    - The global static data in the process manager library.

    - The streams manager

    - The C library

17. Resolves the private address space user library (/lib/userlib.private), if any, and sets an indicator in the global information record if it is found.

18. Determines the high address of the private read/write storage space and saves it in the global information record.

19. Makes the global data section read-only.

20. Calls the process initialization procedure pm_$init to initialize this level 2 process and invoke the user-level boot program (DM, SPM or SH). (The pm_$init procedure is called whenever a new level 2 process is initialized; because it is a user-space routine, it is not covered in detail in this document.)

When system initialization completes, the address space of the first level 2 process will appear as it does in Figure 9-1 and Figure 9-2 in Chapter 9. When a user logs in, the login procedure maps the working and and naming directories to supervisor private addresss space.

Appendix A
Boot LED Codes

Some nodes (and probably all future ones) have four or more LEDs on the CPU board and/or visible from the front panel. MD and AEGIS use these lights to indicate the status of the system. They are particularly useful on server nodes (like the DSP80, where they were first introduced) that do not have a display or other means of communicating with a user.

A complete listing of the various LED codes can be found in the Engineering Handbook. This appendix only covers the codes that can appear during initial power up and reset.

A set of steady state codes or unchanging values are loaded into the LEDS as the PROM is initializing the system from power-up or a reset. If the system hangs before running the diagnostics or printing a prompt character on the display (and dumb terminal), the code in the LEDs will indicate that last operation that MD has successfully performed.

Note: if the system hangs, you should try a reset while watching the LEDs. Don't worry if you don't notice all values; some go by very quickly. Note also that one value -- F -- has two meanings.

F - Power-on value

> When a machine is turned on, the initial state of the LEDS will be F (all on). They will also show F as long as the reset switch is depressed.

0 - First instruction at "init"

> The very first instruction executed by the PROM (i.e., the one pointed to by location 4) sets the LEDS to 0. If this fails to happen, there is probably no clock signal to the CPU. Alternatively, the physical location of the LEDS is wrong, and you are getting an immediate bus error.

1 - Memory passed tests at init

> After turning off the LEDs, MD pushes and pops various data patterns to and from the stack to provide a minimal existence test for a path to real memory. If you don't get to this point, there is something wrong with memory or the address bus.

2 - State saved

> At this point MD has successfully saved the registers, determined whether it is running physical or mapped, and established the appropriate stack base register. If you don't get here, there may be a problem with the stack operations BSR/RTS.

3 - Parity cleared in first 2 pages

> MD next clears out any bad parity in the first two pages of memory (MD's data page and the mapped trap page. A hang here could indicate the the parity logic is ignoring the fact that it hasn't been enabled yet.

(The next few codes are loaded in the init_sys routine, which initializes the I/O devices used by MD.}

4 - VME address modifiers set, ring_id loaded, SIOs initialized, speaker off

> MD is prepared to accept bus errors referencing the ring or the VME interface, but not the SIOs. A hang before this value is shown could indicate a misbehaving VME or ring board, a non-responding SIO chip, or an error in the bus error handling logic.

5 - IOMAP cleared, multibus initialized, display (if any) cleared

> Again, bus errors on the IOMAP and multibus are tolerated, as it a bus error accessing the display. But if a display is present but not working right, MD may hang trying to clear it.

6 - Disp_init called, returned

> This value indicates that a display is present, and it was initialized without error.

7 - Display init got bus error

> This value indicates that the display initialization logic got a bus error accessing the display.

(This is the end of the init_sys routine. You should now see a prompt character (>) on the display and/or dumb terminal.)

8 - We're in service mode, waiting for keyboard input

> This value is displayed when MD determines that it is running in service mode (or on a reset command in normal mode). If the system is running correctly, this may be first really observable LED value. At this point, MD is waiting for you to enter a return or two from the input device you are using (display keyboard or dumb terminal).

9 - Character received from keyboard

A - Character received from line 1

B - Character received from line 2

> These values are displayed when a character has been received from the indicated device. If you do not get one of these responses to a carriage return, check the keyboard (unplug and plug in), check your SIO cabling, suspect a bad SIO chip or associated hardware.

C - Just printed MD's banner message

> As soon as MD has determined the input device, it prints its banner ("MD REV...") and displays this value. (You probably won't see the LED value, since MD immediately enters the command loop -- see below). If you don't see the banner, check the display/terminal hardware, brightness level, display memory functioning.

D - PTT enabled (map routine)

E - MMU initialized (map routine)

F - MMU enabled (map routine)

> These 3 values are displayed by the map routine, which
> is used whenever mapped mode is to be entered -- the 'M' command,
> in certain diagnostics, or, on some models, when disk or ring
> I/O is to be performed. "D" indicates that the PTT was just enabled
> into the address space. If you don't get this value, there is a problem
> enabling the PTT. "E" indicates that the MMU has been completely loaded
> with MD's mapping. "F" indicates that the MMU was actually enabled.
> (If you entered the "M" command, the "F" will immediately be replaced

After MD enters the command loop, starts running diagnostics, or initiates program execution, it uses a short-long two-digit sequence to indicate what's occurring.

Appendix B
Address Space

The following figures show the memory layouts of the various nodes that are presently available. These figures show both the physical layout of memory and the mapped memory address assignments.

Figure B-1. Physical Memory Layout

PPN	ADDRESS	APOLLO 1 DN420/600	APOLLO 2 DN3XX	APOLLO 3 DSP80/90	APOLLO 4 DNX60	APOLLO 5 DN5XX
0-F	0	PROM	PROM	PROM	PROM	PROM
10-1F	4000	pft	pft	pft	PROM2	pft
20	8000	MMU	MMU	MMU	MMU	MMU
21	8400	sio	sio	sio	sio	sio
22	8800	tmrs.cal	timers	timers	tmrs.cal	tmrs
23	8C00	floppy	-	-	floppy	-
24	9000	-	dma	dma	iomap	-
25	9400	-	disp1	-	-	chameleon
26	9800	-	ring	ring	-	ring
27	9C00	-	disks.cal	-	-	dsktape.cal
28	A000	-	-	calendar	-	manf't disk
29	A400	-	-	pbu ctl	-	pbu ctl
2A	A800	-	-	lpr	-	-
2B	AC00	diags	diags	diags	diags	diags
2C	B000	peb ctl	fpu ctl	fpu ctl	useqncer	fpu ctl
2D	B400	fpu cmd	fpu cmd	fpu cmd	dcode ram	fpu cmd
2E	B800	ring1	fpu cs	fpu cs	ring1	fpu cs
2F	BC00	ring2	-	-	ring2	vme ctl
30	C000	fpu cs	-	-	wcs ctl	uvme0
31	C400	cache w0	-	-	-	uvme1
32	C800	cache w1	-	-	-	uvme2
33	CC00	-	-	-	-	uvme3
34	D000	-	-	-	i_cache0	uvme4
35	D400	-	-	-	i_cache1	uvme5
36	D800	-	-	-	i_cache2	uvme6
37	DC00	-	-	-	i_cache3	uvme7
38	E000	color_sup	-	-	color_sup	color_sup
39	E400	color_usr	-	-	color_usr	color_usr
3A	E800	color_wcs	-	-	color_wcs	color_wcs
3B	EC00	-	-	-	-	-
3C	F000	disp1	-	-	disp1	disp1_sup
3D	F400	disp2	-	-	disp2	disp1_user
3E	F800	-	-	-	-	disp1_wcs
3F	FC00	mem ctl	-	-	mem ctl	-
40-4F	10000	pbu mem	-	iomap(4K)	pbu mem	iomap (4K)
50-57	14000	-	-	PROM2	-	PROM2
58-5B	16000	-	-	PROM2	-	PROM2
5C-5F	17000	-	-	PROM2	-	PROM2
60-7F	18000	pbu i/o	-	-	pbu i/o	new dt(32k)
80-FF	20000	disp1 mem	disp1 mem	-	disp1 mem	disp1 mem
100-17F	40000	disp2 mem	-	-	disp2 mem	color mem
180-1BF	60000	-	-	-	-	NGC ctl
1C0-1FF	70000	-	-	pbu i/o	-	pbu i/o

Figure B-2. Physical Memory Layout, Continued

PPN	ADDRESS	APOLLO 1 DN420/600	APOLLO 2 DN3XX	APOLLO 3 DSP80/90	APOLLO 4 DNX60	APOLLO 5 DN5XX
200-3FF	80000	phys mem	phys mem	pbu mem	-	pbu mem
400	100000	md data	md data	md data	-	md data
401	100400	trap pg	trap pg	trap pg	-	trap pg
402-7FF	100800	phys mem	phys mem	phys mem	-	phys mem
800	200000	"	"	"	md data	"
801	200400	"	"	"	md data2	"
802	200800	"	"	"	trap pg	"
803-804	200C00	"	"	"	smaps	"
805-824	201400	"	"	"	pmaps	"
825	209400	"	"	"	pmaps_sp	"
826	209800	"	"	"	rars	"
827-DFF	209C00	"	"	"	phys mem	"
E00-FFF	380000	"	"	pbu mem2	"	pbu mem2
1000-1BFF	400000	-	ptt(020)	ptt(020)	"	ptt(020)
1C00-1FFF	700000	ptt	ptt	ptt	"	ptt
2000-3FFF	800000	-	-	-	"	-

Figure B-3. Object Locations

OBJECT	APOLLO 1	APOLLO 2	APOLLO 3	APOLLO 4	APOLLO 5
--------	---------	---------	---------	---------	---------
PROM	0	0	0	0	0
PROM2	-	-	14000	4000	14000
pft	4000	4000	4000	-	4000
MMU	8000	8000	8000	8000	8000
sio	8400	8400	8400	8400	8400
timers	8800	8800	8800	8800	8800
calendar	8880	8880	A000	8880	9C00
chameleon	-	-	-	-	9400
floppy	8C00	9C00	-	8C00	-
cart. tape	-	-	-	-	9C00
onb cache	-	-	-	-	-
pbu ctl	-	-	A400	-	A400
lpr	-	-	A800	-	-
peb ctl	B000	B000	B000	-	B000
fpu cmd	B400	B400	B400	-	B400
useqncr ctl	-	-	-	B000	-
dcode ram	-	-	-	B400	-
dma	-	9000	9000	-	-
ring1	B800	-	-	B800	-
ring2	BC00	9800	9800	BC00	9800
vme ctl	-	-	-	-	BC00
win	-	9C00	-	-	9C00
fpu cs	C000	B800	B800	-	B800
cache w0	C400	-	-	-	-
cache w1	C800	-	-	-	-
wcs ctl	-	-	-	C000	-
inst cache0	-	-	-	D000	-
inst cache1	-	-	-	D400	-
inst cache2	-	-	-	D800	-
inst cache3	-	-	-	DC00	-
color_sup	E000	-	-	E000	E000
color_user	E400	-	-	E400	E400
color_wcs	E800	-	-	E800	E800
disp1	F000	9400	9400	F000	F000
disp2	F400	-	-	F400	-
disp1_user	-	-	-	-	F400
disp1_wcs	-	-	-	-	F800
mem ctl	FC00	-	-	FC00	-
iomap	-	_	10000	9000-91FF	10000
pbu mem	10000	-	80000	10000	80000
pbu i/o	18000	-	70000	18000	70000
disp1 mem	20000	20000	20000	20000	20000
disp2 mem	40000	-	-	40000	40000
color mem	40000	-	-	40000	40000
MD data	100000	100000	100000	200000	100000
ptt	700000	700000	700000	-	700000

Figure B-4. Virtual Memory Allocation for 16MB Systems

	DN420/600	DN300/320	DSP80	DN550
VIRTJAL	APOLLO 1	APOLLO 2	APOLLO 3	APOLLO 5
0	trap page	trap page	trap page	trap page
400-3FFF	PROM	PROM	PROM	PROM
4000-7FFF	-	-	PROM2	PROM2
7000	fpu_cmd	fpu_cmd	fpu_cmd	fpu_cmd
8000-1FFFFF	GLOBAL ADDRESS SPACE			
200000-AFFFFF	PRIVATE ADDRESS SPACE			
700000-7FFFFF	ptt	ptt	ptt	ptt
B00000-BBFFFF				
BC0000-BFFFFF	PRIVATE PROTECTED ADDRESS SPACE			
C00000-DFFFFF	-	-	-	-
E00000-EFFFFF	OS PROC AND DATA			
F00000-F68000	OS BUFFERS			
F70000-F8FFFF	-	-	-	-
F90000-F9FFFF	-	-	-	NGC ctl
FA0000-FBFFFF	disp2_mem	-	-	color_mem
FC0000-FDFFFF	disp1_mem	disp1_mem	-	disp1_mem
FE0000-FE7FFF	pbu_mem (32K)	-	pbu_i/o	pbu_i/o
FE6800	-	-	pbu_sm_c_page	pbu_sm_c_pg
FE6C00	-	-	pbu_sm_page	pbu_sm_page
FE7800	-	-	pbu_mt_page	pbu_mt_page
FE8000-FEFFFF	pbu_i/o (32K)	-	pbu_i/o	pbu_i/o
FEE800	pbu_sm_c_page	-	-	-
FEEC00	pbu_sm_page	-	-	-
FEF800	pbu_mt_page	-	-	-
FEFC00	pbu_npa_page	-	-	-
FF0000-FF4FFF	-	-	-	-
FF5000-FF5FFF	-	-	iomap (4K)	iomap (4K)
FF6000	color_sup	-	-	color_sup
FF6400	color_user	-	-	color_user
FF6800	color_wcs	-	-	color_wcs
FF6C00	-	-	-	-
FF7000	peb_ctl	fpu_ctl	fpu_ctl	fpu_ctl
FF7400	fpu_cmd	fpu_cmd	fpu_cmd	fpu_cmd
FF7800	fpu_cs	fpu_cs	fpu_cs	fpu_cs
FF7C00	cache_w0	-	pbu_ctl	pbu_ctl
FF8000	cache_w1	-	lpr	chameleon
FF8400	-	-	-	-
FF8800	chk_buff	chk_buff	chk_buff	chk_buff
FF8C00	zbuff	zbuff	zbuff	zbuff
FF9000	mem ctl	par_buff	par_buff	par_buff
FF9400	disp2	-	-	vme_ctl
FF9800	disp1	disp1	-	disp1_sup
FF9C00	ring2	ring	ring	ring
FFA000	ring1	dma	dma	disp1_user
FFA400	-	-	calendar	disp1_wcs
FFA800	flop	flop.win.cal	-	win.tape.cal
FFAC00	timr. cal	timr	timr	timr
FFB000	sio	sio	sio	sio
FFB400	MMU	MMU	MMU	MMU
FFB800-FFF7FF	pft (16K)	pft	pft	pft
FFF800-FFF9FF	iomap	-	-	-

Figure B-5. Virtual Memory Allocation for M68020 Systems

	DN330	DSP90	DN560
VIRTUAL	APOLLO 2	APOLLO 3	APOLLO 5
0	trap page	trap page	trap page
400-3FFF	PROM	PROM	PROM
4000-7FFF	PROM2	PROM2	PROM2
8000-1FFFFF	GLOBAL ADDRESS SPACE		
200000-3BBFFFF	PRIVATE ADDRESS SPACE		
3BC0000-3CFFFFF	PRIVATE PROTECTED ADDRESS SPACE		
3D00000-3EFFFFF	OS PROC AND DATA		
3F00000-3F68000	OS BUFFERS		
3F70000-3F8FFFF	-	-	-
3F90000-3F9FFFF	-	-	NGC ctl
3FA0000-3FBFFFF	-	-	color_mem
3FC0000-3FDFFFF	disp1_mem	-	disp1_mem
3FE0000-3FE7FFF	-	pbu_i/o	pbu_i/o
3FE6800	-	pbu_sm_c_page	pbu_sm_c_pg
3FE6C00	-	pbu_sm_page	pbu_sm_page
3FE7800	-	pbu_mt_page	pbu_mt_page
3FE8000-3FEFFFF	-	pbu_i/o	pbu_i/o
3FF0000-3FF4FFF	-	-	-
3FF5000-3FF5FFF	-	iomap (4K)	iomap (4K)
3FF6000	-	-	color_sup
3FF6400	-	-	color_user
3FF6800	-	-	color_wcs
3FF6C00	-	-	-
3FF7000	fpu_ctl	fpu_ctl	fpu_ctl
3FF7400	fpu_cmd	fpu_cmd	fpu_cmd
3FF7800	fpu_cs	fpu_cs	fpu_cs
3FF7C00	-	pbu_ctl	fbu_ctl
3FF8000	-	lpr	chameleon
3FF8400	-	-	-
3FF8800	chk_buff	chk_buff	chk_buff
3FF8C00	zbuff	zbuff	zbuff
3FF9000	par_buff	par_buff	par_buff
3FF9400	-	-	vme_ctl
3FF9800	dispi	-	disp1_sup
3FF9C00	ring	ring	ring
3FFA000	dma	dma	disp1_user
3FFA400	-	calendar	disp1_wcs
3FFA800	flop,win,cal	-	win,tape,cal
3FFAC00	timr	timr	timr
3FFB000	sio	sio	sio
3FFB400	MMU	MMU	MMU
3FFB800-3FFF7FF	pft	pft	pft

Figure B-6. Virtual Memory Allocation for 256MB Systems

	DN160/460/660
VIRTUAL	APOLLO 4
0	trap page
400-3FFF	PROM
4000-7FFF	PROM2
8000-7FFFFF	User Global
800000-F77FFFF	User Private
F780000-F7FFFFF	Protected Private
F800000-FEFFFFF	OS Proc and data
FF00000-FF68000	OS Buffers
FF70000-FF9FFFF	-
FFA0000-FFBFFFF	disp2_memory
FFC0000-FFDFFFF	disp1_memory
FFE0000-FFE7FFF	pbu_memory (32K)
FFE8000-FFEFFFF	pbu_i/o (32K)
FFEE800	pbu_sm_c_page
FFEEC00	pbu_sm_page
FFEF800	pbu_mt_page
FFEFC00	pbu_npa_page
FFF0000-FFF5FFF	-
FFF6000	color_supervisor
FFF6400	color_user
FFF6800	color_wcs
FFF6C00-FFF87FF	-
FFF8800	check_buff
FFF8C00	zbuff
FFF9000	memory_ctl
FFF9400	display2
FFF9800	display1
FFF9C00	ring2
FFFA000	ring1
FFFA400	-
FFFA800	floppy disk
FFFAC00	timer, calendar
FFFB000	sio
FFFB400	MMU
FFFB800-FFFF7FF	-
FFFF800-FFFF9FF	iomap

Appendix C
Canned UIDs

This appendix lists the system entities that are assigned canned UIDs.

Figure C-1. Canned UIDs

Canned ACL UIDs (file ACLs) -- 0001.xxxx series

Object	Explanation
acl_$nil	All access (file or directory)
acl_$fnil	No access (file)
acl_$fndwrx	All access (file)
acl_$file_nwrx	File creation ACL used by INVOL

Canned ACL UIDs (directory ACLs) -- 0002.xxxx series

acl_$dnil	No access (directory)
acl_$dndcal	All access (directory)
acl_$dir_ncal	System creation ACL used by INVOL

Disk structure canned UIDs -- 0000.02xx series

pv_label_$uid	UID of the physical volume label
lv_label_$uid	UID of logical volume label
vtoc_$uid	UID of the VTOC
bat_$uid	UID of the BAT

Canned object type UIDs -- 0000.03xx series

uid_$nil	Nil uid
records_$uid	File type of record structured files
hdr_undef_$uid	No record structure, but has headers
object_file_$uid	Object module type
UNDEF_$uid	Completely undefined objects type
pad_$uid	Stream/display manager transcript pad
input_pad_$uid	Stream/display manager input pad
sio_$uid	SIO descriptor file type
ddf_$uid	Device descriptor file (DDF) file type
mbx_$uid	IPC mailbox file (MBX) file type
nulldev_$uid	Null device (/dev/null)
d3m_area_$uid	Area files (D3M)
d3m_sch_$uid	Object (sub)schema files (D3M)
pipe_$uid	IPC pipe file type
uasc_$uid	DOMAIN ASCII file type
directory_$uid	Directory - for use with streams only
unix_directory_$uid	Directory - for use with streams only
mt_$uid	Magtape type uid
sysboot_$uid	Type uid for sysboot file

Figure C-2. Canned UIDs, Continued

```
Canned object UIDs -- 0000.04xx series

display1_$uid
display2_$uid
name_$canned_root_uid      Father pointer in node root directory
special_seg_$uid           Place holder in MST for special segment
diskless_$uid              Used by naming server
name_$canned_rep_root_uid  Canned replicated root uid


Canned PPO and subsystem UIDs -- 005xx series

acl_$sys_user_uid          AEGIS user (user.%.%)
acl_$sys_proj_uid          AEGIS project (%.AEGIS.%)
acl_$login_uid             Login manager project (%.login.%)
acl_$locksmith_uid         Locksmith
acl_$sys_org_uid           AEGIS organization (%.%.aegis)
acl_$nil_subs_uid          Nil subsystem

Canned ACL type UIDs --- the kinds of objects ACL protects

acl_$file_acl
acl_$dir_acl
```

GLOSSARY

> "When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean -- neither more nor less."
>
> Lewis Carroll, Through the Looking Glass

absolute pathname

An ordered list of component names that gives the path to an object starting from the network root directory (//). Because it begins at the root of the tree, the absolute pathname is valid throughout the entire network.

access control list (ACL)

A type of object that defines who is permitted access to one or more objects and also how those objects can be accessed.

access violation

An attempt to write to read-only memory.

acknowledge (ACK) byte

A byte in the packet header that indicates whether or not the target node received the packet or if an error occurred during its circulation through the ring.

active object

An object that has been recently used.

activating a segment

The AST manager operation that copies information about an object from its volume table of contents entry (VTOCE) to an active segment table entry (ASTE).

active segment table (AST)

The wired, per-system table that caches the locations and attributes of active (recently used) local and remote objects.

active segment table entry (ASTE)

An entry in the AST that caches information about one object segment.

active segment table entry index (ASTEX)

The index to a specific active segment table entry (ASTE).

address space identifier (ASID)

An integer from 0 to 25 that identifies the virtual address space allocated to a process. User and supervisor global address spaces are assigned ASID 0. The first process created at system initialization uses ASID 1; thereafter, the display manager uses ASID 1. Level 2 processes are allocated ASIDs 2 through 25.

address translation

The hardware operation that generates a physical address from a virtual address.

AEGIS kernel

The protected operating system software that runs in supervisor mode and exists in supervisor global address space.

ASKNODE service

See **node status inquiry service.**

alternate logical volume label

A copy of the logical volume label that INVOL creates when it initializes a logical volume. The alternate logical volume label makes it possible for SALVOL to reconstruct the logical volume label should the original be destroyed.

AST replacement

The procedure that locates and deactivates least recently used ASTEs.

asynchronous fault

A software-defined fault condition that occurs independently of a program's instruction execution.

atomic transaction

A transaction that is completely "enclosed" such that either all subtransactions (steps) occur or none of them occur. An atomic transaction is guaranteed to complete or perform no operation at all.

back segment table (BST)

The per-process table that links togehter the processes that are currently sharing the same object segment. This table only exists on forward-mapped systems.

badspot

A media defect on a disk that renders one or more blocks unusable for data storage.

badspot cylinder

The physical disk structure that records the physical badspots (unusable blocks) that exist on the volume. The badspot cylinder is usually one of the last two cylinders on a disk.

bat step

A field that controls how the BAT manager allocates blocks when it lays out an object. For example, a bat step of 2 tells the BAT manager to allocate the next block at block n+2. Users set the bat step, via INVOL, to optimize disk seeks; the correct bat step for a given volume is the value that allows the disk to capture as many consecutive blocks in a single revolution as possible.

binding a process

The operation that creates a level 1 process and starts it running at a specified procedure entry.

biphase error

An error that occurs when a node receives an electrical signal and cannot regenerate it.

bit BLT

Bit block transfer. Process by which display hardware moves arbitrary rectangular areas at high speeds.

block availability table (BAT)

A table in the logical volume (a bitmap) that describes how the disk blocks on the logical volume are currently allocated.

block availability table (BAT) manager

The manager that allocates and frees disk blocks using the BAT.

bootshell

The system initialization program that displays the Apollo logo and runs the user environment initialization (ENV) program.

bootstrap PROM

The portion of a node's PROM that initializes the node. The bootstrap PROM's system initialization functions include logic that initializes hardware that the PROM references, a set of minimal device drivers for the I/O devices connected to a node, and logic to load SYSBOOT or NETBOOT into memory. See also **mnemonic debugger** and **PROM**.

broadcast

A message that all nodes on the network will receive and process.

cached object storage system (OSS)

The managers that handle the per-node cache of recently used local and remote objects. Also called the virtual memory management subsystem.

canned UID

A pre-defined UID that identifies an object across multiple AEGIS systems. A canned UID is guaranteed to be the same on each AEGIS system, rather than the usual UID, which is always unique. Examples of objects assigned canned UIDs are I/O devices and disk data structures (VTOC, BAT).

cartridge tape bootstrap program (CTBOOT)

The program that initializes the AEGIS system on nodes configured with a cartridge tape as the boot device.

client process

A process that calls a network support software's client side to carry out a remote request on its behalf.

client-server protocol

See **request-response protocol**.

client side

The portion of the network support software that sends out requests chosen from its menu of remote services.

cold start routine (cold_start)

The system initialization routine that the PROM calls to initialize the AEGIS kernel. The cold start routine initializes the memory management unit and the memory map and enables VME and MULTIBUS devices, if they exist.

common fault handler

The fault interceptor module routine that handles *all* faults.

concurrency control

The object attribute that controls simultaneous access to the object among several processes. Concurrency control can be:

- None, where no concurrency control is applied. The system applies no concurrency control to temporary, uncatalogued objects.

- Shared reading or exclusive writing, (file_$nr_xor_1w), where any number of processes can read the object at the same time, but only one process at a time is allowed to write the object.

- Shared writing, (file_$cowriters), where any number of processes can write the object at the same time. The system limits shared writing to processes running on one node, although these processes do not have to be on the same node as the object they are referencing.

The lock manager enforces the object's concurrency control attribute.

context

See **process context.**

context switch

The operation that makes a process the current process. Also called process exchange and dispatching.

current process

The process that is currently using the processor.

data structure

A structure on which many separate operations are performed, for example, a window or a pad.

demand paging

The system process that dynamically transfers 1024-byte pages of an object residing on local disk or remote node to the requestor, be it on the local node or on another node in the network.

device driver

A collection of system-level commands to device-dependent control/status register bits.

device volume table (DVT)

The table that lists the logical volumes on a server node.

diagnostic frame

The data structure that describes a fault to the user-mode fault handler. The fault interceptor module's common fault handling routine builds the diagnostic frame and places it on the process's supervisor stack.

diagnostics cylinder

A disk cylinder that is reserved for disk diagnostic or controller diagnostic operations, and for use by the on-line TESTVOL program (usually the last or next-to-last cylinder on the disk).

directory

An object that contains a set of associations between pathname component names and UIDs.

directory manager (DIR)

The manager that creates, deletes and manages the use of directories.

disk address (DADDR)

A disk block's sequence number relative to the start of the physical or logical volume.

disk address space

The 1024-byte disk blocks on a disk, identified by disk addresses (DADDRs).

disk block

1024 bytes of data plus a 32-byte disk block header. Floppy disk blocks do not have disk block headers.

disk buffer manager (DBUF)

The mechanism that the VTOC and BAT managers use to cache object file maps (their disk addresses on the logical volume).

disk entry directory (/)

The first directory level on a node's boot volume. Also called the local root directory or the node entry directory.

diskless node

A node that is not configured with a disk boot volume. The backing disk storage for a diskless node exists on other nodes on the network.

diskless node bootstrap program (NETBOOT)

The program that initializes the AEGIS system on nodes that are not configured with a system bootstrap device (cartridge tape or disk volume). NETBOOT works in conjunction with the AEGIS user-mode diskless node bootstrap server NETMAN, which runs on the partner node.

diskless node bootstrap server (NETMAN)

The AEGIS user-mode server that waits for dump, echo and boot service requests at its well-known socket. NETMAN works in conjunction with NETBOOT to bootstrap a diskless node. NETMAN runs on the diskless node's partner while NETBOOT runs on the diskless node.

disk structure

The data structures that define how objects are stored on disk volumes attached to a node. The AEGIS system separates disk structures into logical and physical sections.

dispatcher

The level 1 process manager operation that locates the highest ready process and dispatches it.

dispatching

The operation that switches the processor from one process to another. Dispatching makes the context of the first process on the ready list current and saves the context of the previous current process. Also called **context switch** and **process exchange**.

display manager

The software process that manages the screen environment in each node. The display manager allows the user to view and control separate activities both concurrently and independently by dividing the screen into windows whose size, shape, and placement are under user control.

dynamic reference

A references to a system service that is resolved at load time.

early acknowledge (EACK) byte

A field within the ring hardware header that the ring hardware on a node uses indicate that it intends to copy the message passing through it. Because the EACK byte arrives before the ACK byte, the ring hardware that transmitted the message can use this field to determine whether or not to continue a message transmit.

elastic store buffer

The buffer that takes up the slack between a node's input and output ports. An elastic store buffer error occurs when a node's elastic store buffer overflows.

eventcount

An object that keeps a count of the number of events within a particular class that have occurred so far in the execution of the system. Eventcounts are used for process synchronization.

exception

An event detected detected by the hardware that changes the normal flow of instruction execution.

exception error stack frame

The frame of information built by the central processor as the result of a hardware exception. Also called a fault frame.

fault

An exception that is generated by processor hardware or system or program software. The fault interceptor manager (FIM) handles hardware-generated faults. User-mode fault handlers service software-generated faults. Hardware-generated faults restart the instruction that caused the fault. Resumption of execution after a software fault depends on how the user fault handler is designed.

file manager (FILE)

The AEGIS kernel manager that provides location-independent access to an object's attributes and allows user-mode programs to create and delete objects.

file map

The disk volume data structure that provides a road map to the disk blocks that contain an object's pages.

file system object

A named collection of bits.

flushing the cache

The process that removes an object's obsolete pages from the active segment table.

flushing

The operation that writes modified pages to disk and also breaks the object-to-physical page association.

force-write

The process of writing the modified pages of an object to its home disk on demand.

forking a process

The operation that creates a new process and copies the parent process's context -- stack contents, locks, mapped objects -- into the forked process's address space.

forward-mapped MMU

The memory management unit that DNx60 models use. The Forward-mapped MMU organizes its hardware tables into a 3-level tree structure.

global libraries

Libraries that exist in the address space of all processes. The process manager automatically installs these libraries into user global address space as part of its initialization process when a node is bootstrapped (when the DM or SPM is first invoked.)

growth fault

The type of page fault that occurs when a process is adding a new page to an object.

guard fault

The type of fault that occurs when a program tries to reference a guard segment.

guard segment

A segment to which no access at all is allowed. The system surrounds important system resources with guard segments to protect them from accidental corruption or destruction.

hashing

The process by which large, complex numbers are mapped to smaller, more manageable numbers. Hashing reduces the area of data that system has to search because it reduces a large range of numbers to a smaller range. The object storage system hashes UIDS to an index into a volume table of contents (VTOC) entry. The naming server hashes pathname components into an index into the directory's hash thread table.

heap management

A read/write storage management technique that divides the storage into free areas and areas in use. Heap management allows storage to be allocated and freed in any order. In contrast, a stack allocates and frees storage on a last-in/first-out basis.

hint

The node ID portion of an object's UID and an internet address at which the object corresponding to this UID was last located.

hint manager

The AEGIS service that aids in determining an object's location in the network.

home disk

The logical volume that contains an object's permanent storage (its VTOCE entry and its file map).

home node

The node to which an object's home disk is attached.

hop

One passage through a routing node.

immutable object

An object that cannot ever be modified. An access control list is an example of an immutable object.

impure data

Data that changes during execution, for example, program variables. Also called read/write data.

internet

A group of two or more networks connected by communications hardware.

internet address

A 32-bit network number and a 20-bit node ID.

internet datgram protocol (IDP) header

The part of a packet header that contains source and destination information used by the internet routing software and AEGIS low-level IPC during message transfer.

internet routing software

The AEGIS kernel managers that provide the ability to pass messages to other networks through routing nodes.

interprocess communications (IPC) header

The portion of a packet header that contains source and destination information used by the user-mode IPC_$ system services.

interrupt

A hardware-generated event that takes the processor away from the currently running process. Interrupts vector through the trap page directly to driver interrupt service routines.

I/O address space

The region of address space that contains the status registers and data structures for the memory management unit and for the I/O devices connected to a node.

kernel

See **AEGIS kernel.**

known global table (KGT)

The system table that stores the entry points of all the libraries installed in user global address space.

leaf component

The last component in a pathname.

level 1 process

A process that runs supervisor-mode code and thus runs exclusively in M680x0 supervisor operating mode. All processes are level 1 processes. Some processes remain level 1 processes, and some are augmented (bound) to level 2 processes. A level 1-only process does not possess its own virtual address space. Instead, it uses supervisor global address space. Also called kernel process, supervisor process, and supervisor-mode process.

level 1 process (PROC1) manager

The AEGIS kernel manager that creates and deletes level 1 processes and provides scheduling, interrupt handling, and dispatching operations.

level 2 process

A process that is bound to a level 1 process and which possesses its own per-process virtual address space. The user-mode portion of a level 2 process is its pageable virtual address space. The supervisor-mode portion of a level 2 process is the level 1 process underneath it.

level 2 process (PROC2) manager

The AEGIS kernel manager that creates and deletes level 2 processes, manages level 2 process context and UID namespace, and handles asynchronous fault posting and delivery.

linchpin ASTE

The ASTE that describes the object's highest numbered active segment.

link

A pathname component that causes a jump from one point in the naming tree to another. Links are commonly used to refer indirectly to an object whose absolute pathname may change over time, and as shorthand notation for frequently used pathnames.

local object storage system (OSS)

The managers that handle the storage of objects on disk volumes attached to the local node.

location-independence

A property of AEGIS system architecture that separates object identification from object location. In the AEGIS system, an object's UID uniquely identifies the object no matter where it resides, but it does not specify the object's location.

lock key

The object attribute that enforces the higher-level concurrency control that the lock manager has assigned to the object. An object's lock key reflects how an object is currently locked; it can be any access, all readers, or the node ID of the writer. The paging server on an object's home node checks the lock key to ensure the requested access is compatible with the lock on the object.

lock manager

The AEGIS kernel manager that synchronizes process access to objects in the network. The lock manager enforces object concurrency rules and supports the distributed system.

logical volume

The disk structure that the local object storage system uses to record the storage of objects on the disk. One physical volume can contain a maximum of 10 logical volumes.

low-level interprocess communication (IPC) software

The AEGIS software that provides the ability to send messages to and from other nodes in the network. This software is composed of the socket datagram service, the packet protocol, and the network buffer pool.

mailbox facility (MBX)

The user-mode AEGIS message facility.

manager

A collection of modules that provide the means for external users to access and manipulate objects.

mapped mode

The PROM mode of operation in which the memory management unit is enabled and loaded with the PROM's mapping of memory and I/O devices.

mapped section manager (MS)

The manager that caches pieces of object work space by calling on the MST manager (SLS) and the lock manager.

mapped segment table (MST)

A per-process table that lists the segments of process virtual address space and the object segments to which they refer. The system indexes into an array of MSTs by ASID and virtual segment number.

mapped segment table manager (MST)

The manager that provides location-independent access to an object's pages. The MST manager carries out object address to virtual address mapping and unmapping requests and participates in page fault resolution. Also called the single-level store manager.

mapping

The system procedure that sets up an association between a local or remote object and a process's virtual address space so that the process can refer to the object directly by referencing addresses in its virtual memory. Also called single-level store.

memory management unit (MMU)

The hardware that carries out virtual address to physical address translation.

memory map (MMAP)

The per-system data structures that keeps track of each physical page and its availability for use.

memory map manager (MMAP)

The manager that finds an available physical page to hold an object page during object page/physical page association.

message interface (MSG)

The AEGIS kernel manager that provides user-mode AEGIS services and user programs with access to the socket datagram service.

mnemonic debugger (MD)

The portion of the node's PROM that implements the low-level debugging facility. The mnemonic debugger provides a set of debugging commands and an assembler/disassembler. Whenever the system crashes, it automatically enters the MD.

module

A collection of subroutines that carries out a single operation.

MST back thread

A field within an ASTE that points to all the processes that have mapped the active segment to their virtual address spaces.

multitasking

Running multiple programs and/or multiple processes using the mutex as the synchronization mechanism.

mutex lock (ML)

The lock that synchronizes acess to level 1 process resource locks.

naming server

The AEGIS manager that links user level text-string naming to system-level UID naming. The naming server is a kernel-space type manager.

naming server helper (NS_HELPER)

The user-mode server program that manages a master copy of the network root directory. It works in conjunction with the naming server, the directory manager, and the supervisor mode NS_HELPER client software to manage each node's network root (//) directory automatically as nodes are added to the internet, moved from one network to another, and removed from the internet.

network

Any communications medium that allows the AEGIS system to send packets through it.

network buffer pool

The pool of virtual pages available to clients of sockets to hold incoming or outgoing packets.

network buffer pool manager (NETBUF)

The manager that maintains the network buffer pool.

network I/O manager (NET_IO)

The AEGIS kernel manager that fields packet sends and receives to the various networks connected to a routing node.

network manager (NETWORK)

The manager that provides remote access to the attributes and pages of existing objects, exports the ring hardware type mask to AEGIS software, and initializes network-related databases and servers during system initialization.

network number

The number that identifies the communcations line to which a node is connected.

network port

The mechanism that links a network to the AEGIS system on a node. A node can support seven ports. All nodes have one port that identifies the ring network to which they are connected. Routing nodes have several ports that identify the other rings, T1 (bridge) lines, or MULTIBUS network devices connected to the node.

network port descriptor

A small integer that the internet routing software assigns to a network when it connects it to the internet and opens it for AEGIS low-level IPC operations. The port descriptor is the node's way to refer to the network hardware connected to it, and is important only to the node that owns the hardware.

network port table

The table that describes the network devices connected to a node. Each entry in the table gives the network device's network number, the kind of service the network provides, and the kind of network device that exists at that port; for example, ring, T1 or user network.

network root directory (//)

The network-wide root directory that is replicated on every node. The network root directory exists in the logical volume label; no other data structure either catalogues it or points to it.

network support software

The kernel and user-mode AEGIS software that offers a variety of network services such as a remote file system, node status inquiry, internet routing, and remote naming.

network-wide namespace

The database that stores object pathnames and which is structured as a multilevel directory tree. The base of the tree is a network-wide root directory that catalogues each node; branches of the tree represent directory objects; leaves on the tree represent particular objects.

node status inquiry service (ASKNODE)

The AEGIS kernel manager that allows clients to request and receive miscellaneous information about other nodes on the network, such as network statistics, software and hardware performance, and a node's hardware configuration.

normal mode

The mode of system initialization that proceeds automatically from power-on or reset to the point of login.

null page

A physical page whose corresponding disk storage contains invalid data.

null process

A special level 1 process that is always ready and always has the lowest priority. The dispatcher runs the null process when no other processes are ready to run.

orphan process

A process that has no parent process.

OS paging file

An uncatalogued, permanent object that must exist on any logical volume to be used as the boot device for AEGIS. The paging file is the backing storage for the pageable parts of the AEGIS operating system.

object

1. Sealed data and the operations that observe and manipulate the data (the computer science definition).

2. A storage container that is defined by its set of operations (the AEGIS system definition). An object in the AEGIS system is an abstraction of system entities such as files, directories, bitmaps, and processes.

object address space

The network wide object name space identified by 96-bit object addresses. Object address space refers to the total object storage available in the network.

object attributes

Fields in the object's volume table of contents entry (VTOCE) that describe the object to the object storage system and other AEGIS kernel and user-space managers. Object attributes include the date and time last modified; the kind of concurrency control applied; whether or not the object is permanent, temporary, or immutable; the UIDs of its type manager, its associated access control list, and the directory in which it is catalogued; a count of the references made to it.

object locating service

The parts of the AEGIS system that locate an object in the network given its UID.

object locking

The procedure that controls how many processes can simultaneously reference an object page.

object storage system (OSS)

The AEGIS components that carry out object management; also called the file system. The object storage system consists of local, remote and cached subsystems plus an object locating service and a higher-level, location-independent object management service.

object work space

The set of objects that are mapped into a single process's address space.

odd address error

A fault that occurs when a process generates an address that does not lie on a word boundary.

packet

A message on the ring. A packet is divided into a header section and an optional data section.

packet exchange protocol (PEP)

The portion of a packet header that contains control information used by the network support software that implements the request-response protocol; for example, the remote file system.

page

1024 bytes of object address space, virtual address space, or physical address space.

page fault

The hardware exception that occurs when a process references a virtual address that has no corresponding physical address association in the memory management unit tables.

page frame table (PFT)

The reverse-mapped MMU hardware table that describes, for each page in physical memory, the association between the physical page and a virtual page.

page-in request

A request to read multiple object pages from a remote home disk into the requesting node's physical memory.

page map (PMAP)

The data structure that stores the file map for one segment of an object and contains the locations in physical memory of any segment pages that have been referenced.

page map manager (PMAP)

The manager that associates object pages with physical pages and brings in the object pages from disk or from the network if necessary.

page-out request

A request to write out the modifications made to an object page at a remote site to the object's home node for eventual storage on the home disk.

page replacement

The operation that takes away a physical page that one process is using so that another process can use it to hold and object-to-physical page association.

page translation table (PTT)

The reverse-mapped MMU hardware translation buffer. The page translation table contains entries for 1023 physical page numbers, which allows the system to map 1 megabyte of virtual memory to physical memory at any given time. Each PTT entry points to a page frame table entry.

paging

The process that moves pages of virtual memory in and out of physical memory (unless those pages are wired).

paging file

See **OS paging file**.

paging server

See **remote paging server**.

paging system

The mapped segment table manager, the active segment table manager, the page map manager, memory map manager, and remote paging server. Also called the virtual memory management subsystem.

partner node

A node that is set up to handle diskless node bootstrap requests from one or more diskless nodes on the network.

pathname

A text string that identifies an object. The pathname gives the full route through the directory tree name space to the object.

pathname component

The part of a pathname that provides a portion of the route through the directory tree to an object.

permanent object

An object that cannot be deleted by the system during a shutdown or crash and which can be catalogued in the directory subsystem.

physical address space

The main memory that exists in each node and which is identified by physical addresses.

physical mode

The PROM mode of operation in which the memory management unit is disabled.

physical page number (PPN)

The number that identifies a page in physical memory.

physical volume

A collection of disk blocks. One physical volume can contain several logical volumes.

physical volume label

The data structure (a single disk block) that describes the physical disk and locates the logical volumes on the disk.

posting an asynchronous fault

The operation that sends an asynchronous fault to a level 2 process.

private known global table (KGT)

The table that contains the entry points to livraries installed dynamically into user private address space with the INLIB program.

PROC1 manager

See **level 1 process manager**.

PROC2 manager

See **level 2 process manager**.

process

1. An abstract machine available to each user.

2. A thread of execution.

3. The basic schedulable entity in the operating system. A maximum of 16 processes can exist at one time (per node). (Note: limit is due to need for wired memory. If PROC2 bind/unbind rewritten to stop the need to wire memory, unlimited number of processes will be possible.)

4. A program's "computer" that runs on the behalf of a user.

See also **level 1 process** and **level 2 process**.

process blocking

The procedure in which a process blocks itself from being runnable by waiting on an eventcount.

process context

M680x0 processor state and, optionally, a process virtual address space. Only level 2 processes acquire their own process address space. See also level 1 process and level 2 process.

process control block (PCB)

The data structure that stores level 1 context.

process creation record

The segment of a user stack that contains all the information that the user-mode process manager needs to carry out its operations. The process creation record identifies the streams and arguments passed to the process and identifies any p.ograms to be invoked.

process exchange

The AEGIS system operation that switches a process's context in and out of the processor. Also called context switch and dispatching.

process manager

The user-mode AEGIS manager that manages program invocation and execution.

process priority

An integer value from 1 to 16 that determines the process's scheduling priority. Priority 16 is the highest priority. The scheduler decrements a process's priority each time it exhausts its time slice and increments a process's priority each time it awakens after an eventcount wait.

processor access modes

Bits maintained by processor and memory management hardware that define the kind of access permitted to an object page. Processor access modes specify whether the page can be read, written, or executed, and in what hardware access mode (supervisor or user) these types of access are permitted. The processor checks the access mode during virtual-to-physical address translation and issues an access violation if the requested access does not match the processor access mode specified in its tables.

private libraries

Libraries that exist only in the address space of processes in which they have been installed. Users install private libraries with the INLIB program. The program loads the libraries and also places their externally callable entry points into the known global table.

PROM

The part of a node that contains code to bootstrap a node and code to run a low-level debugger. Also called the **mnemonic debugger (MD)** and the **bootstrap PROM**.

pure data

Read-only data; that is, data that does not change during program execution, for example, instructions and constants. Since pure data does not change, the loader can map it read-only into process address space and use the object module as backing storage.

purification

The system operation that writes modifications to an object page from physical memory to the object's home disk volume for storage.

read/write data

See **impure data**.

ready process

A process that runnable. A runnable process is bound, not suspended and not waiting on an eventcount.

ready list

A doubly linked list of ready process's control blocks that is ordered by scheduling priority from highest ready process to lowest ready process.

region

A 256-segment section of virtual memory. The DNx60 machine partitions virtual memory into 31 regions.

registering an eventcount

The operation that records an exported level 1 eventcount in the level 2 eventcount manager's database so that user-mode processes can wait on the level 1 eventcount.

relative pathname

An abbreviated absolute pathname that is "relative" to one of a number of points in the naming tree: the node entry directory, the working directory, or the naming directory.

relocation table

remote file (REMFILE) manager

The manager that handles remote requests for location-indpendent object management, directory maintenance, and lock requests. The manager consists of a client side and server side called the remote file server.

remote file server

The portion of the remote file manager that handles remote file client side requests from remote nodes. The remote file server calls the local FILE manager, AST manager, or lock manager to handle the request, then passes back the answer to the client in a reply packet.

remote file system

See **remote object storage system**.

remote object storage system

The collection of managers that provide access to objects on remote nodes. The remote object storage system is layered on top of the socket datagram service and uses this service to send and receive messages over the network. Also called the **remote file system**.

remote paging server

The special level 1 process that services all requests from remote nodes to read or write pages and attributes of objects on the local node.

replicated data

Multiple copies of files, directories, and other objects.

reply socket

The socket that a process allocates to receive responses to requests that it has generated.

request-response protocol

The protocol that the network support software uses to handle the requesting and servicing of remote operations. Also called **client-server protocol**.

request server

A special level 1 process that waits on the remote file server socket, the information socket, and the routing information protocol (RIP) socket for incoming packets to the remote file server, the ASKNODE server, and the RIP server, respectively. When a packet arrives at one of the sockets, the request server invokes the appropriate server to process the packet.

resource lock

The locks that processes use to synchronize access to protected system resources. A process can only hold a resource lock while it is running in supervisor mode.

reverse-mapped MMU

The memory management unit present on all node models except the DN160, DN460, and DN660. The reverse-mapped MMU does not use forward-referenced hardware tables.

router

See **routing process**.

routing

The AEGIS internet software procedure that sends packets from one network to another.

routing node

A node that is configured to support internet packet routing.

routing process

A special level 1 process that runs only on routing nodes. The routing process forwards packets to their internet destinations and periodically sends out the latest routing information to all nodes in the internet.

routing information protocol (RIP) handler

The AEGIS kernel manager that receives and stores in the RIP table the routing information packets sent out by routing nodes. The RIP handler is divided into client and server sides. The client side allows processes to request information from the local RIP table and from RIP tables on other nodes. The server side (the RIP server) receives incoming routing information packets.

routing information protcol (RIP) table

A per-node data structure that stores routes from the node to other networks in the internet. Each entry in the table corresponds to a network in the internet. The information stored about each network includes its network number, the number of routing nodes the message must pass through to get to the network, and the next network port and hop towards that network.

run file converter (RFC)

The system initialization utility that takes the compiled and bound version of the AEGIS source code (aegis.bin) and makes it an absolute bootable image. The AEGIS RFC image (aegis.rfc) is compiled and bound, but run-time initialization of global variables has not yet occurred.

salvaged flag

A bit in an object's VTOCE that acts as a warning flag to the system that the object may be corrupted. Also called the trouble bit.

salvage volume utility (SALVOL)

The stand-alone utility that attempts to salvage a corrupted disk volume.

scheduling

The level 1 process manager operation that removes compute-bound processes from the processor.

segment

Thirty-two (32) consecutive pages. The virtual memory management subsystem divides both object address space and virtual address space into segments, and maps between these spaces in units of segments.

service mode

The mode of system initialization that invokes the mnemonic debugger command interpreter (MD).

server process

A level 1 or level 2 process that is devoted to managing a certain function. Examples of level 1 server processes are the remote paging server, the routing process, and the remote file server. Examples of level 2 server processes include the mailbox helper (MBX_HELPER)and the naming server helper (NS_HELPER).

server side

The portion of the request-response protocol that receives remote requests, handles them, and passes back the requested information to the client side.

shared object

An object that exists in the address space of more than one process.

sharing fault

The type of page fault that occurs to install the correct virtual page association into the page frame table entry in the case where several virtual pages are associated with a single physical page. A sharing fault occurs only on reverse-mapped systems.

single-level store

The part of the AEGIS file system that permits the user to view the entire network as his own mass storage area. The mapped segment table manager implements the single-level store.

socket

A queue of incoming messages; each socket is identified by a small number such as socket 1 and socket 5.

software control header

The portion of the packet header that contains source and destination information used by pre-internet versions of the AEGIS system and by the bootstrap PROM.

stack object

The object that provides the backing storage for the user process's pageable stack. The stack object contains the process creation record and an area for private read/write storage and the procedure call stack.

stack pointer

The M680x0 processor register (A7) that points to the current process's supervisor stack and, if the process has level 2 context, to the process's user stack.

stream

A channel that contains data. A stream connects a process to a node, a serial I/O (SIO) line, and other devices.

supervisor

1. The collection of AEGIS kernel modules that user-mode programs can run.

2. The AEGIS operating system code that runs only in supervisor mode. Also called the **kernel** and the **nucleus**.

supervisor address space

Protected virtual address space. Supervisor space refers to per-process supervisor private address space and to supervisor global address space. Objects mapped to supervisor address space can only be accessed with supervisor access rights.

supervisor bit

A bit in the processor status register (SR) that the processor hardware sets on a change from user mode to supervisor mode.

supervisor global address space

The protected, shared virtual address space that contains supervisor-mode code. The AEGIS supervisor, level 1 processes, and protected system data structures such as the active segment table exist in this address space. Also called Global B address space.

supervisor mode

The privileged, protected mode of operation on an M680x0 processor. Certain M680x0 intructions can only be executed when the processor is operating in supervisor mode. AEGIS kernel software uses this mode of operation.

supervisor-mode code

Software that runs in M680x0 supervisor operating mode. Supervisor-mode code contains privileged instructions that can only be executed in supervisor mode. The AEGIS kernel is composed of supervisor-mode code. Also called kernel code and supervisor code.

supervisor-mode process

A process that is using a supervisor stack and thus has an active supervisor stack pointer (SSP).

supervisor private address space

Protected, per-process virtual memory that can only be accessed when the process is running in supervisor mode. Level 2 processes must call AEGIS kernel services through an SVC trap to gain access to the information stored in supervisor private space. Also called per-process supervisor space.

supervisor process

A level 1 process that runs entirely and solely in suoervisor mode. A supervisor process does not possess its own process address space. Instead, it uses supervisor global address space. Also called kernel process.

supervisor stack

The stack that a process uses when it runs in supervisor mode. Per-process supervisor stacks exist in supervisor global address space.

supervisor stack pointer (SSP)

The M680x0 processor register that points to a per-process supervisor stack.

suspending a process

The operation that makes a ready or waiting process non-dispatchable.

SVC catcher

The trap handlers that field user-mode calls to AEGIS supervisor-mode services. The SVC catcher takes the user program's call and arguments, changes from user mode to supervisor mode, and dispatches the call to the appropriate AEGIS kernel service. When the target kernel service completes its operation, it passes control back to the user program.

SVC trap

A trap instruction that changes processor operating mode from user to supervisor mode.

synchronous fault

A processor-defined fault condition that occurs as a direct result of a program's instruction execution.

system bootstrap program (SYSBOOT)

The program that initializes the AEGIS operating system on nodes that are configured with a disk volume as their system bootstrap device.

system initialization

The collection of hardware and software procedures that bring a node from power-on or reset to the display manager's login prompt. The procedure begins with the bootstrap PROM and extends through a variety of programs and routines.

system services

The extended machine instruction set.

temporary object

An object that can be deleted by the system when it shuts down or fails. A display manager transcript pads is an example of a temporary object.

time slice

The length of process virtual time during which a process has exclusive use of the processor. The higher a process's priority, the smaller the time slice allotted to it.

touch-ahead count

The number of successive segment pages to be brought into physical memory at a time. The touch-ahead count is an argument to the mst_$touch routine.

touching a page

The operation that brings the object pages mapped to pages in process virtual address space from their home disk into physical memory.

trait

A set of operations of a generic type associated with a particular object. For example, all stream objects are associated with get, put, open, and close operations. Managers handle traits; for example, the stream manager handles the stream trait.

transaction ID

A field within the IDP (or software control) headers of a packet that request-response software uses to match up request packets with response packets. The correct reply to a request will contain the same transaction ID value as the request packet.

transceiving

The ring hardware procedure that transmits messages to nodes on the ring network.

trap

An M680x0 instruction that generates a hardware exception which changes the normal flow of program execution. Traps include SVC traps and traps to the PROM (MD).

trap page

The main memory page that contains the vector addresses that the processor hardware uses to handle hardware exceptions and interrupts.

trouble bit

See salvaged flag.

type

The set of operations that can be performed on an object. An object's type defines how users relate to the object. Two objects are of the same type if they share the same operations.

type manager

The manager that is responsible for interpreting the bits within an object.

typed objects

Objects that are stamped with a file type identifier that declares the writer's intention for the file. In the AEGIS system, it is the type identifier, not the object's name, the determines how the file is to be used.

unbinding a process

The operation that deletes level 1 process context. Any time that a level 2 process is deleted, its level 1 context is deleted as well.

unique identifier (UID)

A 64-bit string that uniquely identifies an object. The UID is the AEGIS system's internal name for an object.

user address space

The per-process virtual address space assigned to a process. The AEGIS system separates user address space into unprotected and protected portions called user private address space and supervisor private address space respectively. Also called per-process address space.

user environment

The collection of programs, referred to as commands, that make up the DOMAIN command line interpreter (the shell).

user environment initialization program (ENV)

The system initialization program that initializes the first process user environment and loads the display manager, the server process manager, or the single-user shell to run in that process.

user mode

The unprotected operating mode of an M680x0 processor.

user-mode code

Software that runs in M680x0 user operating mode. User-mode code refers both to user-written programs and to the unprotected portions of the AEGIS system.

user-mode process

A process that is using a user stack and thus has an active user stack pointer (USP).

user global address space

The unprotected virtual address space that is shared among all the user-mode processes on the system. User global address space contains global libraries and unprotected shared data. Also called Global A.

user private address space

The region of virtual memory that is assigned to a user process when it is created and which contains the process's private programs and data. Also called per-process private address space.

user socket

A socket that a user program allocates for its exclusive use by calling the IPC_$ interface. The IPC_$interface calls the message interface (MSG) to allocate the socket.

user stack

The stack that a process uses when it runs in user mode.

user stack pointer (USP)

The M680x0 processor register that points to a per-process user stack.

virtual address space

The virtual memory that level 1 and 2 processes use to map objects, store data, and run programs.

virtual memory

The simultaneous sharing of main memory among many users and the treatment of disk storage as a direct extension of main memory.

volume initialization utility (INVOL)

The AEGIS program that builds the physical and logical disk structures onto a disk volume.

volume mount/dismount manager (VOLX)

The manager that handles removable private storage volumes.

volume table of contents (VTOC)

The table on each logical volume that describes the objects that exist on that volume. The volume table of contents provides all the information the system needs to locate the disk blocks of an object given its UID.

volume table of contents block (VTOC block)

A hash bucket of five volume table of contents entries (VTOCEs). Each VTOC block represents one value of hash; that is, an object's UID hashes to a particular VTOC block. If an object is created and its UID hashes to a VTOC block whose VTOCEs are already full, the VTOC manager handles the overflow by creating a VTOC extension block and chaining it to the original VTOC block.

volume table of contents entry (VTOCE)

The disk volume data structure that describes a single object. The VTOCE stores the object's attributes and provides a "road map" to the disk blocks that contain the object's pages.

volume table of contents extent (VTOC extent)

A range of contiguous VTOC blocks. The volume initialization utility splits the VTOC into 1 to 8 extents which the VTOC map subsequently describes.

volume table of contents header (VTOC header)

The structure in the logical volume lable that describes the VTOC.

volume table of contents index (VTOCX)

The field in a mapped segment table entry (MSTE) or an active segment table entry (ASTE) that points directly to an object's VTOCE.

volume table of contents (VTOC) manager

The AEGIS manager that maintains the volume table of contents for the disk volume.

volume table of contents map (VTOC map)

The part of the VTOC header that provides the information needed to locate all the VTOC blocks. The VTOC map describes all the VTOC extents on a logical volume.

waiter node

The data stucture that the level 1 eventcount manager places on a process's supervisor stack when it initiates a wait for an eventcount. The waiter node contains the value for which the process is waiting, a pointer to the process's PCB, and links to other processes waiting on the same eventcount.

well-known socket

A socket that the system allocates at initialization to the higher-level network support servers available in every node. The socket ID for a well-known socket is the same on every node.

whole cloth page

A wired portion of virtual address space that has no disk storage behind it. Many of the system data structures occupy whole cloth pages; for example, the AST.

wired memory

Virtual address space that cannot be paged out of physical memory and thus needs no backing storage. Portions of the AEGIS kernel are wired; this wired code and data resides in the first section of the OS paging file.

wired RFC page

A page that contains the code and data brought into physical memory from the RFC file during the system bootstrap sequence.

wiring a page

The operation that touches object pages and also makes them permanently resident in physical memory.

Index

Chapter 2

Memory Organization and Management

Introduction and Overview

This chapter describes the hardware architectures of the memory organization and memory management schemes, as they are implemented in DN330, DN560, and DSP90 nodes. For information regarding the Motorola 68020 microprocessor, refer to the *Motorola M68020 32-bit Microprocessor User's Manual* (©1984, Motorola Inc).

Memory Organization

Besides mapping virtual-to-physical addresses, the memory management unit manages data and program storage in DN330, DN560, and DSP90 nodes. In order to understand the memory management unit's hardware architecture, you must first understand how virtual and physical space are organized within these nodes.

Virtual Space

See Figure 2–1. It depicts virtual address space allocation in DN330, DN560, and DSP90 nodes.

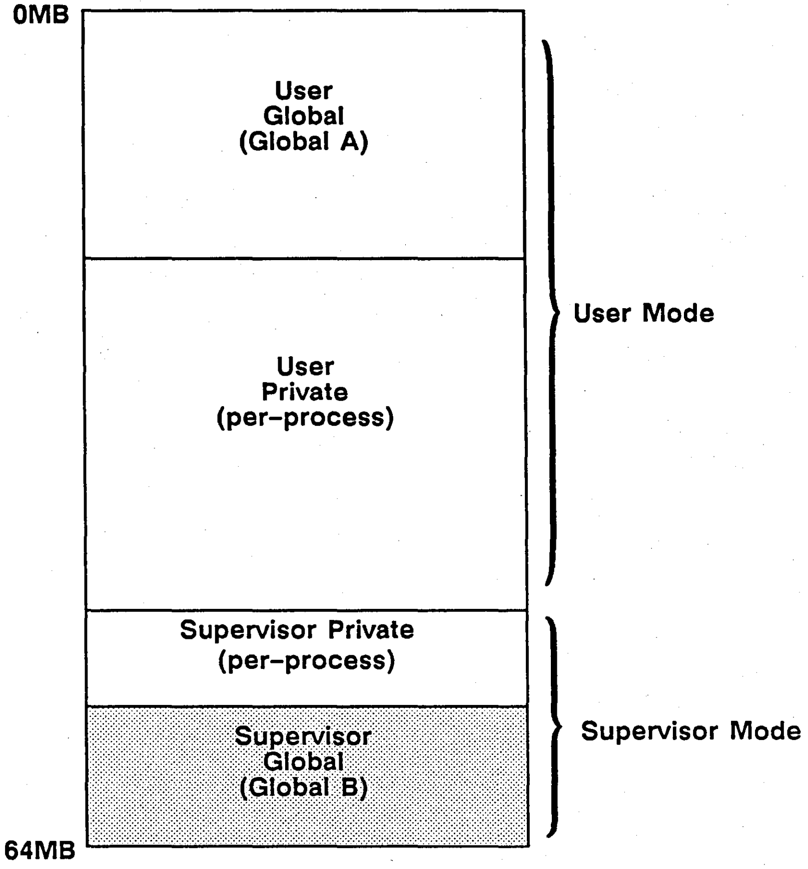

0MB
User Global (Global A)
User Private (per-process)
User Mode
Supervisor Private (per-process)
Supervisor Global (Global B)
Supervisor Mode
64MB

Figure 2–1. Virtual Address Space in DN330, DN560 and DSP90 Nodes

Once you are familiar with the Figure, refer to Table 2–1, Table 2–2, and Table 2–3. These tables provide listings of virtual addresses for devices that are used by the operating system.

Table 2-1. A Virtual Memory Map for DN330 Nodes (Running the AEGIS Operating System)

Location in Virtual Space (Global or User)	Virtual Address	Name
Global A	400	Boot PROM
Global B	3fc0000	Display1 Memory
	3ff9c00	Ring
	3ff9800	Display1 Control
	3ffa000	DMA
	3ffa800	Floppy
	3ffac00	Timers
	3ffb000	SIO
	3ffb400	MMU
	3ffb800	PFT

Table 2-2. A Virtual Memory Map for DN560 Nodes (Running the AEGIS Operating System)

Location in Virtual Space (Global or User)	Virtual Address	Name
Global A	400 4000	Boot PROM Boot PROM2
Global B	3fa0000 3fc0000 3fe0000 3ff5000 3ff6000 3ff7c00 3ff9400 3ff9a00 3ff9800 3ff9c00 3ffa800 3ffb000 3ffb400 3ffb800	Color Memory Display1 Memory PBU I/O I/O Map Color Superv. PBU Control VME Display1 User Display1 Superv. Ring2 Disk/Tape Cal SIO MMU PFT

Table 2-3. A Virtual Memory Map for DSP90A Nodes (Running the AEGIS Operating System)

Location in Virtual Space (Global or User)	Virtual Address	Name
Global A	400	Boot PROM
Global B	3fe0000	PBU I/O
	3ff5000	I/O Map
	3ff7c00	PBU Control
	3ff8000	Line Printer
	3ff9c00	Ring
	3ffb000	SIO
	3ffb400	MMU
	3ffb800	PFT

Physical Space

See Figure 2-2. It depicts physical space allocation in DN330, DN560, and DSP90 nodes.

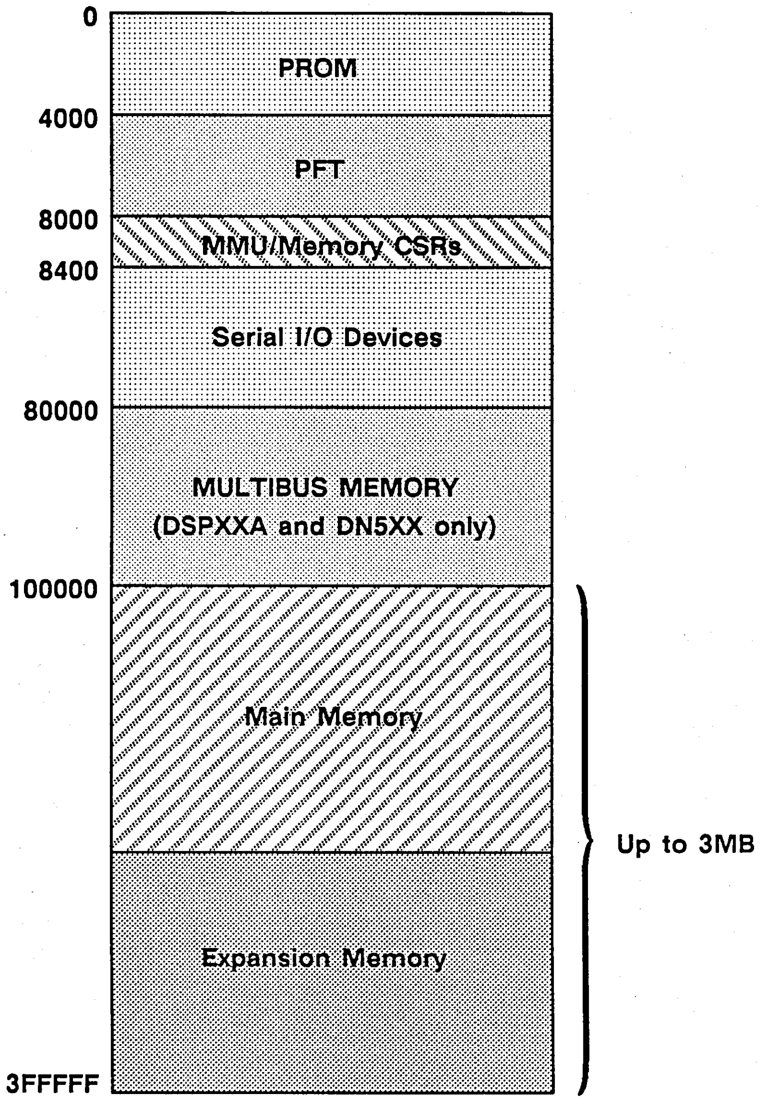


Figure 2–2. Physical Address Space in DN330, DN560 and DSP90 Nodes

Once you are familiar with the Figure, refer to Table 2–4, Table 2–5, and Table 2–6. These tables provide listings of physical addresses for main memory and devices resident in physical space.

Table 2-4. A Physical Memory Map for DN330 Nodes (Running the AEGIS Operating System)

Physical Address	Name
400	Boot PROM
20000	Display1 Memory
9800	Ring
9000	DMA
9c00	Floppy
8800	Timers
8400	SIO
8000	MMU
4000	PFT

Table 2-5. A Physical Memory Map for DN560 Nodes (Running the AEGIS Operating System)

Physical Address	Name
400	Boot PROM
14000	Boot PROM2
40000	Color Memory
20000	Display1 Memory
70000	PBU I/O
10000	I/O Map
e000	Color
a400	PBU Control
bc00	VME
f000	Display1 User
f400	Display1 Superv.
9800	Ring2
9c00	Disk/Tape Cal
8800	Timers
8400	SIO
8000	MMU
4000	PFT

Table 2-6. A Physical Memory Map for DSP90 Nodes (Running the AEGIS Operating System)

Physical Address	Name
400	Boot PROM
70000	PBU I/O
10000	I/O Map
a400	PBU Control
a800	Line Printer
bc00	VME
9800	Ring
9000	DMA
a000	Calendar
9c00	Disk/Tape Cal
8400	SIO
8800	Timers
8000	MMU
4000	PFT

The Memory Subsystem

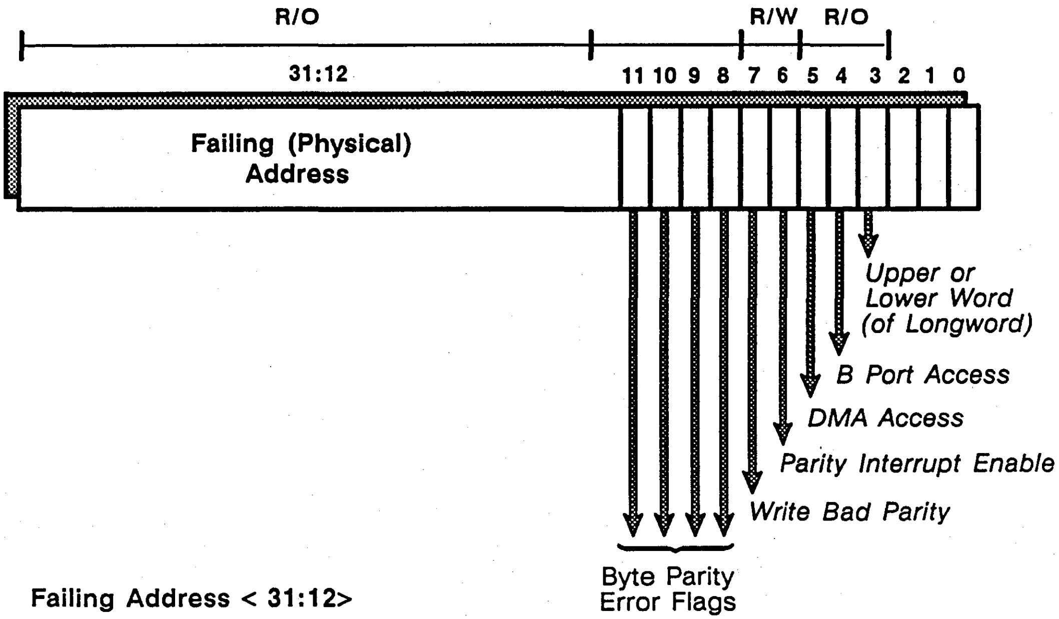
Refer to Table 2–7. It describes those physical memory configurations that are available for DN330, DN560, and DSP90 nodes.

Table 2-7. Physical Memory Configurations Available for DN330, DN560, and DSP90 Nodes

NODE	CPU MAIN MEMORY	EXPANSION MEMORY AVAILABLE
DN330	*2.0 MB (256K RAMS)*	*1.0 MB (64K RAMS)*
DN560	*2.0 MB (256K RAMS)*	*1.0 MB (64K RAMS)*
DSP90	*2.0 MB (256K RAMS)*	*1.0 MB (64K RAMS)*
Actual address space size is 4MB; 3MB are used for main memory, and 1MB is used (reserved) for I/O space.		

The Memory Control/Status Register

See Figure 2-3. It shows and describes the contents of the memory control/status register.



Failing Address < 31:12>

This field contains the physical address of the location where a parity error has occurred.

Byte Parity Error Flags < 11:8>

1 = set, 0 = clear.

Write Bad Parity < 7 >

This bit forces any memory write to generate a bad parity bit in memory.

Parity Interrupt Enable < 6 >

This bit enables an MC68020 level-7 interrupt, when a memory parity error occurs.

DMA Access < 5 >

This bit is set if a DMA access was in progress at the time a memory parity error occurred.

B-Port Access < 4 >

This bit is set if a B-Port access (a memory access from a VMEbus device) was in progress at the time a memory parity error occured.

Upper or Lower Word < 3 >

This bit designates bit 1 in the address bus when a parity error has occurred. It is valid only for B-Port accesses, since the B-Port interface bus is only 16 bits wide.

Figure 2-3. The Memory Control/Status Register

Memory Parity Checking

The memory subsystem calculates parity on each byte written; on memory reads, it checks all four parity bits. When a memory parity error occurs, the memory cycle completes, the longword address where the parity error occurred is frozen in the memory control/status register, and a level 7 (the highest priority) interrupt is generated. Refer to the *Motorola M68020 32-bit Microprocessor User's Manual* (©1984, Motorola Inc) for information regarding interrupts. To see the memory control/status register, refer back to Figure 2–3.

The Memory Management Unit

The memory management unit's hardware functionality is closely tied with that of software. Consequently, we often need to discuss software (and, sometimes, implementation) issues, in order to provide you with a cohesive picture of the MMU. Remember that this is a reverse– (rather than forward–) mapped memory management unit. Reverse–mapping means that the MMU's dynamic address translation (DAT) hardware checks virtual addresses *against the way physical memory is (currently) mapped*, in order to perform each virtual–to–physical address translation.

Memory Cycles

During a memory cycle, the Motorola 68020 microprocessor presents a 26–bit address to the MMU. The microprocessor also emits one of eight 3–bit function codes (five of which are used) and a read/write signal; these explain what type of memory access (user or supervisor read, write, or execute) the CPU is attempting to perform. Memory management hardware can then respond in one of two modes. These modes are:

- *mapped* (where virtual–to–physical address translation takes place), and
- *unmapped* (where virtual addresses are equivalent to physical addresses; this mode is neccessary for bootstrapping, and useful for debugging and diagnostics applications).

Refer to Figure 2–4 to see the format for a virtual address as it is passed from the 68020 to the MMU.

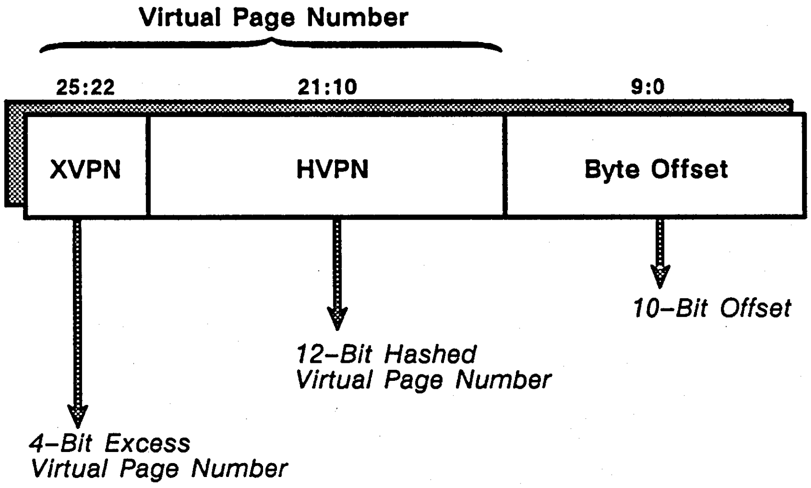


Figure 2-4. A Virtual Address

The MMU in Mapped ("Virtual") Mode

In mapped (MMU enabled) mode, the MMU translates the 26-bit virtual address into a 22-bit physical byte address. The CPU can then access physical memory or device registers. In mapped mode, the MMU performs address space and access permissions checking.

The MMU in Unmapped ("Physical'') Mode

In unmapped (MMU disabled) mode, the MMU passes the 22 least significant bits from the address directly to the physical address bus. The CPU can then access physical memory or device registers. In unmapped mode, the MMU does not perform address space or access permissions checking.

Figure 2-5 shows the path of an address through the MMU, where the MMU is in both mapped and unmapped modes.

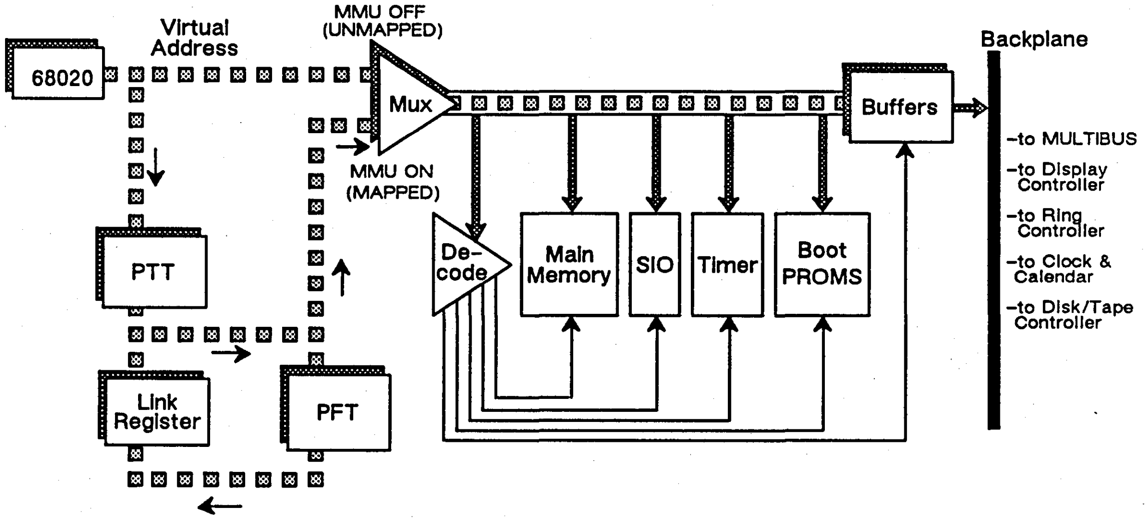


Figure 2–5. A Virtual Address' Path through the MMU (where the MMU is in Mapped and Unmapped Modes)

Before Address Translation Occurs

A virtual address divides into two sections. These are: the Virtual Page Number (VPN) and the Offset. The Virtual Page Number comprises two fields. These are: the Hashed Virtual Page Number (HVPN), and the Extended Virtual Page Number (XVPN). When the MMU is in mapped mode, it must "look at" the virtual page number (i.e., at the XVPN and the HVPN fields). Refer again to Figure 2–4, to examine the fields within a virtual address.

Besides the virtual page number, the MMU must also "look at" the Address Space IDentifier for the current process (from the ASID register, discussed in next section). Without these elements, the MMU cannot begin translating a virtual–to–physical address for any process.

Memory Management Tables and Registers

In order to understand the hardware architecture of the reverse–mapped MMU, it's important to understand how its individual registers and tables function. These registers and tables are:

- The Page Frame Table (PFT)
- The Page Translation Table (PTT)
- The MMU ASID Register
- The MMU Control Register
- The MMU Status Register
- The MMU Parity Register

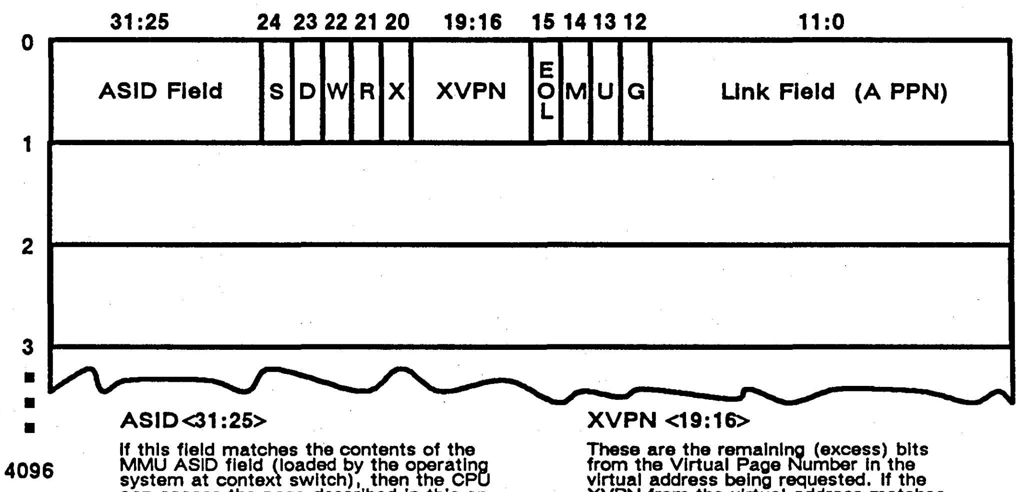
In the text that follows, we show and describe each table and register in detail. When you finish reading the descriptions, see Figure 2–16. It provides a flowchart, illustrating how the MMU, along with its registers and tables, operates.

The Page Frame Table

The Page Frame Table contains 4096 32–bit physical page descriptors, or entries. Entries in the PFT are indexed (pointed to) by physical page number (PPN). Each PFT entry corresponds to a single 1K, or 1024 byte physical page (i.e., one page frame). Those PFT entries that share a common hashed virtual page number are threaded (linked) together. Software sets a "link mark," or End–Of–List bit, in one (any, arbitrary) entry per linked list.

Whenever the operating system brings new pages into physical memory, or, whenever it takes existing pages out, it modifies (re– "maps") the contents of the PFT. Consequently, the table always presents an accurate snapshot of the way physical memory is *currently* mapped.

Refer to Figure 2–6; it shows the Page Frame Table, and the format of a PFT entry (PFTE). While you are looking at Figure 2–6, notice the way in which some PFT entries are linked.



ASID<31:25>

If this field matches the contents of the MMU ASID field (loaded by the operating system at context switch), then the CPU can access the page described in this entry.

Note: If the Global (G) bit (bit 12) is set in the PFTE, it disables this comparison.

Supervisor <24>

The operating system sets this bit if the PFTE describes a page that can be referenced only by the Supervisor (and not by the User).

DOMAIN <23>

This bit is unused.

Write <22>

The operating system sets this bit if the current process can write to the page that this PFTE describes.

Read <21>

The operating system sets this bit if the current process can have read access to the page that this PFTE describes.

eXecute <20>

The operating system sets this bit if the current process is allowed to execute instructions from the page that this PFTE describes.

XVPN <19:16>

These are the remaining (excess) bits from the Virtual Page Number in the virtual address being requested. If the XVPN from the virtual address matches the XVPN in this PFTE, then the PFTE describes the requested page.

End-of-list (EOL) <15>

This bit marks an arbitrary point in each linked list within the PFT. During a search through the PFT, hardware looks for an address match OR to see the end-of-list bit twice. If a match has not been found by the time hardware sees the bit twice, a bus error occurs.

Modified <14>

The MMU sets this bit to tell the operating system that an access has occurred to the page described by this PFTE.

Accessed <13>

The MMU sets this bit to tell the operating system that an access has occurred to the page described by this PFTE.

Global <12>

The operating system sets this bit in the PFTE so that different processes can reference the same ("global") page in virtual memory.

Note: when set, this bit precludes the need for an ASID check.

Link <11:0>

This field serves as a pointer to the next entry in a linked (but not neccesarily contiguous) list of entries; it contains the physical page number of the next entry whose hashed virtual page number is the same as its own.

Figure 2–6. The Page Frame Table (with the Format of a Page Frame Table Entry Shown)

Once you are familiar with Figure 2–6, refer to Figure 2–7. It shows a PFT entry again, but this time, notice that we have re–arranged the entry into logical blocks. The logical blocks within each PFT entry provide:

- MMU **hash management information,** via the **XVPN** and **LINK** fields,
- MMU **address space checking information,** via the **ASID** field and the Global bit,
- MMU **access permissions information,** via the Write, Read, eXecute, and Supervisor bits, and
- O/S **page statistics information,** via the Modified, and Used bits. (Software will use these statistics bits for the page replacement algorithm that keeps the most recently used pages resident in main memory, and for other page management tasks.)

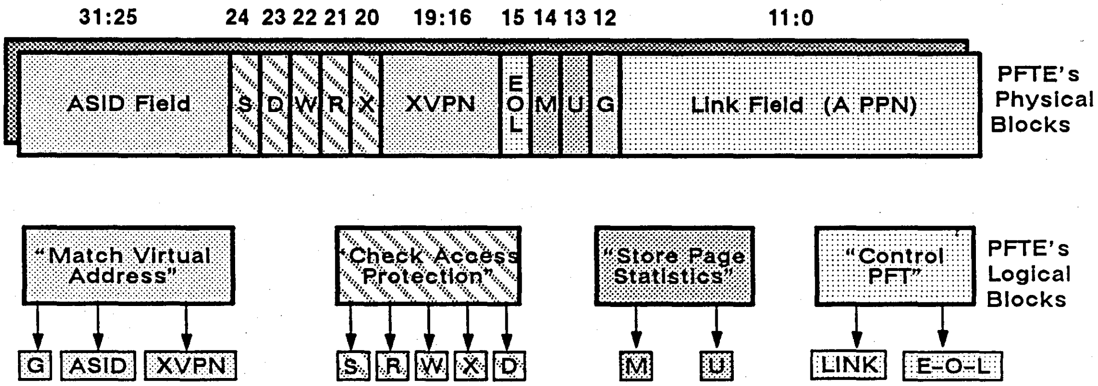


Figure 2–7. A Page Frame Table Entry, Re–arranged into Logical Blocks

The Page Translation Table

The Page Translation Table contains 4096 12–bit entries. It is a cache; it provides a speedy (hashed) lookup into the PFT. Each Page Translation Table entry (PTTE) comprises a 12–bit PPN. This PPN points to one entry in a circular, linked list of PFT entries. All of the linked PFT entries hash to a common PTT entry (i.e., their hashed virtual page numbers are the same). Refer to Figure 2–8; it shows the Page Translation Table and the format of each PTT entry (PTTE).

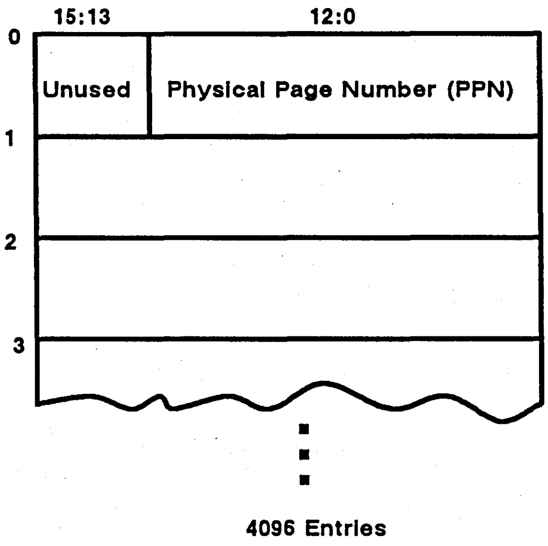


Figure 2–8. The Page Translation Table (with the Format of a Page Translation Table Entry Shown)

Refer to Figure 2–9. It depicts the relationship between the Page Frame Table (the descriptor file), and the Page Translation Table (an index to the descriptor file).

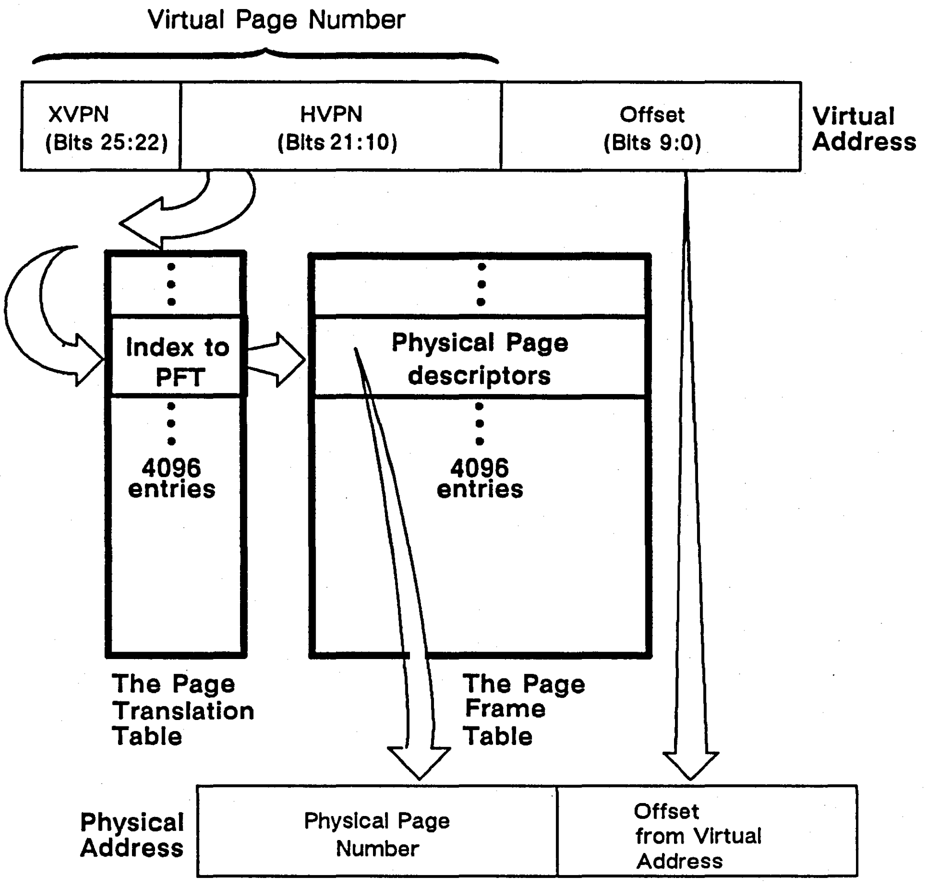


Figure 2-9. The Relationship of the Page Translation Table to the Page Frame Table

The PTT must be dynamically updated, so that it will provide a current index to the PFT. In order to update or initialize the PTT, the operating system -- or the bootstrap loader -- must be able to access (write to) it. Since the PTT cannot be used to access itself via normal, virtual-to-physical address translation, the PTT is assigned a fixed address range ($400000 - $800000).

The PTT is enabled via the **PTT access enable bit** (bit 1) in the MMU control register. When this bit is set, software can access (write to, and update) the PTT. When the bit is not set, however, software cannot access the PTT. Only when the PTT is enabled, and a reference is made to address space range $400000 – $800000, can the PTT can be accessed by the operating system.

If the MMU is enabled (in mapped mode), and the PTT is disabled, address space range $400000 – $800000 is treated as part of per–process virtual address space. If the MMU is disabled, and the PTT is disabled, address space range $400000 – $800000 is not used. This is because in the unmapped mode, the MMU is simply passing the 22 least significant bits of each virtual address directly to the physical address bus. Refer to Table 2–8. It shows the relationship of an adddress to the MMU and PTT. Then, refer to Figure 2–10. It depicts the PTT, enabled and disabled (for the operating system's, or for the bootstrap loader's purposes) within virtual address space.

Table 2–8. The Relationship of an Address to the MMU and PTT

Location of Virtual Address	MMU On PTT On	MMU On PTT Off	MMU Off PTT Off	MMU Off PTT On
400000 – 800000 (Within PTT Range)	"Address PTT" *(Access PTT instead of memory, using V.A.)*	"Translate Virtual Address" *(Perform normal Virtual–to–Physical translation, and access memory)*	"Physical Address" *(Virtual Address is the same as Physical Address)*	"Address PTT" *(Access PTT instead of memory, using V.A.)*
0 – 3FFFFC or 800000 – 3FFFFFC (Not within PTT Range)	"Translate Virtual Address" *(Perform normal Virtual–to–Physical translation, and access memory)*	"Translate Virtual Address" *(Perform normal Virtual–to–Physical translation, and access memory)*	"Physical Address" *(Virtual Address is the same as Physical Address)*	"Physical Address" *(Virtual Address is the same as Physical Address)*

Virtual Mode　　Physical Mode

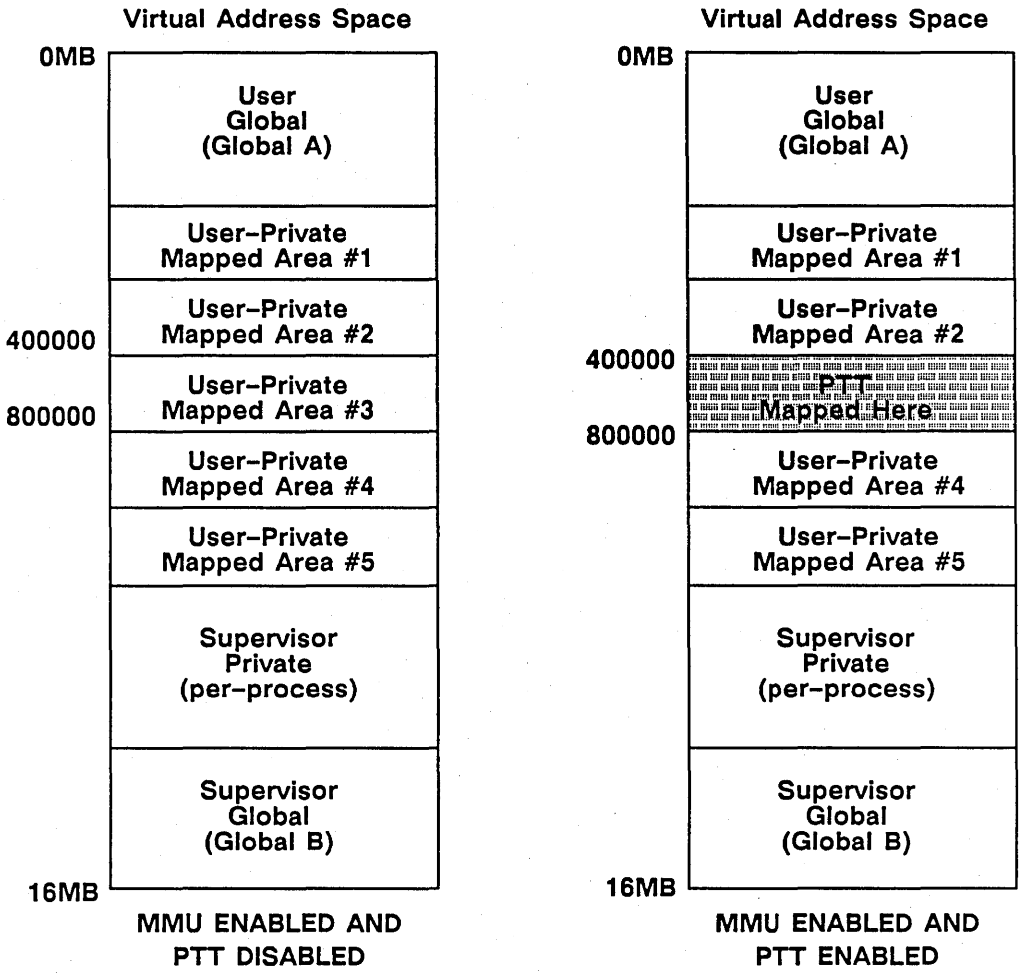

Virtual Address Space
0MB
User Global (Global A)
User–Private Mapped Area #1
User–Private Mapped Area #2
400000
User–Private Mapped Area #3
800000
User–Private Mapped Area #4
User–Private Mapped Area #5
Supervisor Private (per–process)
Supervisor Global (Global B)
16MB
MMU ENABLED AND PTT DISABLED
Virtual Address Space
0MB
User Global (Global A)
User–Private Mapped Area #1
User–Private Mapped Area #2
400000
PTT Mapped Here
800000
User–Private Mapped Area #4
User–Private Mapped Area #5
Supervisor Private (per–process)
Supervisor Global (Global B)
16MB
MMU ENABLED AND PTT ENABLED

Figure 2–10. The PTT (Enabled and Disabled) within Virtual Address Space

The ASID Register

One binary number (of 128) is assigned to each per–process address space as each new process is created; this number is the ASID for the current process. At process (or "context") switch, the ASID is loaded into the ASID register, and presented to the MMU. See Figure 2–11; it is designed to help you understand the concept of Address Space

IDentification. Then, see Figure 2–12. It shows and describes the contents of the ASID register.

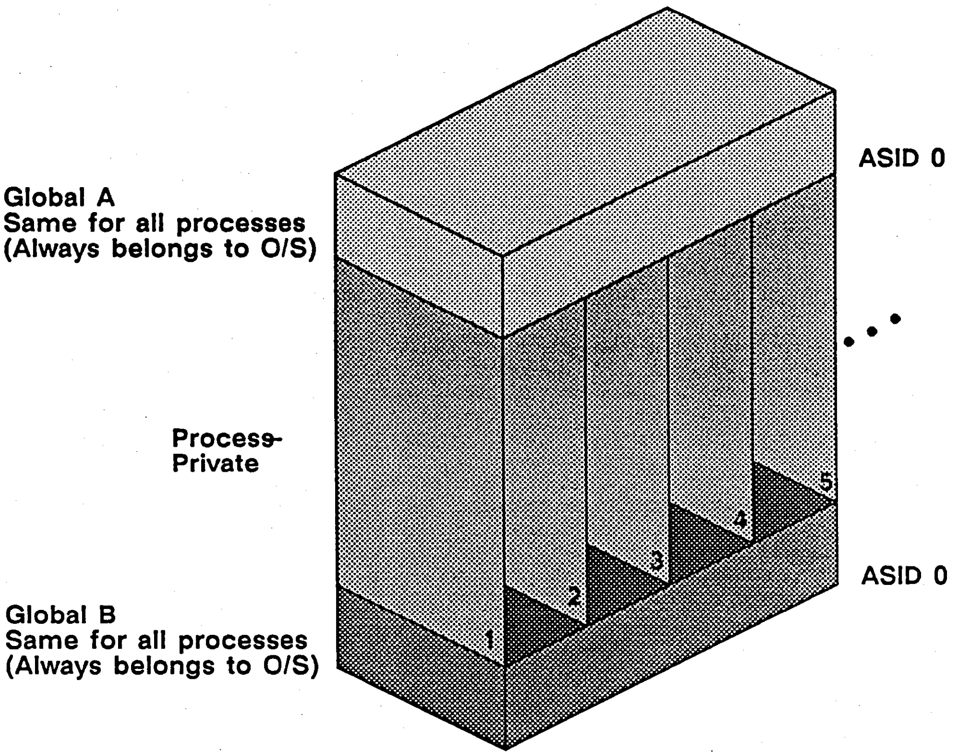
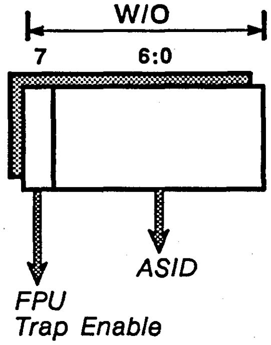


Figure 2–11. Address Space Identification



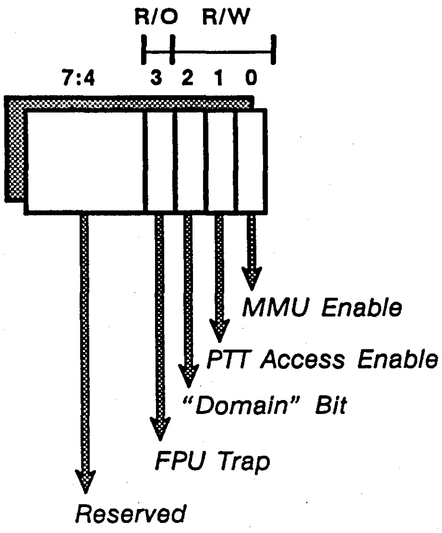
FPU Trap Enable < 7 >
0 = trap next FPU chip access
1 = allow FPU chip access

ASID < 6:0 >
This field identifies the process that is currently running.

Figure 2–12. The ASID Register

The MMU Control Register

CPU writes to this register control MMU operations. See Figure 2–13; it shows and describes the functions of the MMU control register.



MMU Enable < 0 >

When this bit is set, physical addresses are formed using the contents of Page Translation and Page Frame tables. This bit also enables access rights checking, page fault traps, and page statistics updates. When reset (to MMU disable), this bit causes the low-order bits of the virtual address to be passed to the physical address bus, unchanged.

PTT Access Enable < 1 >

When this bit is set, references to those addresses within virtual address range $4000000–$8000000 a software to read or write to (update) the PTT.

DOMAIN Bit < 2 >

This bit is presently unused.

FPU Trap < 3 >

This bit is presently unused.

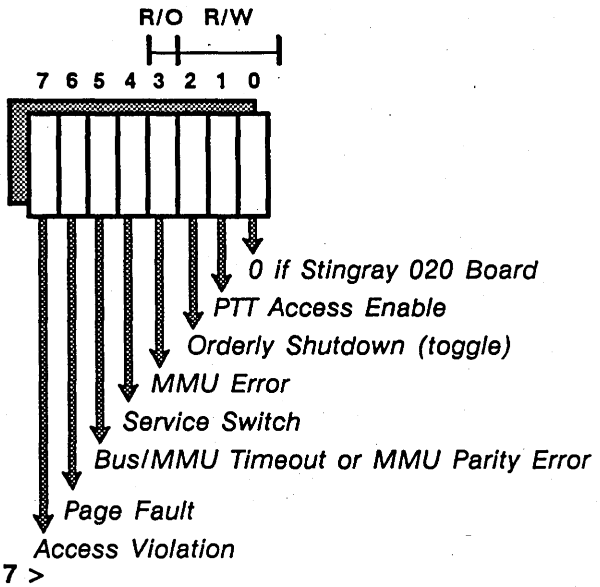
Reserved < 7:4 >

These bits are reserved.

Figure 2-13. The MMU Control Register

The MMU Status Register

The CPU reads the MMU status register in order to monitor MMU operations. Refer to Figure 2–14; it shows and describes the functions of the MMU status register.



Access Violation < 7 >

When this bit is set, it indicates that although the MMU has found a PFT entry with the correct ASID XVPN bits, the entry has failed to pass an access protection check. Writes to this register clear the bit.

Page Fault < 6 >

This bit indicates that the MMU has passed the end-of-list bit twice, in its search for a "correct" PFTE. Writes to this register clear the bit.

Bus/MMU Timeout or MMU Parity Error < 5 >

This bit is set when the MMU times out, the bus times out, or an MMU parity error occurs. Writes to this register clear the bit.

Service Switch < 4 >

This bit indicates the state of the node's service switch. 1 = Normal Mode; 0 = Service Mode.

MMU Error < 3 >

This bit is set when a fault in the MMU causes a bus error.

Orderly Shutdown < 2 >

This bit indicates the state of the orderly shutdown switch. The bit is normally a "1," but it goes to "0" when a shutdown is requested.

PTT Access Enable < 1 >

This bit reflects the state of the PTT Access Enable bit (bit 1) in the MMU Status Register.

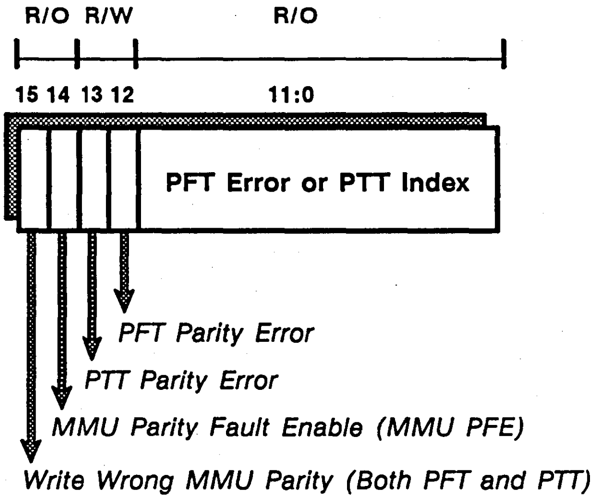
020 Board < 0 >

This bit is wired low ("0") so that the O/S can distinguish an MC68020-based CPU board from an MC68010-based one.

Figure 2–14. The MMU Status Register

The MMU Parity Register

The MMU checks for parity in the Page Frame and Page Translation Tables. Even parity is always used to detect the failure of memory outputs going to a high (logical 1) value. The MMU updates the two page statistics bits in each PFT entry whenever an access is made to the corresponding physical page. These are the **Used bit** (bit 13) and the **Modified bit** (bit 14). As a result, these bits are not used in parity calculation, and parity is calculated only over the remaining 30 bits. See Figure 2–15; it shows and describes the functions of the MMU parity register.



Write Wrong MMU Parity < 15 >

Diagnostics use this bit to test the MMU's error detection hardware. When the bit is set, any data written to the Page Translation Table or the Page Frame Table is written with odd parity instead of even, and parity errors will occur.

MMU Parity Fault Enable < 14 >

When this bit is cleared, MMU parity errors are ignored.

PTT Parity Error < 13 >

This bit is set when a parity error is detected within the PTT.

PFT Parity Error < 12 >

This bit is set when a parity error is detected within the PFT.

PFT Error or PTT Index < 11:0 >

This field designates the location where a parity error has occurred. If the error was in the PFT, this field contains the partial contents (a PPN) of the faulty entry. If the error was in the PTT, this field contains the index (a PPN) of the faulty entry.

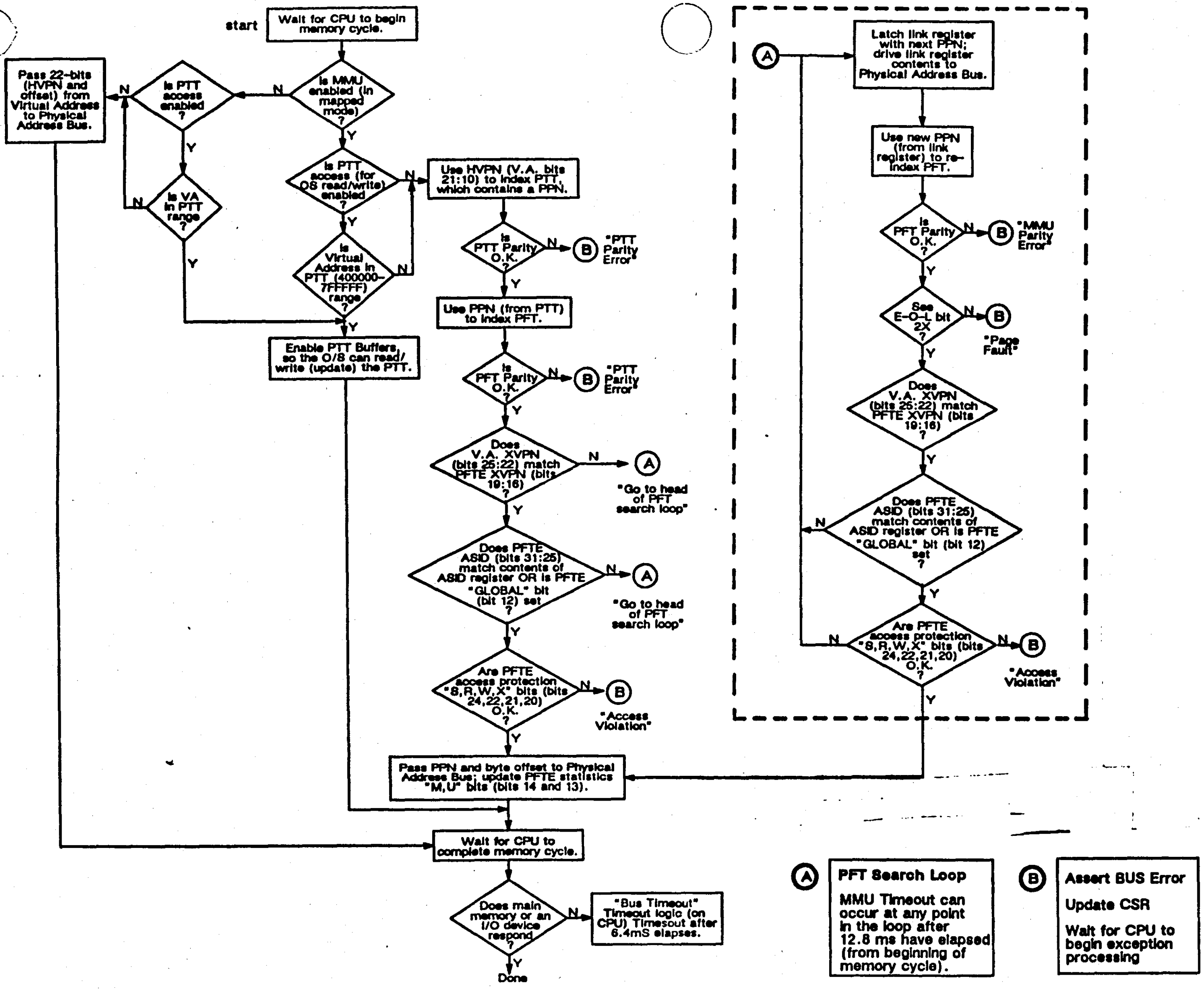
Figure 2–15. The MMU Parity Register

MMU Operations

In the text that follows, we discuss MMU processes, and we explain how each of the previously described registers and tables are utilized within the MMU, when it is in mapped mode. Refer at this time to Figure 2–16. It provides a flowchart of MMU mapped mode operations. When you have familiarized yourself thoroughly with the flowchart, go on to the text that follows it.

See
insert
next
pg →

Figure 2-16. MMU Operations (in Mapped Mode)


start
Wait for CPU to begin memory cycle.
Is MMU enabled (in mapped mode) ?
N
Y
Is PTT access enabled ?
N
Y
Pass 22-bits (HVPN and offset) from Virtual Address to Physical Address Bus.
Is VA in PTT range ?
N
Y
Is PTT access (for OS read/write) enabled ?
N
Y
Is Virtual Address in PTT (400000-7FFFFF) range ?
N
Y
Enable PTT Buffers, so the O/S can read/write (update) the PTT.
Use HVPN (V.A. bits 21:10) to index PTT, which contains a PPN.
Is PTT Parity O.K. ?
N
B
"PTT Parity Error"
Y
Use PPN (from PTT) to index PFT.
Is PFT Parity O.K. ?
N
B
"PTT Parity Error"
Y
Does V.A. XVPN (bits 25:22) match PFTE XVPN (bits 19:16) ?
N
A
"Go to head of PFT search loop"
Y
Does PFTE ASID (bits 31:25) match contents of ASID register OR is PFTE "GLOBAL" bit (bit 12) set ?
N
A
"Go to head of PFT search loop"
Y
Are PFTE access protection "S,R,W,X" bits (bits 24,22,21,20) O.K. ?
N
B
"Access Violation"
Y
Pass PPN and byte offset to Physical Address Bus; update PFTE statistics "M,U" bits (bits 14 and 13).
Wait for CPU to complete memory cycle.
Does main memory or an I/O device respond ?
N
"Bus Timeout" Timeout logic (on CPU) Timesout after 6.4mS elapses.
Y
Done
A
Latch link register with next PPN; drive link register contents to Physical Address Bus.
Use new PPN (from link register) to re-index PFT.
Is PFT Parity O.K. ?
N
B
"MMU Parity Error"
Y
See E-O-L bit 2X ?
N
B
"Page Fault"
Y
Does V.A. XVPN (bits 25:22) match PFTE XVPN (bits 19:16) ?
Y
Does PFTE ASID (bits 31:25) match contents of ASID register OR is PFTE "GLOBAL" bit (bit 12) set ?
N
Y
Are PFTE access protection "S,R,W,X" bits (bits 24,22,21,20) O.K. ?
N
N
B
"Access Violation"
Y
A
PFT Search Loop
MMU Timeout can occur at any point in the loop after 12.8 ms have elapsed (from beginning of memory cycle).
B
Assert BUS Error
Update CSR
Wait for CPU to begin exception processing

Successful Translations

Every memory (CPU) cycle results in one of two occurances, either of which can result in a successful translation. These are:

- a PTT hit, or
- a PTT miss.

A PTT Hit

The fields within the virtual page number and the contents of the ASID register must be checked against the contents of specific Page Frame Table entries, pointed to by the Page Translation Table. A PTT hit occurs when:

1. **The XVPN** field, located in the PFT entry pointed to by the PTT, **matches the XVPN** field that resides within the virtual page number (see Figure 2–17), and

2a. **The ASID** field, located in the PFT entry pointed to by the PTT, **matches the ASID** field that resides within the MMU's ASID register, for the current process (see Figure 2–18),

or

2b. The Global bit **is** set in the PFTE. Refer again to Figure 2–6 in order to see the address space control (ASID and Global bit) fields in the PFTE.

NOTE: When the Global bit is set, it obviates the need for checking the ASID field. The Global bit set indicates that the address space in which this data (or program) resides is global; data (or programs) in this space are shared by all processes (regardless of their respective ASIDs).

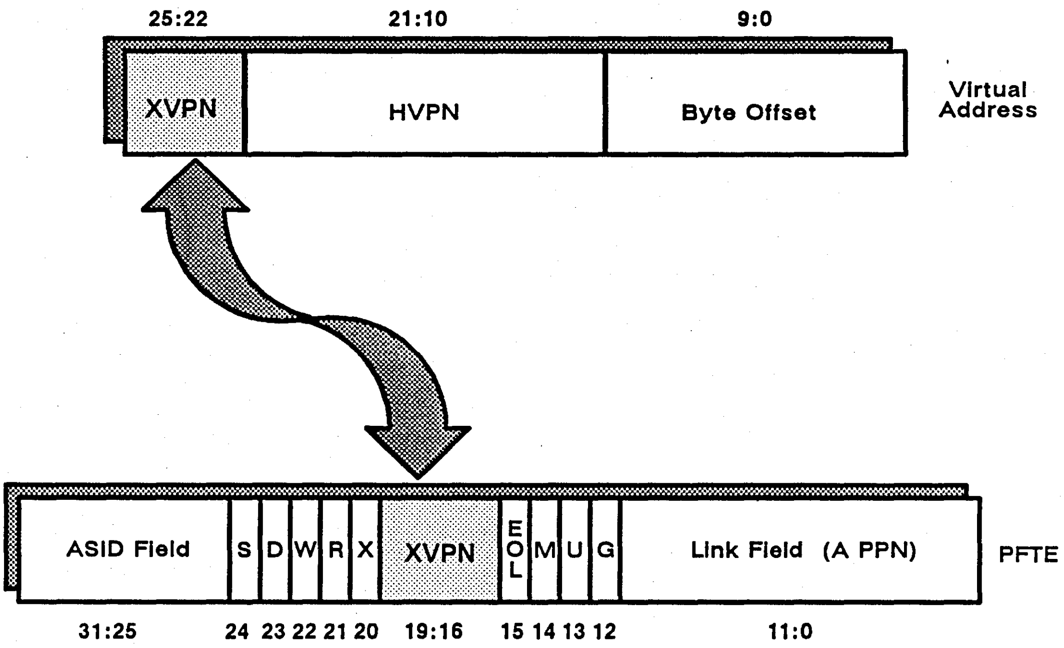


Figure 2-17. Comparing the XVPN Fields

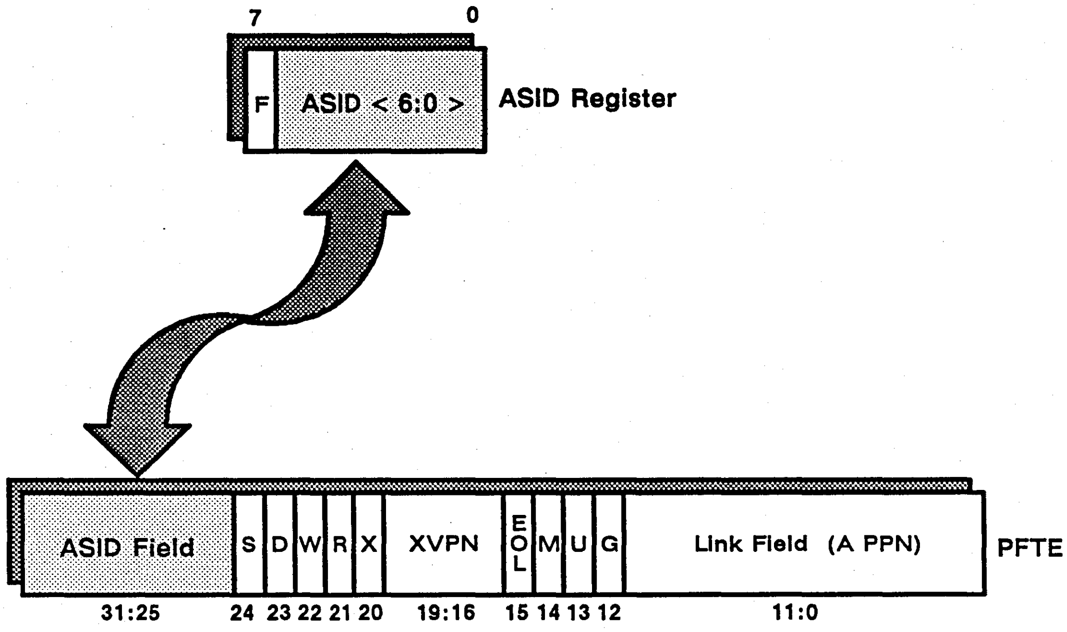


Figure 2–18. Comparing the ASID Fields

Once a PTT hit has occurred, the MMU checks the function code emitted by the 68020 against the way the access permissions bits (Read, Write, eXecute, Supervisor) have been set in the PFT entry pointed to by the PTT. Refer again to Figure 2–6 in order to see the access permissions bits in the PFT. Then, refer to Figure 2–19 to see the function code and access permissions bits comparison.

FC	R/W		PFTE Bits																
			X	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
			W	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
			S	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
			R	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1	0	User	Data	No Access Permitted								X	X	✓	✓	not checked			
	1											✓	✓	✓	✓				
2	0		Program									not checked							
	1											X	✓	X	✓				
5	0	Supervisor	Data									X	X	✓	X	X	X	✓	✓
	1											✓	✓	✓	✓	✓	✓	✓	✓
6	0		Program									not checked							
	1											X	✓	X	✓	X	✓	X	✓
7	0		CPU Space									Never mapped. Used for CPU space access. Refer to 68020 manual function codes.							
	1																		

If R & W both bits bits must be set for data writes. { R* = enables access (data read)
W = allows data writes

X = allows program reads

S = the CPU must be in Supervisor mode to access. If R and X are both set, then then execute program (do data reads).

*R must be set for X or W to occur.

Figure 2-19. Comparing the Function Code and the Access Permissions Bits

If the permissions agree, the CPU can access the memory location described in the PFTE; the MMU will update the page statistics (Modified and Used) bits appropriately in the PFTE. Refer back to Figure 2-6 in order to see the page statistics bits.

If the permissions do not agree, the CPU cannot access the memory location described in the PFTE, and the MMU generates a bus error.

A PTT Miss

The fields within the virtual page number and the contents of the ASID register must be checked against the contents of specific Page Frame Table entries, pointed to by the Page Translation Table. A PTT miss occurs when the first PFTE pointed to by the PTT is not the correct one (i.e., when steps 1 and 2a or 2b, above, do not occur). In the event of a PTT miss, the MMU proceeds around the linked list of PFT entries, searching to resolve the virtual page number. Two possibilities then exist:

3a. Steps 1 and 2a or 2b, above, *occur in a PFTE other than the one first pointed to by the PTT;* remember that this is a reverse-mapped MMU. One of its characteristics is that if the page exists in physical memory, a descriptor for it will reside in the PFT,

or

3b. Steps 1 and 2a or 2b, above, *do not occur at all;* if the page does not exist in physical memory, a descriptor for it will not reside in the PFT.

Completing Translation

If the access is permitted, and the first PFTE pointed to by the PTT provided a successful translation (i.e., if a PTT hit occurs), the MMU updates the page statistics (Modified and Used) bits appropriately in the PFTE. Refer again to Figure 2–6 in order to see the page statistics bits in the PFTE.

If the access is permitted, and a PFTE linked to the first PFTE pointed to by the PTT provided a successful translation (i.e., if a PTT miss occurs, but a translation is subsequently found), the MMU updates the page statistics (Modified and Used) bits appropriately in the PFTE. It also writes back the correct PPN (from the correct PFTE) to the PTT (i.e. the MMU "updates" the PTT). This increases the probablity that a PTT hit will occur on the next access to the same virtual page.

In either case, the MMU will concatenate the 12–bit PPN, yielded by a successful translation, with the offset from the virtual address. This forms a physical address. Refer to Figure 2–20; it shows the format of a physical address that is ready to be sent over the physical address bus.

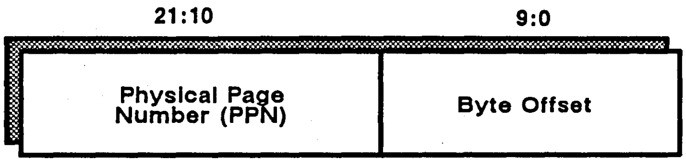


Figure 2–20. A Physical Address (note the PPN and Offset Fields)

Unsuccessful Translations

When translation is unsuccessful, a bus error (one of four types) is generated. The four types of bus errors are:

1. Page Faults,
2. Access Violations,
3. MMU Parity Errors, and
4. MMU Timeouts.

We describe each type of bus error in the text that follows.

Page Faults

If the MMU encounters the "end–of–linked–list" bit twice, there is no match (and it can be inferred that the page does not reside in physical memory). The MMU sets the **page fault bit** (bit 6) in the MMU control register, so that the operating system can bring the requested page into physical memory.

Access Violations

When the current process does not have the appropriate permissions (**R**, **W**, **X**, or **S**) to perform the function that the CPU is trying to perform, the MMU sets the **access violation bit** (bit 7) in the MMU control register.

MMU Parity Errors

When hardware detects a parity error in either the PTT or the PFT, the MMU sets the **bus timeout or MMU parity error bit** (bit 5) and the **MMU error bit** (bit 3) in the MMU status register. The MMU also sets the **PFT error bit** (bit 12) or the **PTT error bit** (bit 13) in the MMU parity register. Additionally, the MMU loads the PPN from that location where the fault has occurred into the MMU parity register's **PFT index or PTT contents** field.

MMU Timeouts

At most, the MMU needs .68ms to completely search the PFT for a valid translation. When a memory error in the PFT occurs, the link mark bit may be inappropriately cleared. If a search is not completed (a match is not found, nor is a page fault generated) within 12.8ms, the MMU sets the **bus timeout or MMU parity error bit** (bit 5), and the **MMU error bit** (bit 3) in the MMU status register.

Appendix A

Floating Point Processing

Introduction

This chapter describes the architecture of floating-point units used in DN3XX and DN5XX nodes (the DSPXXA does not incorporate floating-point capabilities). If you have a DN320 or DN550 node, floating-point operations are controlled by the Performance Enhancement Board (PEB, or floating-point accelerator). If you have a DN330 or DN560 node, floating-point operations are performed by the MC68881 coprocessor. Refer to Chapter 1, and see Figures XXX and XXX. These figures show the system architectures of DN3XX and DN5XX nodes. Notice where the PEB, or the MC68881, is located within each figure.

The Performance Enhancement Board

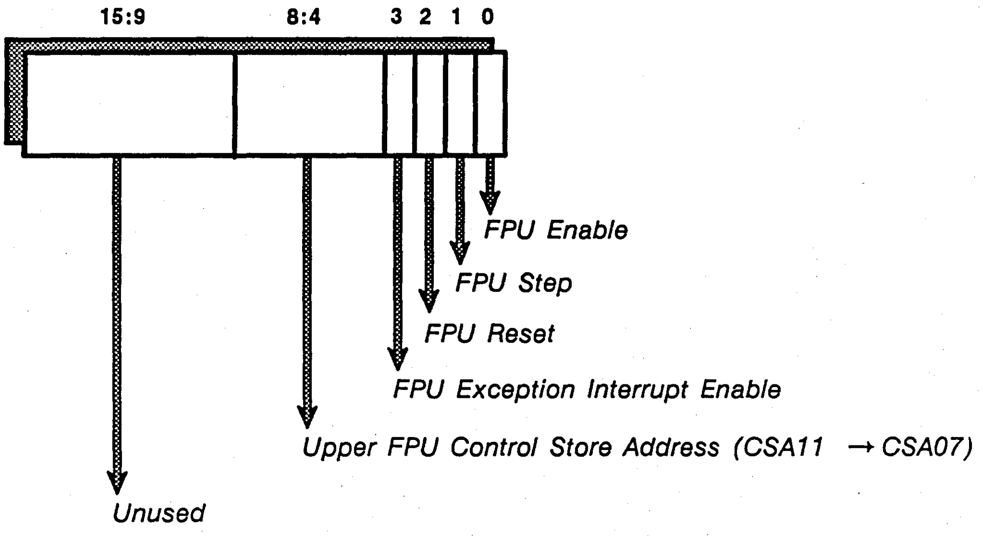
The PEB is a 32–bit microprogrammed computer, capable of executing 255 different instructions. User–visible elements of the PEB include the accumulator, temporary, and constant (or "x") registers; an integer 32–bit accumulator, and the PEB control page. Supervisor–visible elements of the PEB include user–visible elements, along with its writable control store (WCS), and a control register.

PEB Registers

The PEB control page contains those registers required for control of the PEB floating–point unit. These registers are intended for supervisor processes only.

The PEB Control Register

CPU writes to this register control PEB operations. See Figure A–1; it shows and describes the functions of the PEB control register.


15:9
8:4
3
2
1
0
FPU Enable
FPU Step
FPU Reset
FPU Exception Interrupt Enable
Upper FPU Control Store Address (CSA11 → CSA07)
Unused

FPU Enable < 0 >

0 = clock able to be single-stepped;
control store accessible.
1 = clock running;
control store inaccessible.

FPU Step < 1 >

If FPU Enable = 0 then toggling FPU Step (0 to 1 and back to 0) will advance the FPU through one microinstruction.

FPU Reset < 2 >

0 = reset FPU, start microprogram.
1 = allow FPU to run.

FPU Exception Interrupt Enable < 3 >

0 = do not allow interrupts.
1 = allow interrupts.

Upper FPU Control Store Address < 8:4 >

4 = Halt.
5 = Run.
6 = Step.

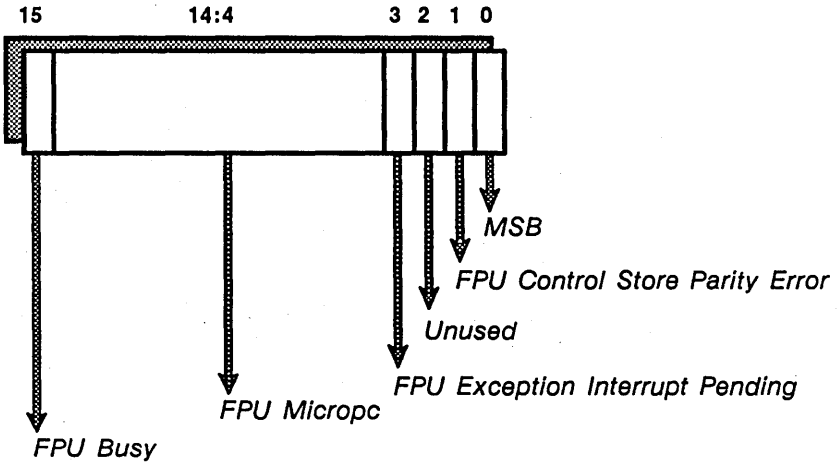
Unused < 15:9 >

These bits are presently unused.

Figure A-1. The PEB Control Register

The PEB Status Register

The CPU reads the PEB status register in order to monitor PEB operations. Refer to Figure A–2; it shows and describes the functions of the PEB status register.


15
14:4
3 2 1 0
MSB
FPU Control Store Parity Error
Unused
FPU Exception Interrupt Pending
FPU Micropc
FPU Busy

MSB < 0 >

(Bit 11) of micropc.

FPU Control Store Parity Error < 1 >

0 = FPU WCS ok.
1 = FPU error.
This bit is cleared by FPU Enable (bit 0) in the control register.

Unused < 2 >

This bit is unused.

FPU Exception Interrupt Pending < 3 >

0 = Not interrupting.
1 = Interrupt pending.
This bit is cleared when an interrupt occurs, or FPU reset in the control register.

FPU MicroPC < 14:4 >

These bits are the current micropc (bit 4 is lsb; bit 14 is msb).

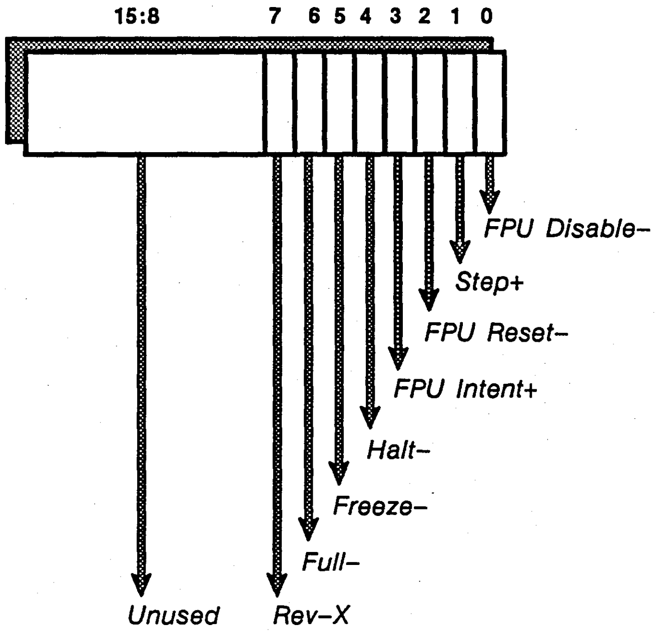
FPU Busy < 15 >

0 = FPU not busy
1 = FPU busy.

Figure A-2. The PEB Status Register

The PEB Diagnostics Register

See Figure A–3. It shows and describes the functions of the PEB diagnostics register.



FPU Disable– < 0 >

0 = halted; r/w WCS.
1= clock on; can't read WCS.

Step+ < 1 >

0 = clock running.
1 = clock stepping.

FPU Reset– < 2 >

0 = FPU is reset;, clear parity error.
1 = FPU can be running.

FPU Intent+ < 3 >

0 = do not allow interrupt to occur.
1 = allow interrupt to occur.

Halt– < 4 >

0 = FPU is halted.
1 = FPU is running.

Freeze– < 5 >

0 = FPU has hit a clock freeze and is halted.
1 = no freeze encountered.

Full– < 6 >

0 = microstack is full and micropc is unable to increment.
1 = microstack is ok.

Rev–X

0 = hardware Rev 0 1k WCS.
1 = hardware Rev 1 4k WCS.

Rev–X

These bits are unused.

Figure A–3. The PEB Diagnostics Register

PEB Floating-Point Formats

Floating-point formats for the PEB are as follows (where s = sign, e = exponent, and m = mantissa):

Single Precision	seee eeee emmm mmmm mmmm mmmm mmmm mmmm (an 8-bit exponent, a 23-bit mantissa + 1 hidden bit; eeeeeeee -> actual exponent + 127)
Double Precision	seee eeee eeee mmmm mmmm mmmm mmmm mmmm mmmm mmmm mmmm mmmm mmmm mmmm mmmm mmmm (an 11-bit exponent, 52-bit mantissa + 1 hidden bit; eeeeeeeeeee -> actual exponent + 2047)

Floating-Point Operations

There are 6 types of floating-point operations. These are:

1. Single precision dyadic operations
2. Double precision dyadic operations
3. Single precision monadic operations
4. Double precision monadic operations
5. Regular monadic operations
6. Special operations

A physical address within the FPU page represents each flotating-point operation. Mnemonics for floating-point operations are constructed from abbreviations. Abbreviations are as follows:

DA	Double Precision Add
DS	Double Precision Subtract
RD	Double Precision Reverse Subtract

DM	Double Precision Multiply
DMA	Double Precision Multiply and Accumulate
DD	Double Precision Divide
RD	Double Precision Reverse Divide
XCP	EXcePtion Status and Interrupt Enable
FA	Floating Accumulator
FAL	Floating Accumulator Low
FAH	Floating Accumulator High
FT	Floating Temporary
LFT	Load Floating Temporary
LIT	Load Integer Temporary
SA	Single Precision Add
SS	Single Precision Subtract
RS	Single Precision Reverse Subtract
SM	Single Precision Multiply
SMA	Single Precision Multiply and Accumulate
SD	Single Precision Divide
RDS	Single Precision Reverse Divide

Refer to Table A1, Table A–2, and Table A–3. These tables provide listings of floating-point operations, by type. The tables also provide those FPU page address assigned to every operation.

Table A-1. Single Precision Dyadic Floating-Point Operations

Single Precision Dyadic Operations

Mnemonic	Address and Meaning	
BUS_FA_SA	$7004	single precision add
LFT_FA_SA_FT	$7008	
FA_BUS_SA	$700C	
BUS_FA_SS	$7010	single precision subtract
LFT_FA_SS_FT	$7014	
FA_BUS_SS	$7018	
BUS_FA_RS	$701C	single precision reverse subtract
LFT_FA_SS_FA	$7020	
FA_BUS_RS	$7024	
BUS_FA_SM	$7028	single precision multiply
LFT_FA_SM_FT	$702C	
FA_BUS_SM	$7030	
BUS_FA_SD	$7034	single precision divide
LFT_FA_SD_FT	$7038	
FA_BUS_SD	$703C	
BUS_FA_RDS	$7040	single precision reverse divide
LFT_FA_RDS_FA	$7044	
FA_BUS_RDS	$7048	

Single Precision (Dyadic) Operations continue on the next page

Table A-1 (continued). Single Precision Dyadic Floating-Point Operations and their Addresses within the FPU Page

Single Precision Dyadic Operations		
Mnemonic	Address and Meaning	
LFT_SMIN	$7138	Min (FA, FT) -> FA
FA_BUS_SMIN	$713C	Condition Codes -> IAC high word
LFT_SMAX	$7150	Min (FA, FT) -> FA
FA_BUS_SMAX	$7154	Condition Codes -> IAC high word
LFA_SAX	$7168	FA + FX -> FA (single precision)
FA_BUS_SAX	$716A	
LFA_SMX	$717C	FA * FX -> FA (single precision)
FA_BUS_SMX	$7180	
BUS_FA_SP	$71A4	write sp "x" register for sp polynomial
LFT_FA_RDS_FA	$71A8	load FA for SP polynomial operation
FA_BUS_RDS	$71AC	(FA * FX) + FT -> FA
W_IAC_SP	$7214	load integer accumulator, float to sp
BUS_FA_SMA	$7234	sp multiply and accumulate
LFT_SMA	$7238	(FA * FT) + FX -> FA, FX
FA_BUS_SMA	$723C	
LFT_PWR	$7260	FA ** FT -> FA
IAC_SP_PWR	$726C	FA ** IAC -> FA

Table A-2. Double Precision Dyadic Floating-Point Operations

Double Precision Dyadic Operations

Mnemonic	Address and Meaning	
BUS_FAH_RDS	$704C	double precision add
BUS_FAL_DA	$7050	
BUS_FTH_DA	$7054	
LFTC_FA_DA_FT	$7058	
FAH_BUS_DA	$705C	
FAL_BUS_DA	$7060	
BUS_FAH_DS	$7064	double precision subtract
BUS_FAL_DS	$7068	
BUS_FTH_DS	$706C	
LFTC_FA_DS_FT	$7070	
FAH_BUS_DS	$7074	
FAL_BUS_DS	$7078	
BUS_FAH_RD	$707C	double precision reverse subtract
BUS_FAL_RD	$7080	
BUS_FTH_RD	$7084	
LFTC_FA_DS_FA	$7088	
FAH_BUS_RD	$708C	
FAL_BUS_RD	$7090	
BUS_FAH_DM	$7094	double precision multiply
BUS_FAL_DM	$7098	
BUS_FTH_DM	$709C	
LFTC_FA_DM_FT	$70A0	
FAH_BUS_DM	$70A4	
FAL_BUS_DM	$70A8	
BUS_FAH_DD	$70AC	double precision divide
BUS_FAL_DD	$70B0	
BUS_FTH_DD	$70B4	
LFTC_FA_DD_FT	$70B8	
FAH_BUS_DD	$70BC	
FAL_BUS_DD	$70C0	
BUS_FAH_RDD	$70C4	double precision reverse divide
BUS_FAL_RDD	$70C8	
BUS_FTH_RDD	$70CC	
LFTC_FT_DD_FA	$70D0	
FAH_BUS_RDD	$70D4	
FAL_BUS_RDD	$70D8	

Double Precision (Dyadic) Operations continue on the next page

Table A-2 (continued). Double Precision Dyadic Floating-Point Operations

Double Precision Dyadic Operations

Mnemonic	Address and Meaning	
BUS_FTH_DMIN	$7140	Min (DFA, DFT) -> DFA
LFTL_DMIN	$7144	Condition Codes -> IAC high word
FAH_BUS_DMIN	$7148	
FAL_BUS_DMIN	$714C	
BUS_FTH_DMAX	$7158	Max (DFA, DFT) -> DFA
LFTL_DMAX	$715C	Condition Codes -> IAC high word
FAH_BUS_DMAX	$7160	
FAL_BUS_DMAX	$7164	
BUS_FAH_DAX	$716C	FA + FX -> FA (double precision)
LFAL_DAX	$7070	
FAH_BUS_DAX	$7174	
FAL_BUS_DAX	$7178	
BUS_FAH_DMX	$7184	FA * FX ->FA (double precision)
LFAL_DMX	$7088	
FAH_BUS_DMX	$718C	
FAL_BUS_DMX	$7190	
BUS_FXH	$71B0	write dp "x" register for dp polynomial
BUS_FXL	$71B4	
BUS_FAH_DP	$71B8	dp polynomial
BUS_FAL_DP	$71BC	(DFA * DFX) + DFT -> DFA
BUS_FTH_DP	$71C0	
LFTL_DP	$71C4	
FAH_BUS_DP	$71C8	
FAL_BUS_DP	$71CC	
W_IAC_DP	$7218	load integer accumulator, float to dp
BUS_FAH_DMA	$721C	dp multiply and accumulate
BUS_FAL_DMA	$7220	(DFA * DFT) + DFX -> DFA, FFX
BUS_FTH_DMA	$7224	
LFTL_DMA	$7228	
FAH_BUS_DMA	$722C	
FAL_BUS_DMA	$7230	
BUS_FTH_PWR	$7264	FA * * FT -> FA
LFTL_PWR	$7268	
IAC_DP_PWR	$7270	FA * * IAC -> DP

Table A-3. Single Precision Monadic Floating-Point Operations

Single Precision Monadic Operations		
Mnemonic	Address and Meaning	
SP_NINT	$720C	nearest integer of sp -> sp
SP_TRUNC	$7240	truncate FA -> FA
SP_LOG	$7248	FA <- log(FA)
SP_EXP	$7250	FS <- exp(FA)
SP_SQRT	$7258	FA <- sqrt(FA)
SP_SIN	$7274	FA <- sin(FA)
SP_COS	$727C	FA <- cos(FA)
SP_TAN	$7284	FA <- tan(FA)
SP_ATAN	$728C	FA <- atan(FA)

Table A-4. Double Precision Monadic Floating-Point Operations

Double Precision Monadic Operations		
Mnemonic	Address and Meaning	
DP_NINT	$7210	nearest integer of dp -> dp
DP_TRUNC	$7244	truncate FA -> FA
DP_LOG	$724C	FA <- log(FA)
DP_EXP	$7254	FA <- exp(FA)
DP_SQRT	$725C	FA <- sqrt(FA)
DP_SIN	$7278	FA <- sin(FA)
DP_COS	$7280	FA <- cos(FA)
DP_TAN	$7288	FA <- tan(FA)

Table A-5. Monadic Floating-Point Operations

Monadic Operations		
Mnemonic	**Address and Meaning**	
F_NEG	$71E0	negate sp/dp accumulator
F_ABS	$71E4	absolute value of sp/dp accumulator
SP_DP	$71E8	convert sp in accumulator to dp
DP_SP	$714C	convert dp in accumulator to sp
L_SP	$71F0	float integer accumulator into sp
L_DP	$71F4	float integer accumulator into dp
SP_1	$71F8	fix sp to integer accumulator
DP_1	$71FC	fix dp to integer accumulator

Table A-6. Special Floating-Point Operations

Special Operations		
Mnemonic	Address and Meaning	
PEB_BASE	$7000	base address for PEB
FPU_RESET	$7000	reset FPU
R/W_XCP	$70F8	read/write exception register
HIGH_BUS	$7124	read upper 24 bits of DP mantissa
FX_TO_FA	$7194	FX -> FA
FA_TO_FX	$7198	FA -> FX
REV_BUS	$719C	read microcode revision level
FAH_BUS	$71D0	read dp "x" register
FXL_BUS	$71D4	
XCHNG	$71D8	FA <-> FX
FTH_BUS	$71DC	read dp register (high part) only
LIT_INTMUL	$7200	load integer "x" register, multiply
LIT_INTDIV	$7204	load integer "x" register, divide
LIT_INTDIV	$7208	load integer "x" register, reverse divide
FPU_STATUS_BUS	$73FC	fpu status register

The MC68881 Coprocessor

For information regarding the Motorola 68881 co-processor, refer to the *Motorola MC68881 Floating-Point Coprocessor User's Manual* (©1985, Motorola Inc).