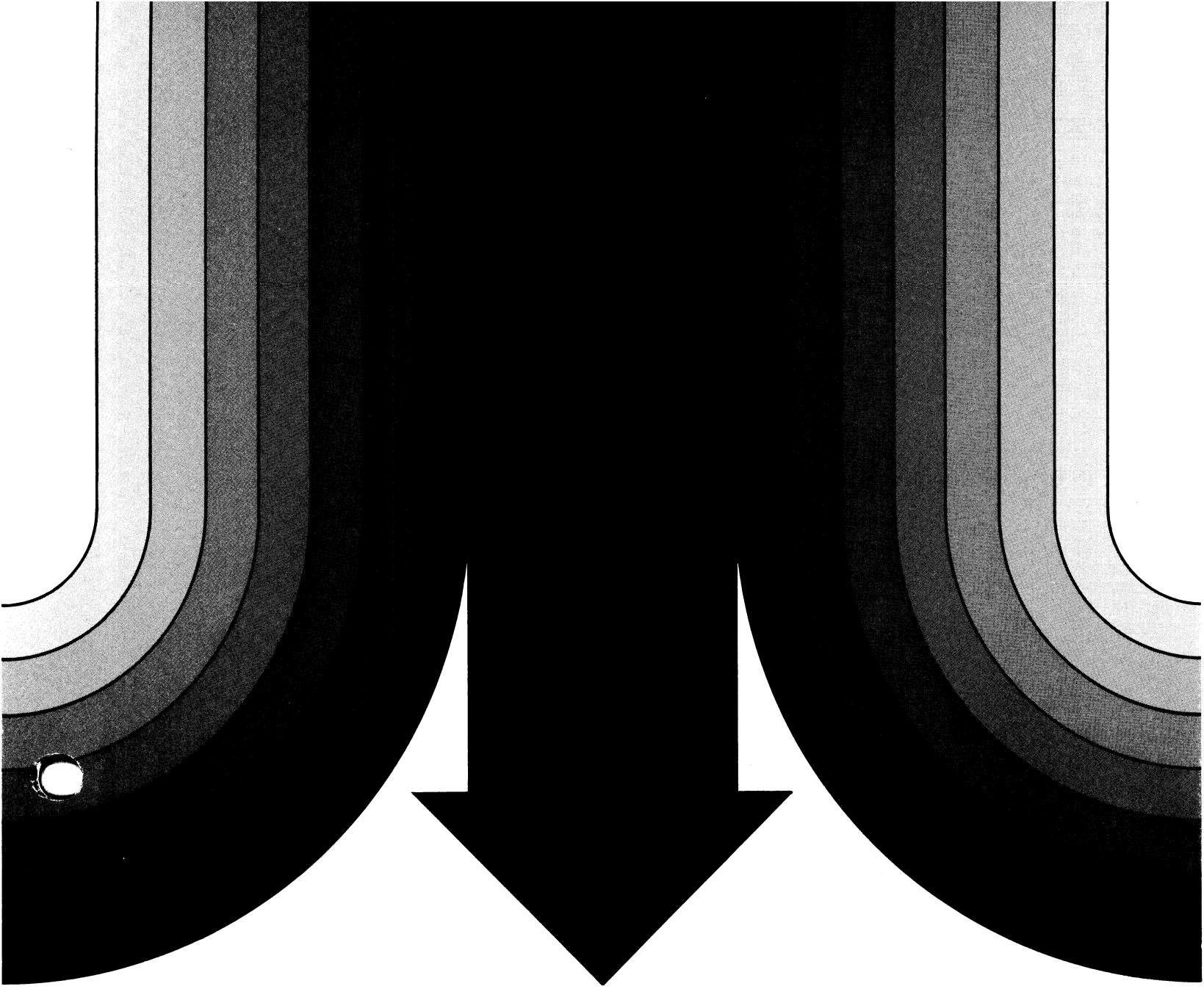


AOS/VS Internals Reference Manual

(AOS/VS Revision 5.00)

DataGeneral

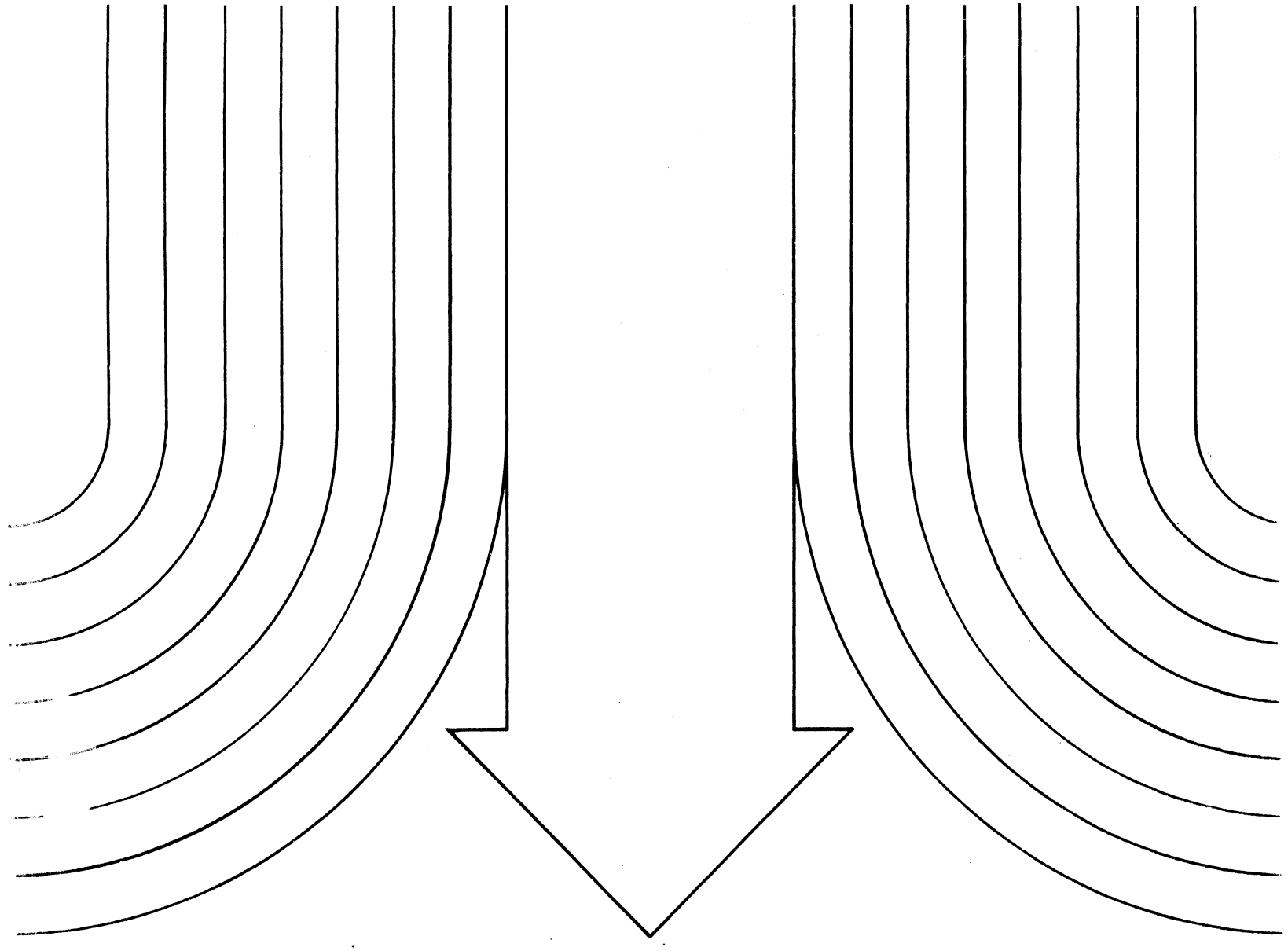
educational services

053-001001

NOTICE

IMPORTANT NOTICE

I UNDERSTAND THAT INFORMATION AND MATERIAL PRESENTED IN THE VS INTERNALS MANUAL MAY BE SPECIFIC TO A PARTICULAR REVISION OF THE PRODUCT. CONSEQUENTLY USER PROGRAMS OR SYSTEMS BASED ON THIS INFORMATION AND MATERIAL MAY BE REVISION-LOCKED AND MAY NOT FUNCTION PROPERLY WITH PRIOR OR FUTURE REVISIONS OF THE PRODUCT. THEREFORE DATA GENERAL MAKES NO REPRESENTATIONS AS TO THE UTILITY OF THIS INFORMATION AND MATERIAL BEYOND THE CURRENT REVISION LEVEL WHICH IS THE SUBJECT OF THIS MANUAL. ANY USE THEREOF TO YOU OR YOUR COMPANY IS AT YOUR OWN RISK. DATA GENERAL DISCLAIMS ANY LIABILITY ARISING FROM ANY SUCH SITUATIONS AND I AND MY COMPANY HOLD DATA GENERA HARMLESS THEREFROM.



AOS/VS Internals Reference Manual

(AOS/VS Revision 5.00)

DataGeneral

educational services

053-001001

Chapter 1 INTRODUCTION

Chapter 2 MV ARCHITECTURE

Chapter 3 KERNEL and DATA STRUCTURES

Chapter 4 AGENT

Chapter 5 THE USER ENVIRONMENT

Chapter 6 DISKWORLD

Chapter 7 EXEC

Chapter 8 PMGR

Chapter 9 CLI

Chapter 10 SYSTEM INITIALIZATION

CHAPTER 1 -- INTRODUCTION
(AOS/VS revision 5.00)

This chapter introduces operating systems and in particular, it describes the relation of AOS/VS to other Data General operating systems. It also introduces the building blocks of AOS/VS and their interface to the user.

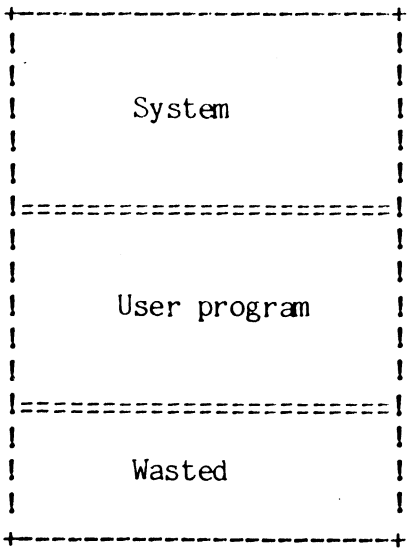
Introduction to Operating Systems

What is an operating system? It is a program which is a bit more complex than a typical application. However there are some very complicated application programs in the world. The operating system's purpose in life is to provide a mechanism which a person can use to manipulate data in various forms to achieve a specified result. It acts as a resource manager, an interface to the hardware, and the protector of each user's process space.

As a resource manager the operating system controls four major functions. They are: memory, processor, devices, and information.

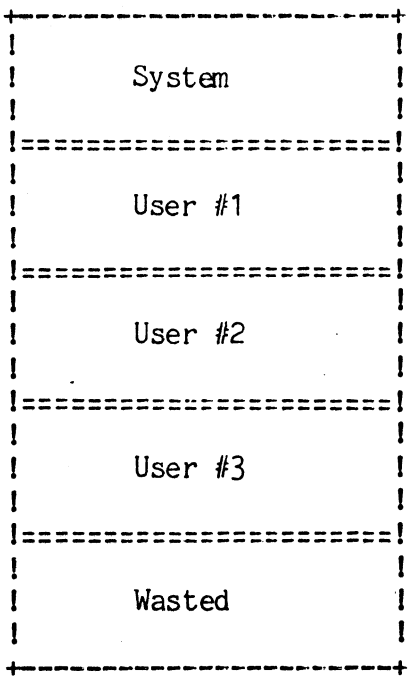
In memory management there are four common methods of control. They are: contiguous, partitioned, paged, and demand paged.

* Simple contiguous allocation



The above is typical of an MP/OS or DOS type system.

* Partitioned



Advantages:

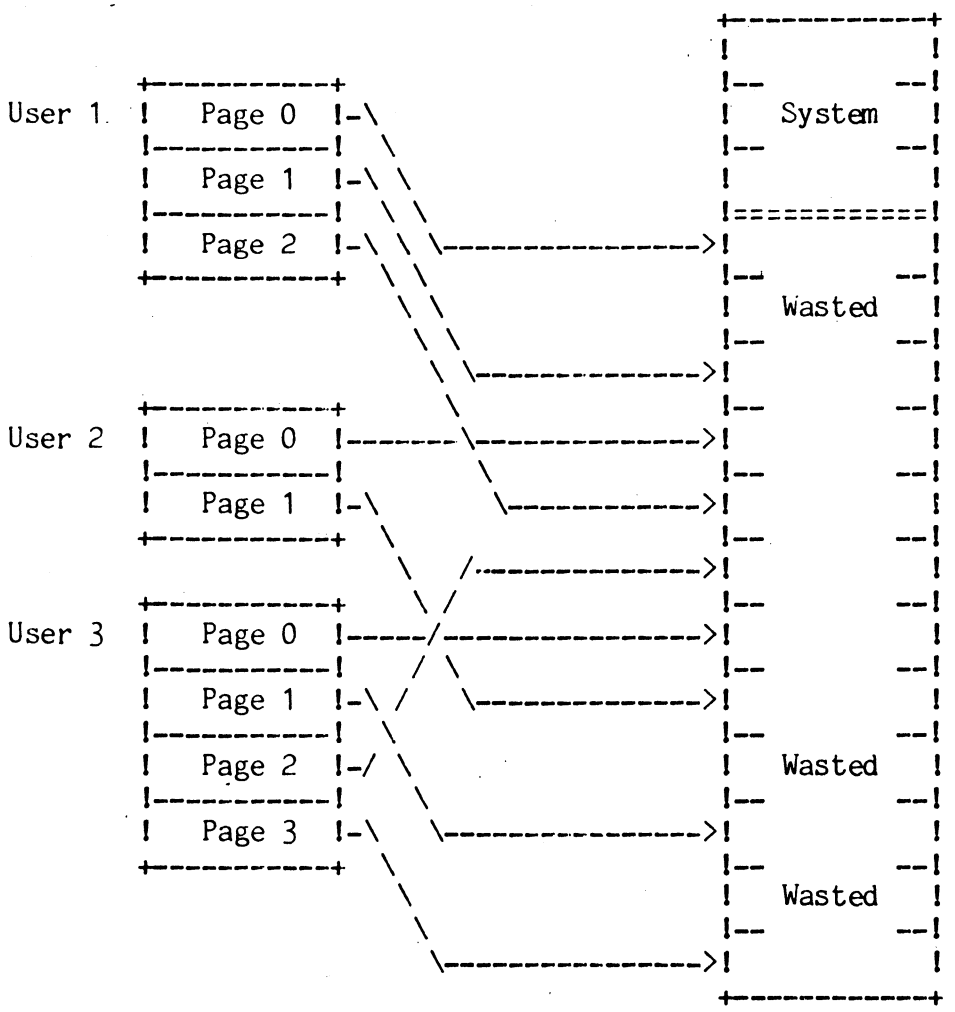
1. less wasted memory
2. less wasted CPU time (multiprogramming)

Disadvantages:

1. Special hardware
2. OS is more complex
3. Fragmentation

The above is typical of an RDOS type system except that RDOS is only a two ground (user) system.

* Paged allocation



This is typical of an AOS operating system.

Advantages:

1. Solves fragmentation problem

Disadvantages:

1. Additional hardware needed (page tables or map registers)
2. Non contiguous programs
3. Entire program must be in memory

* Demand paging

This is a variation on the paged system and is typical of an AOS/VS operating system. The variation is that programs running tend to have only the required pages in memory to be run. The rest of the program is either in the paging area or on the Shared Page LRU chain.

Advantages:

1. Allows partially loaded programs to be executed

Disadvantages:

1. Different coding philosophies (i.e. minimal indirection, modular code)
2. Extra overhead
3. Thrashing
4. Additional hardware (beyond paged allocated additions) to provide for referenced and modified flags, fault flag, and the restarting of instructions after a page fault.

Processor management techniques

There are typically three methods of processor management. They are: run to completion, run until blocked (i.e. pended waiting for I/O completion), and time-slice.

In a run to completion environment each program runs until it is done. This is typical of a batch environment. Each program runs in the sequence in which they are entered.

* Run to completion

    1. Simple to implement
    2. Adequate only for stand-alone or batch operating systems
    3. Need ability to terminate run-away programs
    4. CPU time is wasted waiting for completion of I/O.

In a run until blocked environment each program has control of the CPU until it needs to do some type of I/O. If a program is very CPU intensive it can control the CPU preventing anyone else from running.

* Run until blocked (pended waiting for I/O completion)

    1. Allows multiprogramming
    2. CPU bound programs can monopolize the CPU

In a time-slice environment the Real-Time clock assists in the control of program run time. Each program runs until it uses up its time-slice or it needs to do some type of I/O. There are several different methods for determining the scheduling frequency of these programs. They are: first come/first serve, round robin and priority. Each of these methods has its advantages. They will be disscussed in the scheduler section.

* Time-slice

    Run until either:

    1. Process blocks
    2. The time slice expires

    Possible algorithms:

    1. First come/first serve
    2. Round robin
    3. Priority

Device Management

In device management the operating system controls three type of devices. They are: input, output, and storage. For input and output these devices can be CRT's, hardcopy terminals, printers, plotters and card readers. The storage devices are magnetic tape (serial) and disk (direct access).

The above device access techniques are dedicated allocation, spooled, and shared access. CRTs, hardcopy terminals, and magnetic tape are usually dedicated allocation. Printers, plotters and card readers are usually spooled devices. Disks are shared devices.

There are problems associated with devices. Devices can be very expensive. Sharing helps to solve the expense problem. Device speeds vary but are very slow compared to the speed of the CPU. Error handling of devices can be very complex and costly in software as well as performance.

Information Management

An operating system also manages the flow and storage of information. The storage of information on a disk is the most important function of information management.

In the allocating of files on a disk there are two basic approaches which can be taken. The building of all files as contiguous is one approach. The other approach is to build the files using an indexed method (not ISAM).

Contiguous file allocation

Advantages

1. Simple
2. I/O is fast, efficient

Disadvantages

1. Disk fragmentation
2. Difficult to expand files
3. Must allocate disk space for 'holes'

Indexed File allocation

Advantages

1. Solves fragmentation problems
2. Easy to expand files
3. Need not allocate disk space for 'holes'

Disadvantages

1. Several accesses may be required to get data
2. More disk space required for a given file

AOS/VS Building blocks

The previous section discussed the principle functions of an operating system. The most important, is resource management. In AOS/VS, this is done by a number of different 'programs'. The following discussion will specifically tie parts of AOS/VS to the resources they handle. This will also serve as an introduction to the parts of AOS/VS.

The KERNEL

This is the heart of AOS/VS; it is the code that is created by the VSGEN program. The KERNEL is responsible for scheduling (process management), file / memory management, and interrupt processing.

The AGENT

The AGENT is responsible for labelled magtape, system call pre-processing (including the conversion of 16-bit packets to 32-bit packets),interfacing to GSMGR, deflecting calls to RMA, and generic file management.

The PMGR

All character oriented devices that are not on the data channel or BMC are controlled by the PMGR (peripheral manager). These include the consoles, card readers, and plotters. In each system, the PMGR exists three places: PID 1, ring 3 and ring 7; and in the IOP , IACs or COMMBATs of the MV machine.

The GSMGR

All synchronous oriented devices are controlled by the GSMGR. There is a separate BSCGEN needed to define the synchronous devices which the GSMGR controls.

AOS/VS Auxiliary Processes

EXEC

EXEC is responsible for the management of batch / print queues, labelled magtape mounts / dismounts, and log on / log off.

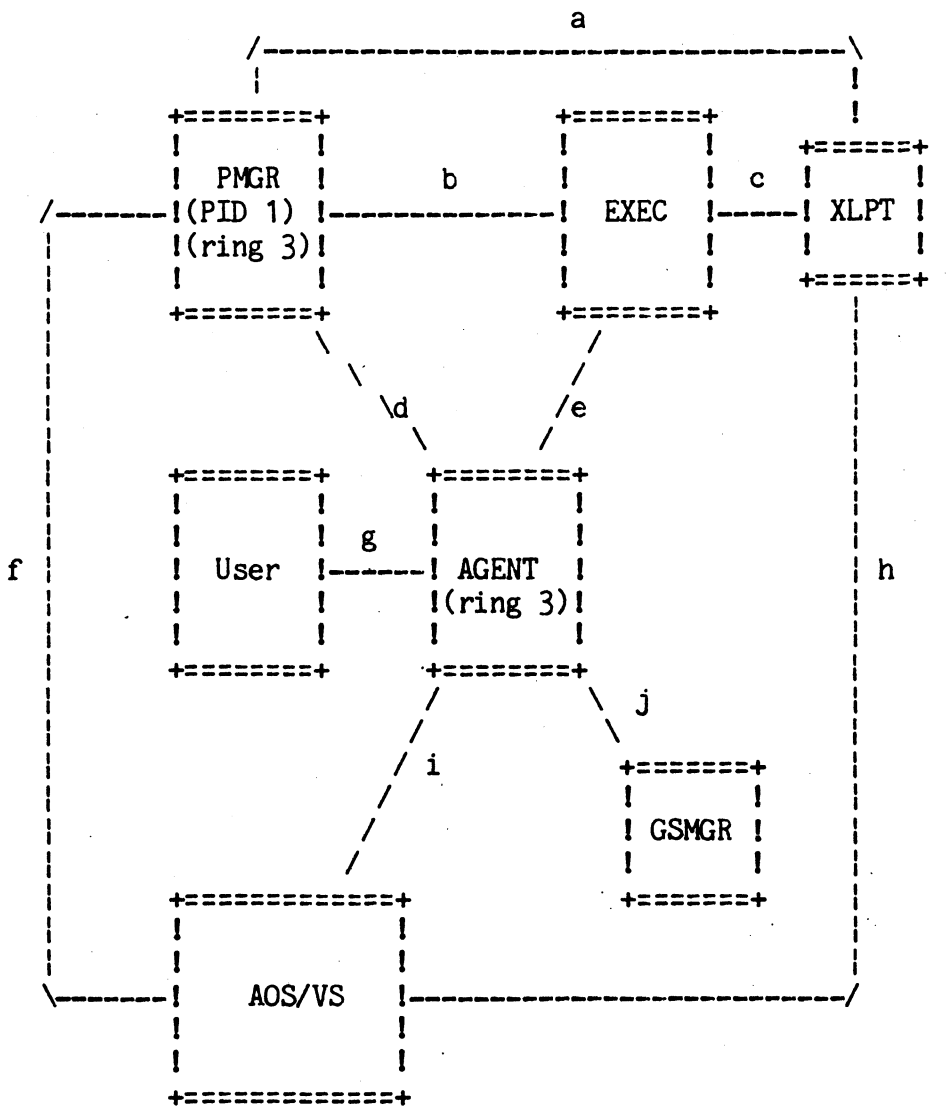
CLI

The CLI is an elaborate system call translator with a large number of bells and whistles (template expansion for example).

AOS/VS block diagram

On the following pages, there are two large block diagrams of AOS/VS. The first outlines the overall system picture and does not discuss

what goes on inside each module. The second is an attempt to represent the functions performed by the AOS/VS kernel. A complete description of the system would include some "special" users like PMGR and EXEC. All these are included in the box called "User events".



a. XLPT <--> PMGR — This interface is used in printing to non-data channel devices (consoles)

b. PMGR <--> EXEC — This interface is used for logon/logoff

c. EXEC <--> XLPT — This interface is used for queuing print requests and handling such things as restarts, flushes

..etc.

d. PMGR <--> AGENT — This interface handles I/O to terminals (?READ / ?WRITE are translated into IPC send/rec)

e. EXEC <--> AGENT — This interface handles the ?EXEC system call

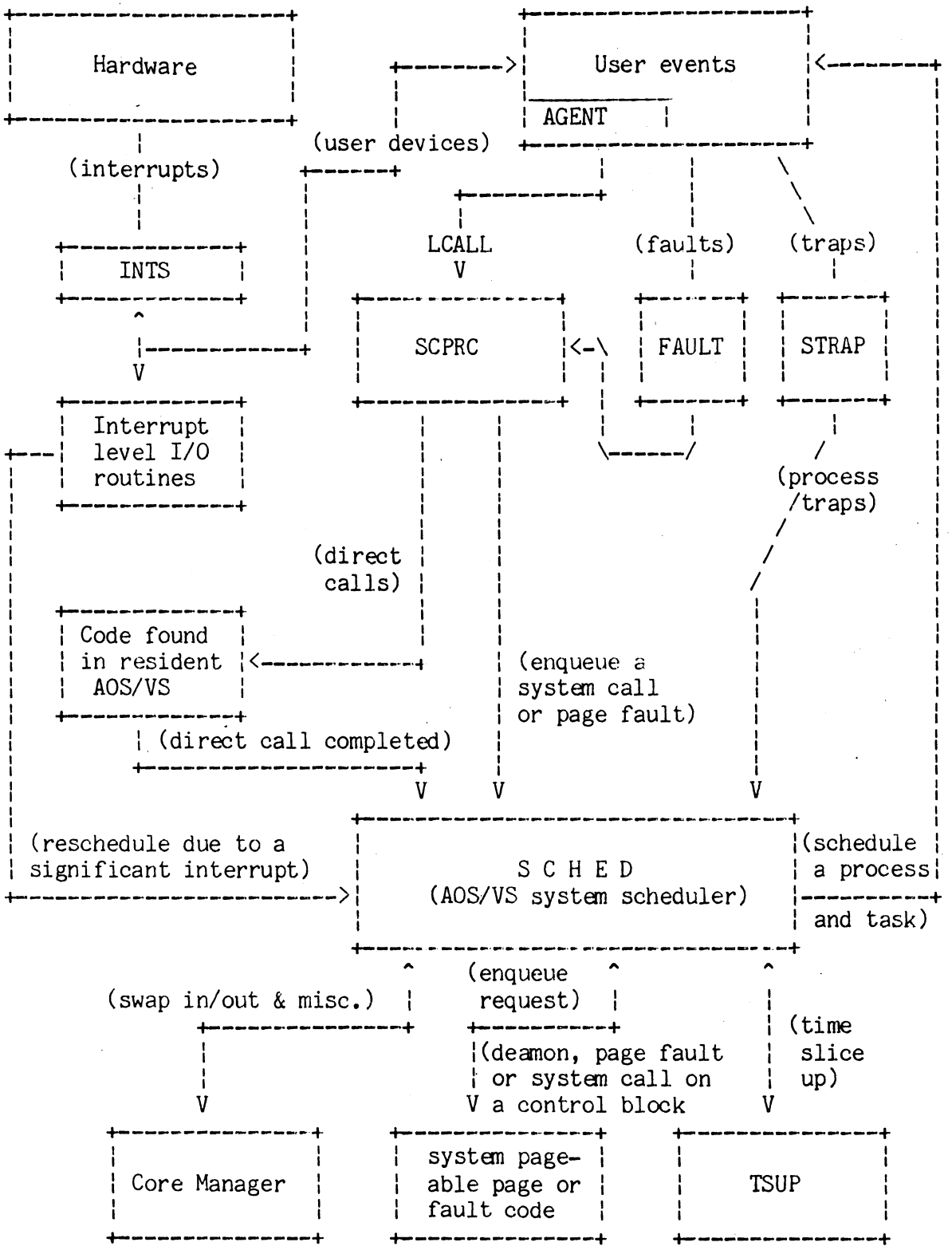
f. PMGR <--> AOS/VS — Character oriented I/O is handled by the PMGR at AOS/VS's request. In return, the PMGR can request that AOS/VS reschedule users.

g. USER <--> AGENT — System calls are processed through this link.

h. XLPT <--> AOS/VS — This interface is used in printing to data channel devices.

i. AOS/VS <--> AGENT — Many system calls are preprocessed by the AGENT, but eventually need the kernel.

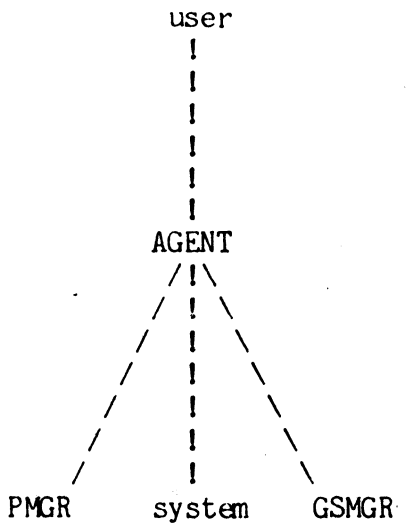
j. GSMGR <--> AGENT — Sync system calls are intercepted by the AGENT and are sent to the GSMGR by IPCs


Hardware
User events
AGENT
(user devices)
(interrupts)
INTS
LCALL
(faults)
(traps)
SCPRC
FAULT
STRAP
Interrupt level I/O routines
(process /traps)
(direct calls)
Code found in resident AOS/VS
(enqueue a system call or page fault)
(direct call completed)
(reschedule due to a significant interrupt)
S C H E D
(AOS/VS system scheduler)
(schedule a process and task)
(swap in/out & misc.)
(enqueue request)
(deamon, page fault or system call on a control block
(time slice up)
Core Manager
system page-able page or fault code
TSUP

User - system interface

Since the AGENT and the PMGR exist as seperate programs, in the same space, the apparent system-user interface is really several sub-interfaces:

user-AGENT, AGENT-system, PMGR-AGENT, GSMGR-AGENT, and PMGR-system.



The interfaces involving either AGENT or PMGR will be discussed in the AGENT and PMGR chapters. The user-system interface is summarized below and its various aspects will be discussed in detail in the appropriate chapters of the manual.

Control interfaces

There are four ways AOS/VS can take control away from a user :

- the process makes a system call
- the process takes a page fault
- the process traps
- an interrupt from a device comes in.

In the last three cases, control is yanked away from the user by the hardware without any software preparation. The user's AC's and PC are saved.

If a trap occurred, the process will be aborted by the system.

If an interrupt came in, control will be restored to the user after servicing the interrupt, unless the interrupt was significant enough to ready a higher priority process or control block.

Hardware Supported

The following is a list of the hardware supported by AOS/VS as of revision 5.00.

Mnemonic	Description
ATI	Asynchronous Terminal/Modem Interface
BBU	Battery Backup Unit
CRA	DGC 4016 Card Reader
CRA1	DGC 4016 Card Reader
DCU0	DGC 4254 Data Control Unit
DCU1	DGC 4254 Data Control Unit
DCU2	DGC 4254 Data Control Unit
DCU3	DGC 4254 Data Control Unit
DKB	DGC 6063, 6064 OR 6066 Fixed Head Disk
DKB1	DGC 6063, 6064 OR 6066 Fixed Head Disk
DKB2	DGC 6063, 6064 OR 6066 Fixed Head Disk
DKB3	DGC 6063, 6064 OR 6066 Fixed Head Disk
DKB4	DGC 6063, 6064 OR 6066 Fixed Head Disk
DKB5	DGC 6063, 6064 OR 6066 Fixed Head Disk
DKB6	DGC 6063, 6064 OR 6066 Fixed Head Disk
DKB7	DGC 6063, 6064 OR 6066 Fixed Head Disk
DPD	DGC 4234, 6045 OR 6030 Disk
DPD1	DGC 4234, 6045 OR 6030 Disk
DPF	DGC 60<60,61.67>, 61<22,60,61> OR 6214 Disk
DPF1	DGC 60<60,61.67>, 61<22,60,61> OR 6214 Disk
DPF2	DGC 60<60,61.67>, 61<22,60,61> OR 6214 Disk
DPF3	DGC 60<60,61.67>, 61<22,60,61> OR 6214 Disk
DPF4	DGC 60<60,61.67>, 61<22,60,61> OR 6214 Disk
DPF5	DGC 60<60,61.67>, 61<22,60,61> OR 6214 Disk
DPF6	DGC 60<60,61.67>, 61<22,60,61> OR 6214 Disk
DPF7	DGC 60<60,61.67>, 61<22,60,61> OR 6214 Disk
DPG	DGC 6070 Disk
DPG1	DGC 6070 Disk
DPI	DGC 60<97,98,99> 61<00,03> 62<25,27,34> Disk
DPI1	DGC 60<97,98,99> 61<00,03> 62<25,27,34> Disk
DPJ	DGC 62<36.37> Disk
DPJ1	DGC 62<36.37> Disk
DPJ2	DGC 62<36.37> Disk
DPJ3	DGC 62<36.37> Disk
DPJ4	DGC 62<36.37> Disk
DPJ5	DGC 62<36.37> Disk
DPJ6	DGC 62<36.37> Disk
DPJ7	DGC 62<36.37> Disk
DPM	DGC 45<13.14> Floppy Disk
DPM1	DGC 45<13.14> Floppy Disk
DRT	Dual Receiver/Transmitter
IAC	Intelligent Asynchronous Controller
IAC1	Intelligent Asynchronous Controller
IAC2	Intelligent Asynchronous Controller
IAC3	Intelligent Asynchronous Controller
IAC4	Intelligent Asynchronous Controller
IAC5	Intelligent Asynchronous Controller
IAC6	Intelligent Asynchronous Controller

IAC7	Intelligent Asynchronous Controller
IAC8	Intelligent Asynchronous Controller
IAC9	Intelligent Asynchronous Controller
IAC10	Intelligent Asynchronous Controller
IAC11	Intelligent Asynchronous Controller
IAC12	Intelligent Asynchronous Controller
IAC13	Intelligent Asynchronous Controller
IAC14	Intellignet Asynchronous Controller
IAC15	Intellignet Asynchronous Controller
LPB	DGC 42<15,16.18,19> OR 43<27,28,56> Line Printer
LPB1	DGC 42<15,16.18,19> OR 43<27,28,56> Line Printer
LPB2	DGC 42<15,16.18,19> OR 43<27,28,56> Line Printer
LPB3	DGC 42<15,16.18,19> OR 43<27,28,56> Line Printer
LPB4	DGC 42<15,16.18,19> OR 43<27,28,56> Line Printer
LPB5	DGC 42<15,16.18,19> OR 43<27,28,56> Line Printer
LPB6	DGC 42<15,16.18,19> OR 43<27,28,56> Line Printer
LPB7	DGC 42<15,16.18,19> OR 43<27,28,56> Line Printer
LPD	DGC 6088, 6089 OR 6192 Line Printer (LP2)
LPD1	DGC 6088, 6089 OR 6192 Line Printer (LP2)
LPE	DGC 4425 Laser Printer
LPE1	DGC 4425 Laser Printer
LPE2	DGC 4425 Laser Printer
LPE3	DGC 4425 Laser Printer
LPE4	DGC 4425 Laser Printer
LPE5	DGC 4425 Laser Printer
LPE6	DGC 4425 Laser Printer
LPE7	DGC 4425 Laser Printer
MCA	DGC 4206 Multiprocessor Communications Adaptor
MCA1	DGC 4206 Multiprocessor Communications Adaptor
MTB	DGC 6026 Magnetic Tape
MTB1	DGC 6026 Magnetic Tape
MTC	DGC 6123 OR 6231 Magnetic Tape
MTC1	DGC 6123 OR 6231 Magnetic Tape
MTC2	DGC 6123 OR 6231 Magnetic Tape
MTC3	DGC 6123 OR 6231 Magnetic Tape
MTD	DGC 4307 Magnetic Tape
MTD1	DGC 4307 Magnetic Tape
PLA	DGC 4017 Digital Plotter
PLA1	DGC 4017 Digital Plotter
TTY	Hardcopy Terminal
CRT	DGC D200 Compatible Console

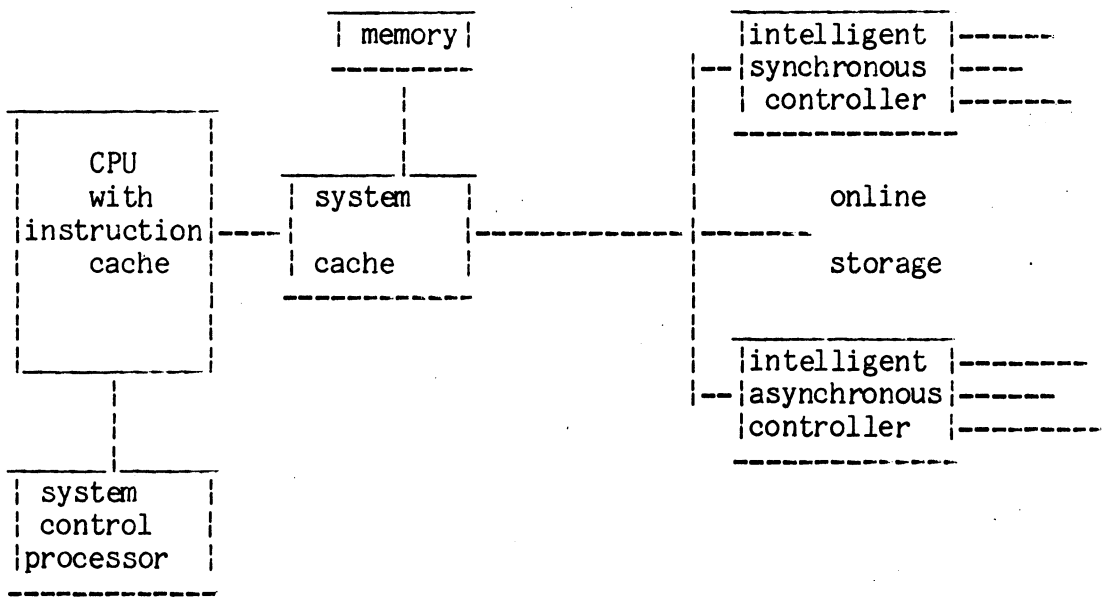
As can be seen from this chapter AOS/VS is a complex virtual operating system. The building blocks and auxiliary services provided by AOS/VS were introduced. In chapter 2 the MV hardware and microcode are discussed.

CHAPTER 2 -- MV ARCHITECTURE (AOS/VS REVISION 5.00)

This chapter introduces the MV architecture. its hardware. micro code, and software.

The MV series computer incorporates four main systems:

- The central processing unit (CPU) which consists of the instruction processor for decoding and executing instructions,
- The memory system, which consists of a system cache (except for MV-4000) that contains 1024 16-byte blocks and functions as a look-ahead / look-behind buffer; and some assortment of memory module.
- The input / output system which consists of one or two(mv10000) I/O channels suporting distrbuted processors for asynchronous and bi-synchronous communications.
- The system control processor (SCP), is a soft system console that performs diagnostics and operator controlled functions.



MV-10000 System Diagram

The 32-bit CPU provides facilities to manage data, access memory, and control program flow. The processor can perform fixed-point or floating-point computation, as well as stack, program, queue, device, system, and memory management. In addition, the processor contains the Eclipse® C/350 compatible instructions for 16-bit program development and upward compatibility.

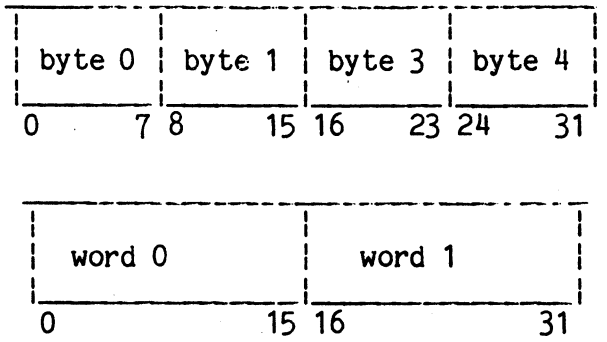
Fixed-Point Computation

Fixed-Point computation consists of fixed-point binary arithmetic with signed and unsigned 16-bit and 32-bit numbers. The processor also performs decimal arithmetic, logical operations, and manipulates 8-bit bytes.

The processor contains four 32-bit accumulators (AC0 - AC3) and a processor staus register (PSR).

Fixed-Point Accumulators

Fixed-point accumulators can be accessed by instructions that manipulate a bit, byte. word, or double word.



A word or double word operand must begin on a word boundary. A byte must begin on a byte boundary.

In addition to using an accumulator for fixed-point computation:

AC1 can contain a fault code placed there by the processor,

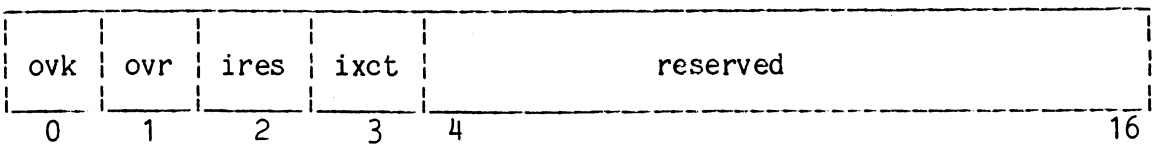
An instruction can be built in an accumulator and executed,

AC2 and AC3 can be used in relative addressing in place of the PC.

Processor Status Register

The Processor Status Register contains status flags such as an overflow fault service mask, a fixed-point overflow fault flag, and an interrupt resume flag. The overflow fault service mask enables or disables the processor from servicing the fault. The processor sets the overflow fault flag when the results of a fixed-point computation exceed the processor storage capacity. The interrupt resume flag reports an instruction status to the processor.

The processor status register bits can be accessed by instruction that set a bit or test and skip on condition of a bit.



OVK - overflow mask on = enable fixed-point overflow detection

OVR - overflow flag set on when a fixed-point overflow occurs
cleared by:

I/O interrupt request
Fault detection and servicing
Power up, I/O reset. or system reset
Processor executes instruction which accesses the register.

IRES - interrupt resume flag

This flag is set when the processor interrupts a resumable instruction that requires the processor to save its state on the user stack.

IXCT - is an interrupt-executed opcode flag

When the processor executes a BKPT instruction,it pushes a wide return block onto the current stack. AC0 in the return block contains the one-word instruction. When returning program control, the PBX instruction pops the wide return block and continues the normal program flow with the saved instruction in AC0.

Reserved - These bits are set to zero when stored in memory and ignored when loaded.

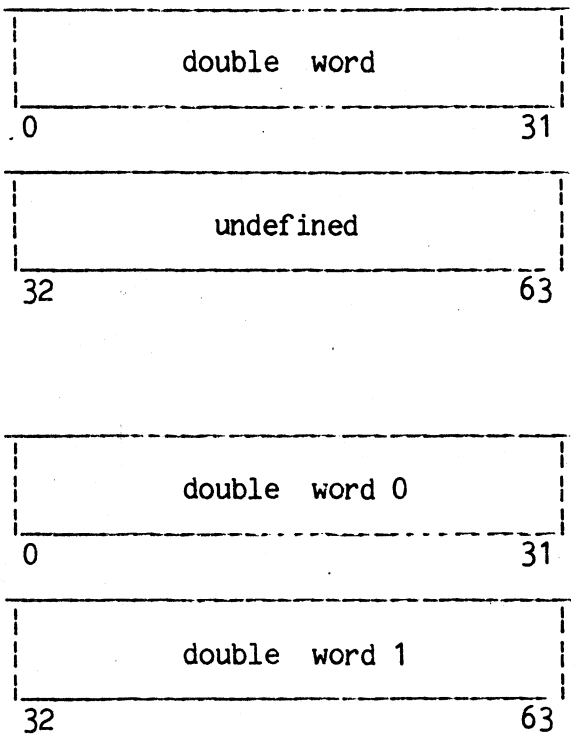
Floating-Point Computation

Floating-Point computation consists of floating-point binary arithmetic with signed, single precision (32 bits) and double precision (64 bits), numbers.

The processor contains four 64-bit floating-point accumulators (FPAC0 - FPAC3) and a floating-point status register (FPSR).

Floating-Point Accumulators

Floating-point accumulators can be accessed by instructions that manipulate single and double precision floating-point numbers.



A single precision number requires a double word (two consecutive words), while a double precision number requires two double words (four consecutive words).

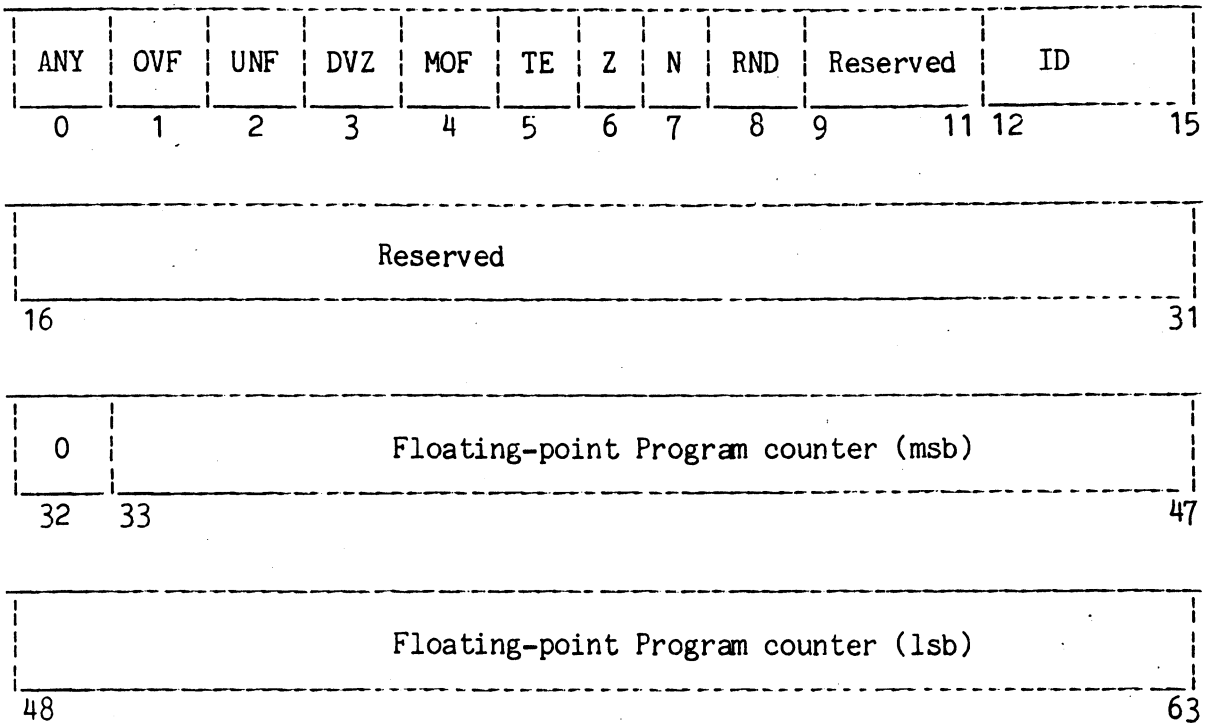
Floating-Point Status Register

The floating-point status register contains overflow and underflow fault flags, fault service mask, mantissa status flags, rounding flag, and processor status flags.

The processor sets an overflow or underflow fault flag when the result of a floating-point computation exceeds the processor storage capacity. The fault service mask enables or disables the processor from servicing a fault. The remaining flags provide processor status.

The contents of the register can be accessed by instructions that can

initialize it or test and skip on a condition.



Floating - Point status register format

ANY - error status flag set on when OVF, UNF, DVZ, or MOF is set

OVF - exponent overflow flag

UNF - exponent undeflow flag

DVZ - mantissa divide by zero

MOF - mantissa overflow flag

TE - trap enable mask

Z - true zero flag

N - negative flag

RND - round flag

Reserve - bits 9 - 11 are processor specific

ID - id which reflects floating-point revision

Reserve - bits 16 - 32 are processor specific

Floating-point Program counter - address of instruction causing error

Stack Management

The processor contains facilities for narrow and wide stack management. A stack is a series of consecutive locations in memory. Typically, a program uses a stack to pass arguments between subroutine calls and to save the program state when servicing a fault. After executing a subroutine or fault handler, the processor restores the program and continues program execution.

Narrow Stack Management

The narrow stack consists of a contiguous set of words for supporting ECLIPSE® C/350 program development and upward program compatibility. Narrow stack management includes three 16-bit narrow stack management parameters.

There are three parameters used to define and control the narrow stack.

Narrow Stack Limit - defines upper limit of the narrow stack

Narrow Stack Pointer - initially defines lower limit of the narrow stack. After access the narrow stack pointer defines the current location of the last word written onto or read from the narrow stack.

Narrow Frame Pointer - defines a reference point in the narrow stack

The C/350 (or narrow) return block normally consists of five words: the contents of the least significant 16 bits of the four accumulators, the least significant 15 bits of the program counter or the frame pointer, and the carry in bit 0 of the last word pushed.

Wide Stack Management

The wide stack consists of a contiguous set of double words for supporting the 32-bit processor programs. Wide stack management includes four 32-bit wide stack management parameters, for each memeory segment. (A memory segment is a logically addressable subset of memory, see memory management section)

Wide stack management for the current segment also includes four 32-bit wide stack management registers.

Wide Stack Base - defines the lower limit of the wide stack. When initialized it points to one double word below the actual address of first double word in stack.

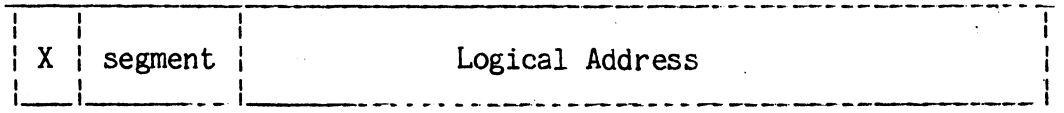
Wide Stack Limit - defines the upper limit of the wide stack.

Wide Stack Pointer - address of top location of the wide stack. It is either the location of the last word placed on the stack or the next available word on the stack.

Wide Frame Pointer - defines a reference point in the wide stack. The processor stores and resets the value of the wide frame pointer when entering or leaving subroutines. The wide frame pointer identifies theboundary between words placed on the wide stack before a subroutine call, and between words placed on the wide stack during a subroutine execution. Using the wide frame pointer as a reference, the processor can move back into the wide stack and retrieve arguments stored there by a preceding routine.

Stack overflow and underflow are stack faults. Stack overflow occurs when a program pushes data into the area beyond that allocated for the stack. Stack underflow occurs when a program pops data from the area beyond the allocated for the stack. Once detected, the processor always processes a stack fault.

Loading a 37777777777 into the wide stack limit register disables wide stack overflow fault detection. Loading a 20000000000 into the wide stack status register disables wide stack underflow fault detection.



Wide Stack Management Register Format

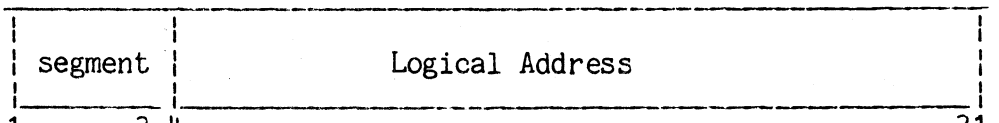
X - reserved

Segment - segment location of the stack

Logical address - logical address within the segment. Address wraparound can occur within the current segment.

Program Flow Management

Program flow management consists of controlling the program execution (such as calling a subroutine) and handling faults.



Program Counter Format

Segment - Bits 1-3 specify current segment

Logical address - logical address within the segment. Wraparound can occur within the segment.

The program counter specifies the logical address of the instruction to execute. Thus, it controls the sequence of executing the instructions. Address wraparound occurs within the current segment since only bits 4 through 31 take part in incrementing the program counter.

To address the next instruction(for normal program flow), the processor increments the program counter

By one on single word instructions

By two on two word instructions

By three on a three word instruction

By four on a four word instruction

Any of the following events can alter the normal program flow sequence.

executing an XCT instruction

executing a jump instruction

executing a skip instruction

executing a subroutine call

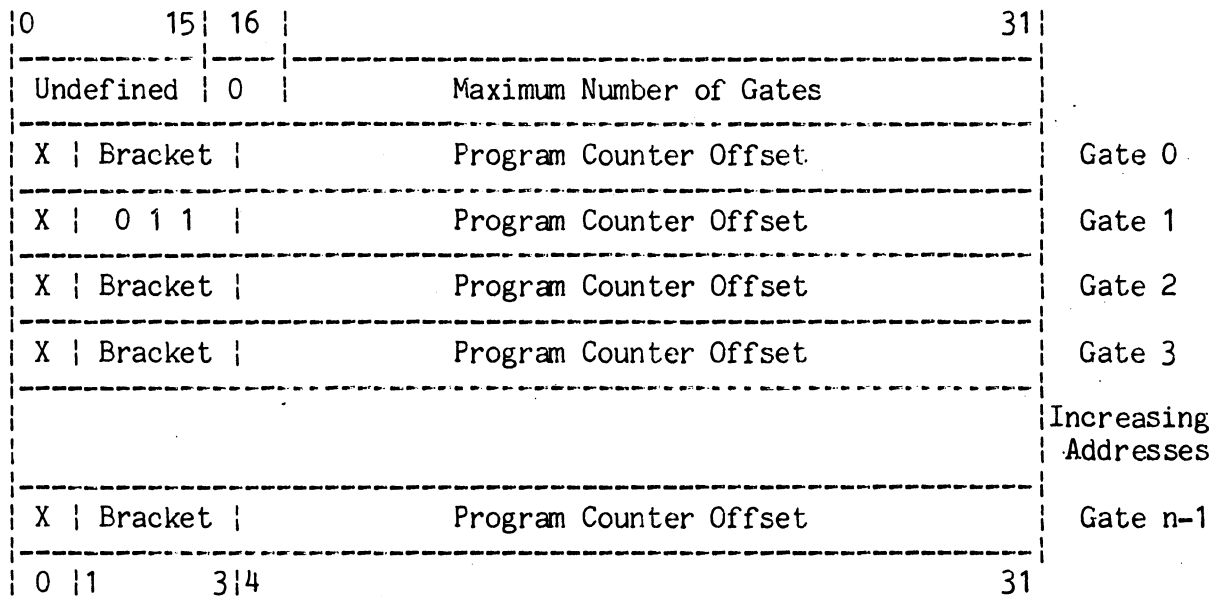
detecting a fault

detecting an I/O interrupt request

In a subroutine call the call is made using either an LCALL or XCALL instruction. The processor when using an LCALL or XCALL instruction performs four steps.

1. Verifies that the instruction can access destination segment
2. Validates the entry point through a gate array in the destination segment.
3. Redefines the wide stack and transfers call arguments to it
4. Transfers program control

A Gate Array is a series of locations that specify entry points(or Gates) to the segment. The processor accesses a gate array through an indirect pointer in page zero of the destination segment.



Gate Array Format

Undefined - processor does not care

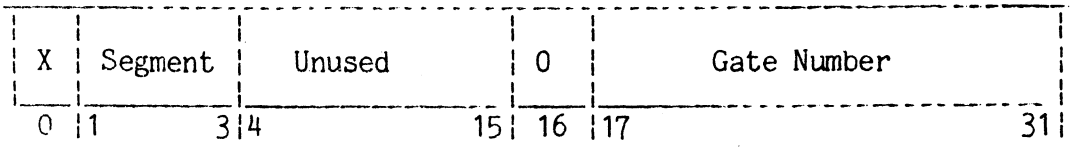
Maximum number of gates - total number of Gates

X - processor does not care

Bracket - gate bracket unsigned integer value 0 - 7 identifies the highest segment that can use the gate. If Gate 1 bracket contains 011, only segments 0 - 3 can access the segment.

Program Counter offset - address of first instruction of the subroutine in the destination segment.

The processor interprets the effective address of the XCALL or LCALL instruction as shown below.



XCALL or LCALL effective address

X - ignored by the processor

Segment - segment number of the destination segment

Unused - ignored by processor

Gate Number - gate in destination segment, used as index to an element(gate) in the vectored array.

When executing a subroutine in another segment, the processor uses the access privleges of the destination segment to determine the validity of the reference. A Trojan Horse pointer exists if one of the arguments passed from the source segment points to a location in the destination segment. (A privileged access fault would occur if a program refers to a locatin in a lower numbered segment.)

For example: a trojan horse pointer can exist when a program in segment 6 calls a subroutine in segment 2 and one of the arguments passed is a pointer to information in segment 2.

Fault Handling

While executing an instruction, the processor performs certain checks on the operation and the data. If the processor detects an error, a privleged or nonprivleged fault occurs before executing the next instruction. When the processor detects a fault, it pushes a return block onto the stack and jumps to the fault handler through the indirect pointer in reserved memory. The initial and indirect pointers to a fault handler (except a page fault handler) are 16 bits. Levels of indirection, if any, occur within the segment initially containing the pointer. A nonprivleged fault pointer is located in page zero of the current segment. A privileged fault pointer is located in page zero of segment 0.

If a privilege fault occurs while processing a nonprivilege fault, the processor aborts the nonprivileged fault and processes the privileged fault.

If an I/O interrupt occurs during the processing of a nonprivileged fault the processor pushes the fault return block, updates the program counter to the first instruction of the fault handler, then services the I/O interrupt. Upon returning from the I/O interrupt, the processor services the nonprivileged fault.

Fault	Type
Protection violation	Privileged
Nonresident page	Privileged
Stack operation	Nonprivileged
Fixed-point computation	Nonprivileged
Floating-point computation	Nonprivileged
Invalid decimal or ASCII data format	Nonprivileged

Faults

Device Management

Device management entails the transferring of data between memory and a device. The processor can transfer data (bytes, words, or blocks of words) with the programmed I/O (PIO), the Data Channel I/O (DCH), or the high speed burst multiplexor channel (BMC). Common to the three transfer facilities are the I/O instructions, mapped or unmapped memory addressing, and the interrupt system.

Programmed I/O

With programmed I/O bytes or words are transferred between an accumulator and a device. Programmed I/O is used to transfer data to low speed devices or to initialize a data channel or a burst multiplexor channel.

Data Channel I/O

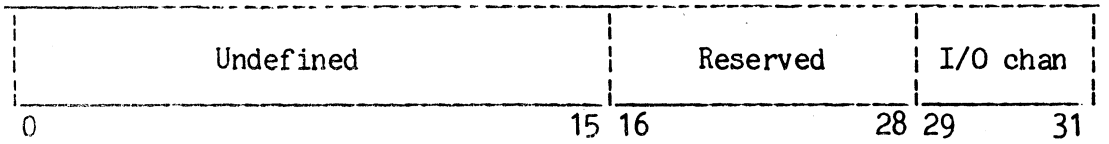
With the Data Channel I/O, a transfer of words is initiated between memory and a device. The data channel accesses memory directly (with or without a device map). Thus the data transfer bypasses the accumulators.

High Speed Burst Multiplexor Channel

With the burst multiplexor channel, a transfer of blocks of words between memory and a device is initiated. The burst multiplexor accesses memory directly (with or witout a device map). Thus, the data transfer bypasses the accumulators.

With the introduction of the MV/10000 there became a need to be able to specify which IOC (I/O channel) to use for data transfer. The PRTSEL instruction was created for this purpose. It changes the default I/O channel for data transfer.

PRTSEL - NIO 3,CPU with AC0 specifying the channel



Layout of AC0

System Management

System management provides facilities that determine processor dependent configurations, such as the processor identification and the size of main memory.

The processor supports memory management and system management faclities for an operating system. The memory management facilities transform a logical address into a physical address and monitor the contents of the physical memory. The system management facilities return or modify implementation dependent information about the system and the service faults.

The processor uses a virtual memory of 4 Gbytes. Virtual memory consists of eight segments or rings, which facilitate memory management. A segment is an addressable unit of memory that contains programs and data. A ring is a collection of protection mechanisms, which safeguards the contents of a segment.

The processor addresses a segment through a 0 - 7 numbering system. Each segment contains 512 Mbytes.

Segment 0
The processor executes privileged and non-privleged instructions as the kernel of the operating system.

Segments 1 - 7
The processor executes non-privleged instructions in segments 1 - 7.

Since the logical address space is larger than the physical address space, the processor uses a demand-paging scheme. THe processor maintains pages of logical memory on disk until it needs them in the physical memory.(A page equals 2 Kbytes.) when referring to an instruction or to data that currently resides on disk, the processor moves the page to physical memory. However. when the physical memory is full, the processor may first copy a page from memory to disk before moving the referenced page into memory. To facilitate the operation the processor maintains a table in memory that determines:

Where a page resides (memory or disk resident)

Bits 13 - 31 of a segment base register specify a physical address of a page table in memory. Each segment contains a page table, which occupies at leat 2 Kbytes and begins on an integral 2 Kbyte boundary. A page table contains entries that indicate where the pages reside in memory.

When to overwrite a page in memory with a page from disk.

The processor maintains a table of referenced and modified bits.

Segment access and address translation

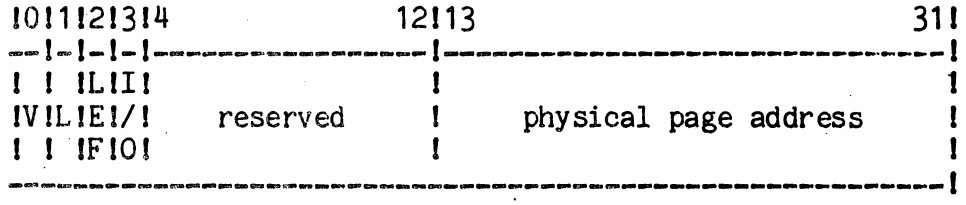
To access a memory word or words, the processor accesses a segment, translates a logical address(indirect or effective address) to a physical addresss, and accesses the physical page, which contains the word or words.

SBR (Segment Base Register)

For the processor to access a segment, it first checks the segment base register specified in the logical address. Bit 0 of the segment base register controls access to the segment by specifying if the processor can refer to the segment for the instruction execution. If the processor cannot refer to the segment. the processor aborts executing the instruction and services a segment validity protection fault.

The processor maintains eight segment base registers (SBR0 - SBR7) -- one for each of the eight segments. A segment base register contains information which

Validates a segment access
Validates an I/O access
Specifies a one- or two-level page table
Specifies for the segment the address of the first entry in the page table.



V	segment validity	0 - invalid	/ 1 - valid
L	length (in PT levels)	0 - 1 level	/ 1 - 2 level
LEF	LEF mode indicator	0 - I/O	/ 1 - LEF mode
I/O	I/O allowed	0 - no I/O	/ 1 - I/O allowed

Notes:

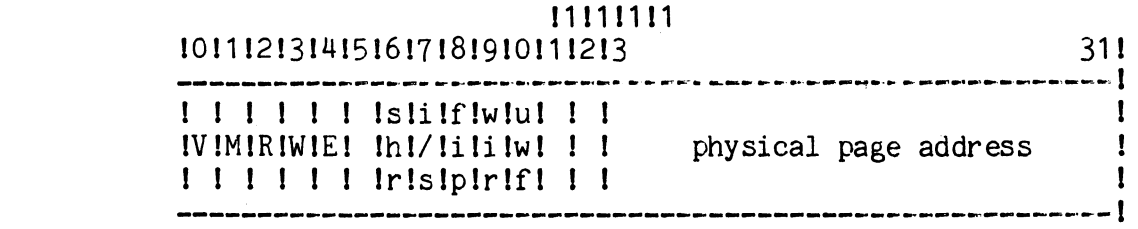
SBRs are loaded with one of the following instructions:

LSBRA - load all (0-7) of the SBRs
LSBRS - load some (1-7) of the SBRs

PTE (Page Table Entry) with software extensions

In each segment. the processor accesses a page table that specifies the status of the pages for the segment in memory. The page table contains one entry (PTE) for each page, which

Indicates if a page is a valid access and the type of access
Indicates if a page is currently in physical memory
contains information needed to translate a logical address to a physical address.



V	page validity	0 - invalid	/ 1 - valid
M	resident in memory	0 - no	/ 1 - yes
R	read access bit	0 - invalid	/ 1 - valid
W	write access bit	0 - invalid	/ 1 - valid
E	execute access bit	0 - invalid	/ 1 - valid

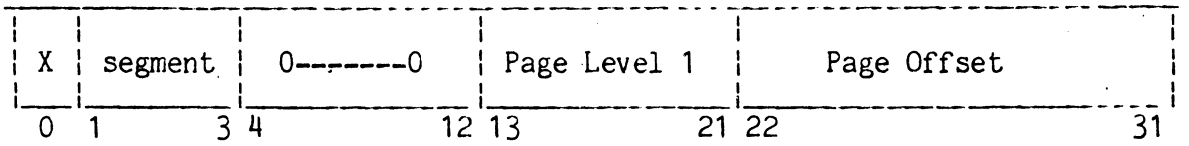
The following bits are software defined:

shr	page is shared	0 - not shared	/ 1 - shared
i/s	initially loaded if unshared	0 - not loaded	/ 1 - loaded
	data vs code if shared	0 - if code	/ 1 - if data
fip	fault in progress	0 - no	/ 1 - yes
wir	page is wired	0 - no	/ 1 - yes
uwf	unpend waiters of fault in prog	0 - no	/ 1 - yes

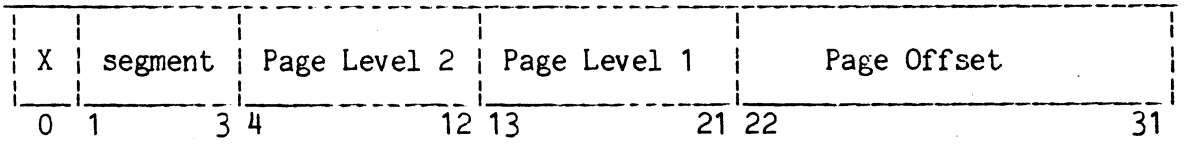
Address Translation

Following a valid segment reference, the processor checks the range of the logical address space within the segment, and compares it to the address range of the logical address. Bit 1 of the segment base register defines a one- or two-level page table, which specifies the addressing range.

The processor compares bit 1 of the segment base register with bits 4 - 12 of the logical address. When bit 1 equals a zero, the logical address bits 4 - 12 must be all zeros. The processor aborts executing the instruction and services the protection fault (page table depth fault) when any of the logical address bits 4 - 12 contain a one.



One-level Page Table Logical Word Address



X - ignored by processor when using direct addressing. Tested by processor when using indirect addressing, and continues testing the bit in subsequent indirect address until bit is zero

Segment - specifies one of eight segment base registers

Page Level 2 - specifies an entry in the first of two page tables for a two-level page table translation. The page table entry contains the address of the second page table. For a one-level page table translation, the page level 2 field must be all zeros. If not zeros then page table validity protection fault occurs.

Page Level 1 - specifies an entry in a page table. For a one- or two-level page table translation, the page table entry contains the address of the final page to be accessed for data or an instruction.

Page Offset - The page offset specifies the final entry in the final page. The page offset completes the address translation.

Page Access

When an instruction refers to a page, the processor determines the validity of the access by checking the access request with the appropriate validation and access validation bits in the page table entry.

When an instruction refers to a valid page that is not currently in physical memory, a page fault occurs. The fault handler saves the current state of the processor in reserved memory (context block), moves a memory page to disk (if required), and then transfers the referenced page from disk to memory.

Access Validation

When a referenced page is valid, the processor determines whether the page is restricted to a particular access. Bits 2 - 4 of the referenced page table entry contain the access bits that specify any restriction.

When the reference to memory is for reading, the processor checks bit 2. A one in bit 2 indicates a valid read, while a zero indicates an invalid read. When the reference is invalid, a protection fault occurs and AC1 contains the error code 0.

When the reference to memory is for writing, the processor checks bit 3. A one in bit 3 indicates a valid write, while a zero indicates an invalid write. When the reference is invalid, a protection fault occurs and AC1 contains the error code 1.

When the reference to memory is for executing, the processor checks bit 4. A one in bit 4 indicates a valid execute. while a zero indicates an invalid execute. When the reference is invalid, a protection fault occurs and AC1 contains the error code 2.

*** Note - In general, READ access must always be available to any page with execute access....

Demand Paging

Since the logical address space is larger than the physical memory space, all pages cannot reside in physical memory at the same time. A paging facility(under control of the page fault handler) moves referenced pages in and out of memory whenever necessary-- demand paging.

When an instruction refers to a valid page not currently in physical memory or refers to a location that requires two-level page table when only a one-level page table is allocated, then a page fault occurs. A status field in the context block indicates the cause of the page fault. Refer to the specific functional characteristics manual for more information on the context block.

Page faults

A page fault can occur for the following reasons:

Page table depth (an attempt was made to translate a two level page table entry when only a one level table was specified)

Page fault when referencing a page table

Page fault when referencing an object page

When a page fault occurs, the MV will copy the current context block into the locations pointed to by offsets 32 and 33 of segment 0. The processor then crosses to ring 0 and jumps indirect throught locations 30 and 31 of segment 0 which contains the fault handler.

Referenced and modified flags

A referenced and a modified flag are associated with a physical page in memory. When the processor reads a word from memory, it sets the referenced flag associated with the physical page to one. When the processor writes a word to memory, the processor sets the referenced and modified flags associated with the physical page to one. A read or write operation occurs when the processor accesses memory witout a protection fault occurring on a memory resident page.

**** Note: An I/O memory reference does not affect the state of the flags

The referenced flag helps to determine which page in physical memory the page fault handler should replace with a new page from disk. The referenced flag allows an operating system and the page fault handler to determine the frequency of references to individual pages.

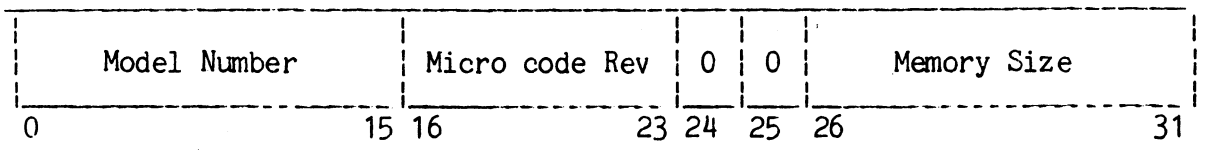
The modified flag indicates if the processor wrote a memory page. When a modified flag equals one, the processor modified the contents of the page. The page fault handler must first copy the page to disk before moving a new page from disk to memory. If a modified flag is zero, the processor did not modify the contents of the page, and the page fault handler can immediately move a new page from disk to memory.

Central Processor Identification

The processor stores information about the processor parameters (such as the memory size and micro code revision level) in one or more fixed-point accumulators. Refer to the specific functional characteristics manual for further information on the accumulators.

The following three load cpu identification instructions return the infomation as shown.

LCPID or ECLID



bits 0 - 15 the binary value of model number (10001001001100) for MV10000

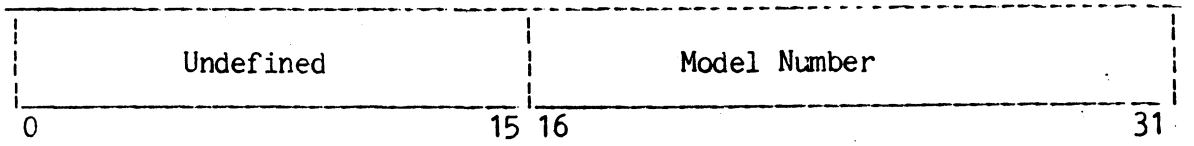
bits 16 - 23 current micro code revision

bits 24,25 set to 0

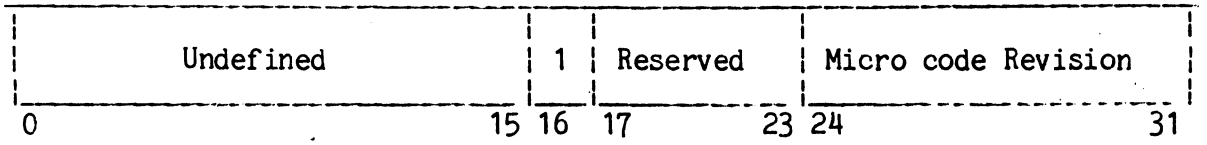
bits 26 - 31 amount of physical memory available
a 0 is 256 Kbytes
a 1 is 512 Kbytes to a maximum indicating 16 Mbytes.

NCLID

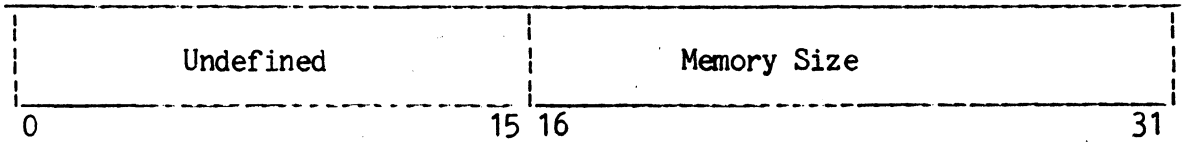
Returned in AC0



Returned in AC1



Returned in AC2



AC0 - model number binary representation (10001001001100)

AC1 - Micro code revision

Bits	Meaning
16	Always set to 1
17 - 23	Reserved for future use
24 - 31	current micro code revision

AC2 - Memory size
A 0 indicates 32 Kbytes
A 1 indicates 64 Kbytes

Protection Violation

The processor performs certain checks on the operation and on the data while executing an instruction. If the processor detects an error, a privleged or non-privleged fault occurs. Since an operation could produce multiple protection violations, the processor imposes priorities on the faults. The processor services the highest priority fault and ignores lower priority faults, when two or more occur. For instance, the processor services a level 2 priority and ignores a level 4 priority, when both occur simultaneously.

When the processor detects a fault, it performs a segment crossing to segment 0 (if the fault occurs in segment 1 to 7) and jumps to the protection violation fault handler through the indirect pointer in reserved memory. The initial and indirect pointers to the protection violation fault handler are 16 bits. Levels of indirection, if any, occur within segment 0.

If a protection violation fault occurs while handling a nonpriveleged fault, the processor aborts the nonpriveleged fault and processes the protection violation fault. The return block pushed onto the stack for the protection violation fault is undefined, as are the contents of AC0 and AC1.

Level of Priority	Fault Description
0	Privleged or I/O instruction violation
1	Indirect addressing violation
2	Inward reference violation
3	Segment validity violation
4	Page table validity violation
5	Read, write. or execute access violation
6	Segment crossing violation

Priority of protection violation faults

C/350 Programming

The 32-bit processor executes 16-bit procesor instructions to provide upward program compatibility and to develop 16-bit programs (for instance, for the ECLIPSE C/350 procesor). Programs that include C/350 memory-referenced and C/350 stack-referenced instructions must meet certain requirements or restrictions. The specific functional characteristics manual presents any machine restrictions.

C/350 registers

The C/350 fixed-point accumulator bits 0-15 correspond to the wide fixed-point accumulator bits 16-31. When a C/350 instruction loads data into an accumulator, it alters bits 16-31, and ignores bits 0-15. When a C/350 instruction reads data from an accumulator (bits 16 - 31), it does not alter the contents.

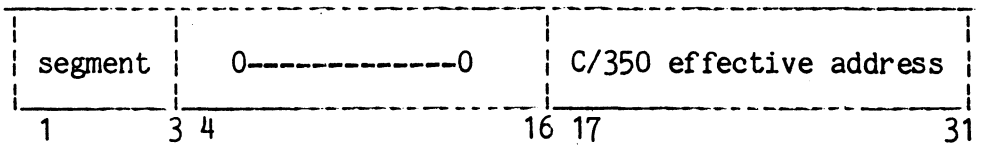
The C/350 fixed-point accumulator bits 1-15 correspond to the wide accumulator bits 17-31 for relative addressing.

The C/350 instructions do not affect processor status register.

The C/350 floating-point accumulators are identical to the 32-bit processor floating-point accumulators.

The C/350 program counter bits 1-15 correspond to the wide program counter bits 17-31. A C/350 program flow instruction modifies bits 17-31, while the most significant bits are the current segment and zeroes.

The C/350 reserved memory in the MV processor does not implement the auto-increment and auto-decrement locations 20 thru 37. The processor reserves these locations for storage of certain system parameters.



C/350 program counter format

Segment - current segment

C/350 Effective address - remains within the first 64Kbytes of the segment

C/350 Stack

The C/350 stack (or narrow stack) supports C/350 program development and upward program compatibility. Unlike the wide stack the narrow stack only uses three parameters (in reserved memory) to define and to control the narrow stack.

1. narrow stack limit - defines upper limit of stack

2. narrow stack pointer - current location of last word written onto or read from the narrow stack

3. narrow frame pointer - defines a reference point in the narrow stack

C/350 Faults and Interrupts

The 32 bit processor services (with the same pointers and fault handlers) the 16- and 32-bit floating-point and decimal/ASCII faults. It also processes I/O interrupts the same way.

System Control Processor (SCP)

The system control processor (SCP) is a system within the MV computer and has its own microcomputer. That is, the SCP has its own CPU and its own operating system. The SCP is a soft system console. It performs diagnostic functions and loads micro code into the microsequencer.

As a soft console, the SCP performs system control functions under operator control. It permits the operator to load or examine and modify main memory and to single-step through a program instruction by instruction.

As a diagnostic tool, the SCP runs programs designed to help isolate hardware problems. It also maintains an error log. When an error occurs, the SCP records the type of error, its location, and the time it occurred.

The SCP provides all the system timing for the MV computer system. It also connects to other components via several buses to allow examination and modification or internal registers.

The operator terminal of the SCP gives the operator control over the MV processor by transmitting commands to the system and provideing direct responses and reports.

The SCP also contains the real-time clock, the programmable interval timer, and the primary asynchronous line, all of which appear to the main processor to be I/O devices.

Data Channel/Burst Multiplexor Channel

The data channel (DCH) provides I/O communications for medium-speed devices and synchronous communications. The burst multiplexor channel (BMC) is a high speed communications pathway that transfers data directly between main memory and high-speed peripherals. The I/O-to-memory transfers for both DCH and BMC always bypass the address translator.

DCH/BMC Maps

A map controls a DCH or BMC. This map is a series of contiguos map slots, each of which contains a pair of map registers - and even-numbered register and its corresponding odd-numbered register.

The MV computer supports 16 DCH maps, each of which contains 32 map slots. The DCH sends to the processor a logical address with each data transfer. The processor translates the logical address into a physical address using the appropriate map slot for that address.

The device controller performing the data transfer controls the BMC. No program control or CPU interaction is required, except when setting up the BMC's map table. The BMC has two address modes and contains its own map.

BMC address modes

The BMC operates in either the unmapped mode - that is, the physical mode - or the mapped mode - that is, the logical mode.

In the unmapped mode, the BMC receives 20-bit addrsses from the device controllers and passes them directly to memory. As the BMC transfers each data word to or from memory, it increments the destination address, causing successive words to move to or from consecutive locations in memory.

If the controller specifies the mapped mode for data transfer, the high-order 10 bits of the logical address from a logical page number, which the BMC map translates into a 10-bit physical page number. This page number, combined with the 10 low-order bits from the logical address, forms a 20-bit physical address, which the BMC uses to access memory.

BMC Map

The BMC uses its own map to translate logical page numbers into physical ones.The map table contains 1024 map registers, the odd-numbered registers each containing a 10-bit physical page number. The BMC uses the logical page number as an index into the map table, and the contents of the selected map register becomes the high-order 10 bits of the physical address.

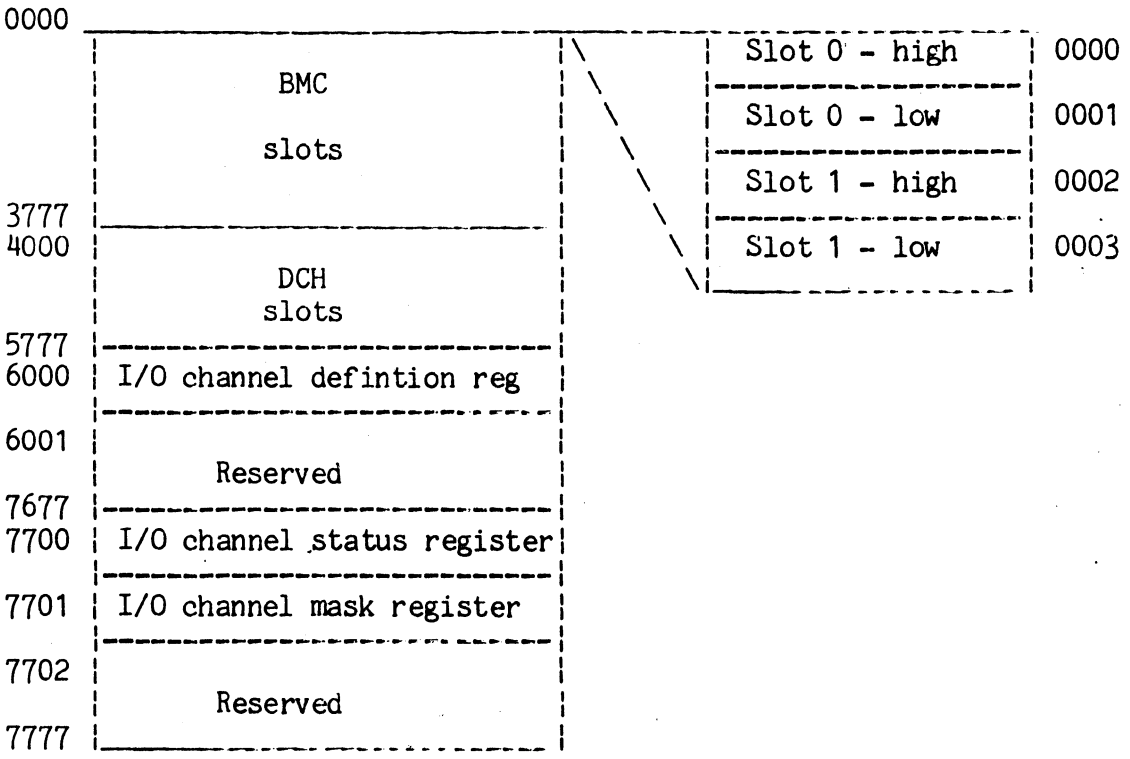
Note that when the BMC performs a mapped transfer, it increments the destination address after it moves each data word. If the increment causes an overflow out of the 10 low-order bits, this selects a new map register for subsequent address translation. Depending on the contents of the map table, this could mean that the BMC cannot transfer successive words to or from consecutive pages in memory.

DCH/BMC Registers

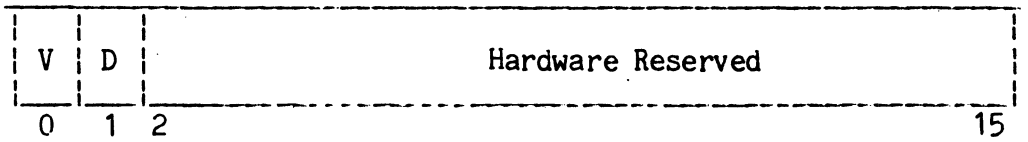
The MV computer system contains 512 DCH registers and 1024 BMC registers. The map registers are numbered form 0 through 7777.

Registers	Description
0000 - 3776	Even-numbered registers most significant half of BMC map positions 0 - 1777
0001 - 3777	Odd-numbered registers least significant half of BMC map positions 0 - 1777
4000 - 5776	Even-numbered registers most significant half of DCH map positions 0 - 777
4001 - 5777	Odd-numbered registers least significant half of DCH map positions 0 - 777
6000	I/O channel definition register
6001 - 7677	reserved
7700	I/O channel status register
7701	I/O channel mask register
7200 - 7777	reserved

Device map registers



DCH/BMC registers

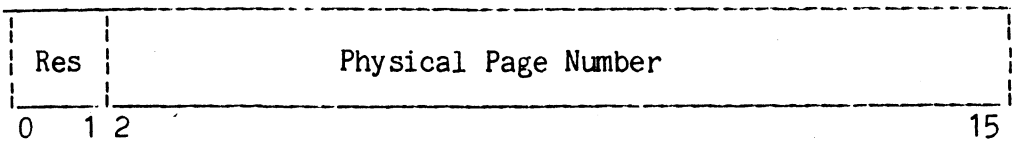


Even-Numbered Register Format

V - validity bit ; if 1 processor denies access

D - data bit
if 0, the channel transfers data
if 1, the channel transfers zeros

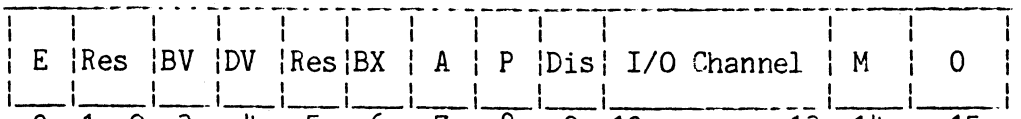
Reserved write to with zeros; reading these bits returns undefined state



Odd-Numbered Register Format

Res - Hardware Reserved

Physical Page Number - associated with logical page reference



I/O Channel Definition Register Format

E - Error flag

Res - reserved

BV - BMC validity error flag if 1 BMC protect error has occurred

DV - DCH validity error flag if 1 DCH protect error has occurred

Res - reserved

BX - BMC transfer flag BMC transfer in progress

A - BMC address parity error has occurred

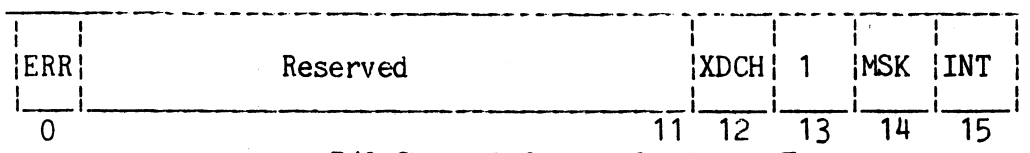
P - BMC data parity error hs occurred

DIS - disable block transfer

I/O channel - I/O channel number

M - DCH mode if 1 DCH mapping is enabled

0 - always set to 0



I/O Channel Status Register Format

ERR - I/O channel detected error by IOC or memory parity error

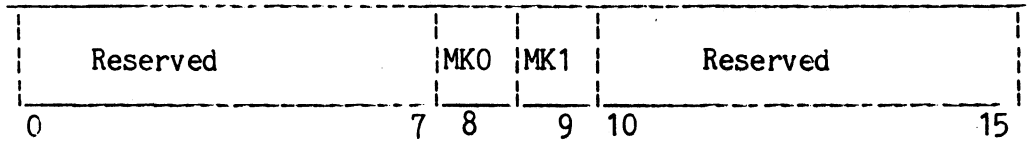
Reserved

XDCH - DCH map slots and operations supported

1 - always set to 1

MSK - prevents all devices connected to channel from interuppting the CPU

INT - Interrupt pending



I/O Channel Mask Register Format (MV/1000)

Reserved

MK0 - prevents all devices on channel 0 from interrupting CPU

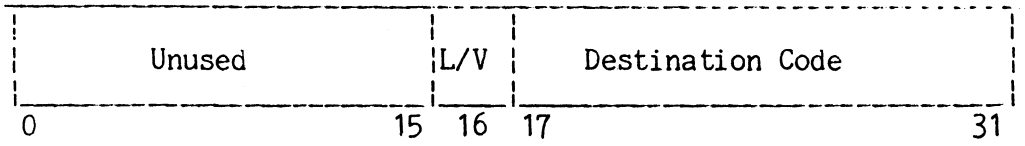
MK1 - prevents all devices on channel 1 from interrupting CPU

Micro Code

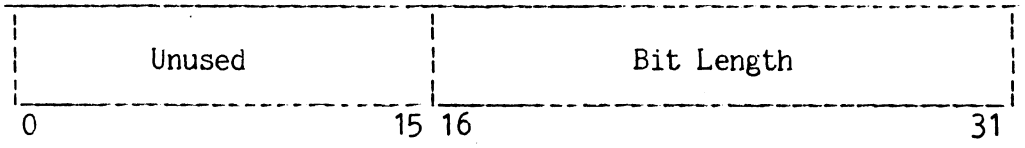
The heart of the MV hardware/software system is the micro code. This micro code contains the data paths used by the firmware to decode the instructions executable by each user. It is loaded from either disk or tape at cold startup time. There is one istruction which accesses this area of the machine. It is the LCS instruction. This instruction loads and verfies the soft internal states of the machine(for example, micro-store, decode rams, and scratch pad). In conjunction with bits 16 through 31 of three accumulators (AC0, AC1, AC2), the LCS instruction performs a load and verify, or verify only, using the contents of a micro code file.

AC0 contains the load and verify, or verify only, argument, and the destination code; AC1 contains the bit length of the code data; and AC2 contains a pointer to the first block of data.

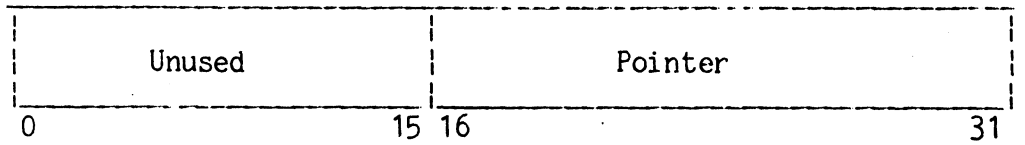
AC0



AC1



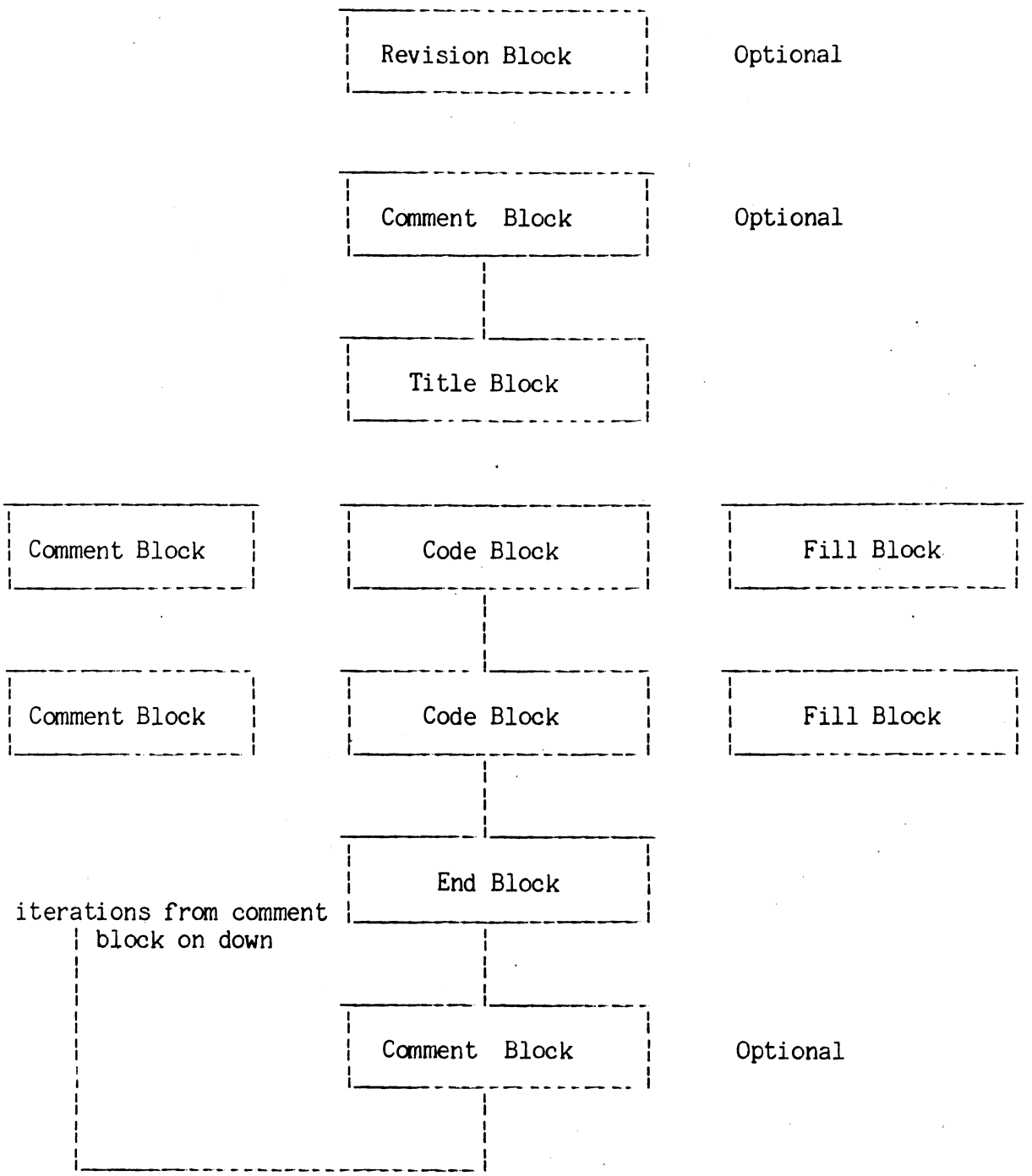
AC2



AC #	Contents	Meaning
0	L/V	Load/Verify option 0 implies load and verify 1 implies verify only
	destination code	Code for where the data is to be loaded
1	bit length	Bit length of code data
2	Pointer	Pointer to first block of data

Micro code File Format

The micro code file format contains data for use in various parts of the machine's state. The micro code format is block-oriented format (arranged into packets or blocks) that contains a description of the size of the block and the type of data it contains.



Micro code Block Format

Each micro code file must begin with a Title block and finish with an End block(Title/End block pair). Fill and Code blocks must be placed between the Title/end block pair. The Revision block precedes the first Title block. Comment blocks can appear anywhere within the micrc code file.

Kernel Functionality

The kernel is the minimum set of micro code necessary for the machine to function properly. With the kernel instruction set (including the LCS instruction) the processor can read in target micro code from an I/O device (using kernel I/O instructions) and then load this micro code into control store using the LCS instruction.

Because there is a 16k-word limit to the amount of data that can be loaded with a single LCS instruction, it may take several iterations of accessing the I/O device and executing the LCS instruction to completely change the machine from the kernel to the target.

In this chapter the MV hardware was introduced along with the System Control Processor and Micro code. The following chapter will discuss the AOS/VS kernel.

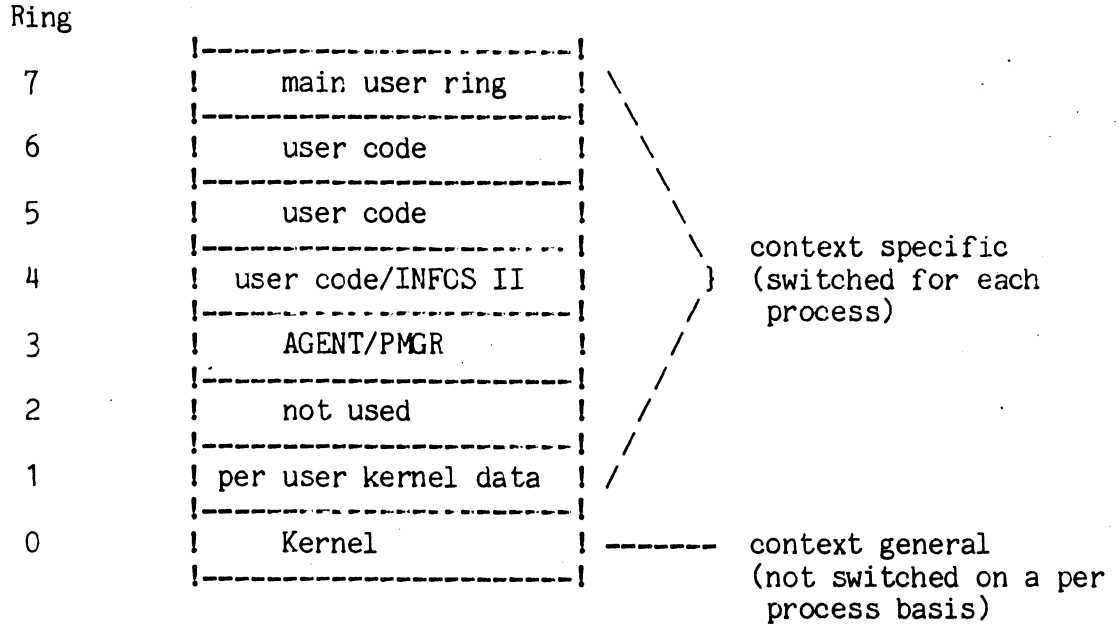
CHAPTER 3 -- AOS/VS KERNEL and DATA STRUCTURES (AOS/VS revision 5.00)

This chapter deals with the layout of memory for ring 0 and ring 1 of the operating system. It also describes the data bases, queues and stacks used by the kernel of the operating system. The operating system can be broken into four pieces. They are:(1) base system, (2)memory management, (3)processor management.and (4)I/O device drivers. The first three pieces will be discused in this chapter. The I/O device drivers do not need discussion as the listings are pretty complete.

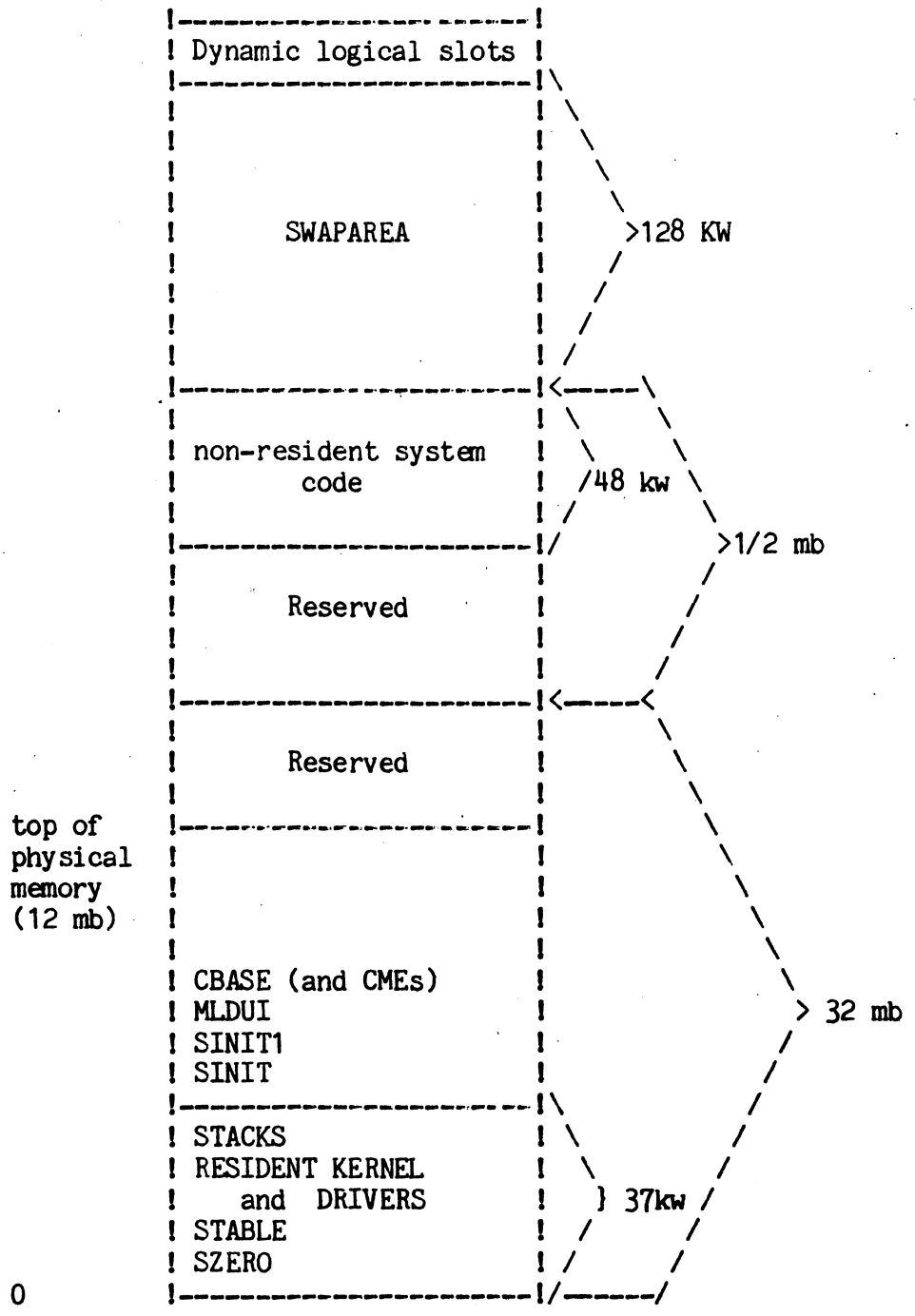
The rings

The following section describes the ring structure found in the MV hardware. and diagrams the layout of the two system rings.

General memory layout



Ring 0 memory structure



RING 1 memory layout

Ring 1 is defined as the context dependant system ring. For each process, ring 1 is unique in contents, but the same in structure. This implies that the first user CCB page for any user is located at the same logical address in that users context. This obviously simplifies the lookup of information in the databases.

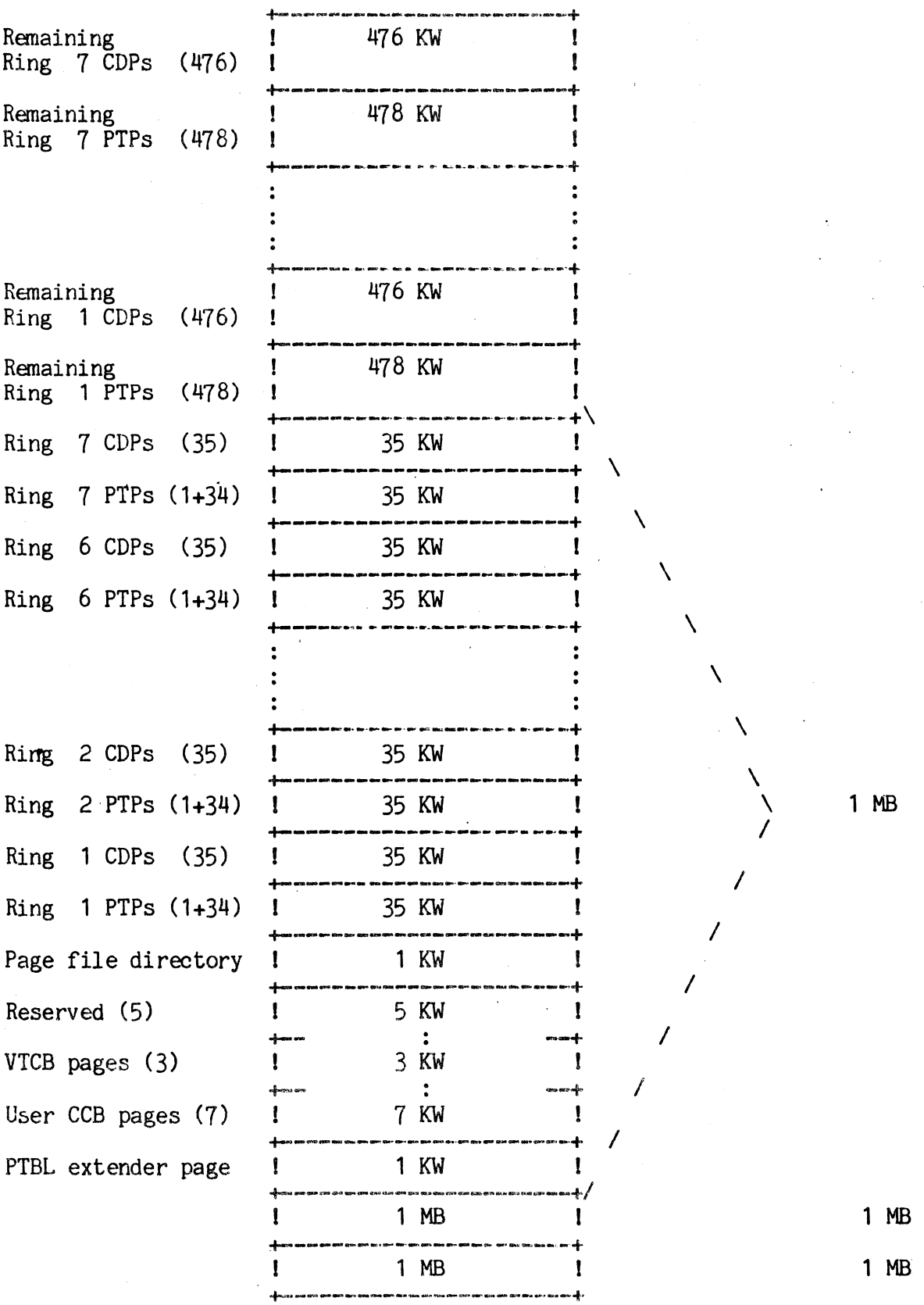
The PTPs and CDPs for all context switched rings (1-7) are located in ring 1. The memory in ring 1 is like any other ring, ie. it requires high and low PTPs and CDPs to describe the logical/physical relationship of its memory. This implies that ring 1 is self defining, or in other words, the PTPs and CDPs that define ring 1 are themselves located in ring 1.

To help localize PTPs and CDPs, and therefore keep the number of page 1 PTPs required to define the memory to hold the other ring PTPs and CDPs, the logical memory of each of rings 1-7 are divided into two groups. The third MB of ring 1 contains the PTPs and CDPs to describe up to 34 MBs of each ring. The databases required to do this are:

One high level PTP per ring
Up to 34 low level PTPs per ring
Up to 34 CDPs per ring

If a process' ring requires more than 34 MB, the remaining PTPs and CDPs will be built in a fixed location in the 4 and up MBs.

The diagram on the next page represents the ring 1 structure. Note that the first RSRVD K words of the 3rd MB are dedicated to the PEXTN, the user CCBS. some reserved locations, and the page file directory.


Remaining Ring 7 CDPs (476)
476 KW
Remaining Ring 7 PTPs (478)
478 KW
Remaining Ring 1 CDPs (476)
476 KW
Remaining Ring 1 PTPs (478)
478 KW
Ring 7 CDPs (35)
35 KW
Ring 7 PTPs (1+34)
35 KW
Ring 6 CDPs (35)
35 KW
Ring 6 PTPs (1+34)
35 KW
Ring 2 CDPs (35)
35 KW
Ring 2 PTPs (1+34)
35 KW
Ring 1 CDPs (35)
35 KW
Ring 1 PTPs (1+34)
35 KW
Page file directory
1 KW
Reserved (5)
5 KW
VTCB pages (3)
3 KW
User CCB pages (7)
7 KW
PTBL extender page
1 KW
1 MB
1 MB
1 MB
1 MB

Memory chains

GSMEM

GSMEM is the term used to describe the pool of various size blocks of general system memory. There are 8 sizes of GSMEM blocks, and therefore 8 different chains:

FC8	The chain of free 8 word chunks (blocks)
FC16	The chain of free 16 word chunks (blocks)
FC32	etc.
FC64	
FC128	
FC256	
FC512	
FC1024	The chain of free 1024. blocks (or pages)

GSMEM is also managed using a modifed buddy system (described below).

The Modified Buddy System

GSMEM chains are managed using a modification of the buddy system described in Knuth. "The Art of Computer Programming", vol 1. An explanation, by way of example, as to how it works is as follows:

Assume we need a chunk of 36 words of GSMEM to hold a database. First, we round 36 up to the next multiple of 8, which is 40. We next allocate a chunk of memory from the FC64 chain, passing its address to the routine requiring the memory. Since the database will only occupy the first 40 words, we break the remaining 24 words up into two chunks, one of 16 words, the other of 8 words, and put the addresses of these chunks onto the FC16 and FC8 chains respectively. If there are no chunks on the FC64 chain, we break the first entry on the FC128 chain into two 64 word blocks and put their addresses on the FC64 chain (we now have something on the FC64 chain and can proceed as above). If there are no entries on the FC128 chain, split the FC256 chain and continue. If there are no entries on the FC256 chain, try the FC512, and finally the FC1024 chain. If there are no free blocks on the 1024 chain, we either pass back an error to the calling routine if the routine can pend, or ... we are in trouble.

When we are done using the database, we return the memory to GSMEM, and attempt to regroup the block into larger blocks. However, a block can only regroup with the chunk it was split from ... it's buddy. The determination of a buddy is done using the following algorithm:

GSMEM block address size = 8. buddy address

```
130  =  001 011 000  XOR  000 001 000 = 001 010 000  =  120
120  =  001 010 000  XOR  000 001 000 = 001 011 000  =  130
110  =  001 001 000  XOR  000 001 000 = 001 000 000  =  100
100  =  001 000 000  XOR  000 001 000 = 001 001 000  =  110
```

GSMEM block address size = 16. buddy address

```
120  =  001 010 000  XOR  000 010 000 = 001 000 000  =  100
100  =  001 000 000  XOR  000 010 000 = 001 010 000  =  120
```

In general:

Buddy address = (block address) XOR (size of block)

Again it is important to stress that a block of GSMEM will only combine with its buddy.

LRU

The LRU chain in AOS/VS is composed of pages of memory that have a use count of 0. These can be either shared or unshared. When the PFF algorithm (see below) removes pages from a user's working set, the pages are put at the end of the LRU. If the user faults in the page before the page is removed from the LRU (the CME will have a status bit set saying that the page is on the LRU, a search is not necessary), the page will be unlinked from the list, and put back into the user's working set (the CME also contains the logical address/PID number of the user using the page to prevent a page from being removed from the LRU and put back on before the original user requests the page again).

When AOS/VS requires memory, it will look at the LRU (from the beginning, or oldest page) if nothing is available on the FC1024 chain.

Process memory chain (PSMEMQ)

Each process table extender has a doubly linked list of memory that has been allocated for a process, but not assigned. As an example, when the GCORE routine (called when a process is selected for swapin) gets the required page frames for a process the pages will be linked onto PSMEMQ. Later. when pages are needed for the process, a special routine (PGMBLK) will be called which will
look first at the process memory queue.

The queue is made up of chunks of contiquous memory linked together through their first words.

	offset	
PMFL.W	0	forward link
PMBL.W	2	backward link
PMST	4	status bit 0 - LPA memory area 1 - SPH memory area
PMSIZ	5	memory size (usually 1024.)

Databases

The following databases are used in managing AOS/VS memory.

CMEs

There is one CME (core memory entry) associated with each physical page of memory (each page frame), and the database serves as the principle descriptor for that page. The CMEs are allocated at SINIT time after the SINIT code determines the size of the ECLIPSE MV's physical memory (obtained from the CPUID). There is a page 0 location called CBASE.W that points to the base of the CMEs, which are allocated contiguously from that point. Therefore, the formula to find a CME is:

CBASE.W + (page # * CMPLN) where CMPLN is the size of a CME

The CME looks like this:

	offset	
CMPFL.W	0	Forward link if on the LRU
CMPBL.W	2	Backward link if on the LRU
CMPST	4	Status of the page
		bit - 0 I/O in progress
		1 unused
		2 this is a shared page
		3 I/O error detected on page I/O
		4 page in use by AOS/VS
		5 CDE page
		6 release to LRU list when count = 0
		7 call unpend when I/O completes
		8 page is on LRU
		9 PTE page
		10 level 2 PTE page
		11 page is modified
		12 flush waiting for page CCB unlock
		13 another path waiting for CME flush
		14 unsued
		15 unused
CMPUC	5	use count
CMPID	5	PID (when on LRU)
CMPWC	6	wired count
CMPPT.W	7	various information
		Physical page # when on free chain
		SPH address if a shared page
		FCB header address if a FCB page
		LPA header address if a LPA page
		if an unshared user page
		bits 0 - 21 logical page #
		22 - 31 PID
CMTIM.W	11	RTIM.W value at last fault (time stamp)
CMPGNR	13	physical page number corresponding to this CME

The current length of a CME is CMPLN (12.) words.

Note that CMPUC and CMPID are defined as the same offsets

VCME

Virtual CMEs (VCME) are written to the swap file along with the memory image when a user swaps out. They are used to reconstruct the CMEs associated with the user when the user swaps in. They are 8 words long.

	offset	
VCMPFL.W	0	forward link
VCMPBL.W	2	backward link
VCMPST	4	status (same as CME status CMPST)
VCMPWC	5	wired count
VCMPPT.W	6	logical address

SPH

Shared page headers (SPH) are used to define a shared page. There is one SPH per shared page independent of how many processes are sharing the page. SPHs are linked off of the FCB (file control block) for the file in which the page is found.

	offset	
CFFLK.W	0	Forward FCB link
CFBLK.W	2	Backward FCB link
CMMAP	4	physical page number
CMLAH.W	5	Disk logical address (block number)
CMFCB.W	7	parent FCB address
CFBLH.W	11	File logical address (block number)
CMLPQ.W	13	LPA queue descriptor (4 words)
SPHST	17	shared page status word (bit 0 - page is locked bit) (bit 1 - someone is waiting on lock)
CMHLKF.W	20	system wide hash link (forward)
CMHLKB.W	22	system wide hash link (backward)

SPHs are SPHLEN (20.) words long.

LPAs

LPAs, logical page associators, are used to associate a logical page in a process's address space with a shared page. LPAs are grouped in pages reserved for there use. Additional pages are allocated on demand and returned when not needed. LPAs are pointed to by the SPH they are associated with.

	offset	
LPFLNK.W	0	Forward link off SPH
LPBLNK.W	2	Backward link off SPH
LPLOG.W	4	Logical address
LPWC.W	4	temporary storage location
LPPID	6	PID
LPWC	7	Wired count (system+user)

Note that LPWC.W and LPLOG.W are define the same offsets. Each LPA is currently LPLEN(8.) words long, and there are NUMLPA (128.) LPAs per LPA page.

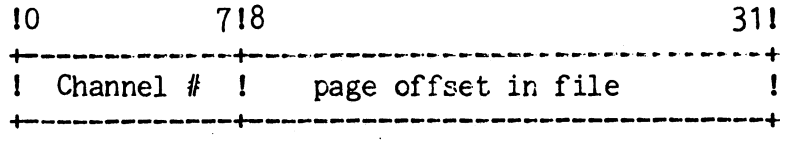
Each LPA page has associated with it a descriptor. These desciptors are linked off of LPACH.W and SPHCH.W and look like this:

	offset	
PGDFL.W	0	Forward link (off LPACH.W)
PGDBL.W	2	Backward link (off LPACH.W)
PGFRAME	4	Physical page #
PGUSC	5	Number of LPAs in use on this page
PGFFL.W	6	Head of free LPA chain on this page
PGFBL.W	10	Tail of free LPA chain on this page

LPA page descriptors are 10. words long

CDEs (control directory entry)

For each shared data page in a user's address space, it is necessary to map the physical page to the corresponding file and disk address. This is done with the CDE. A CDE is a double word database found in ring 1 and is managed in the same way PTEs are.



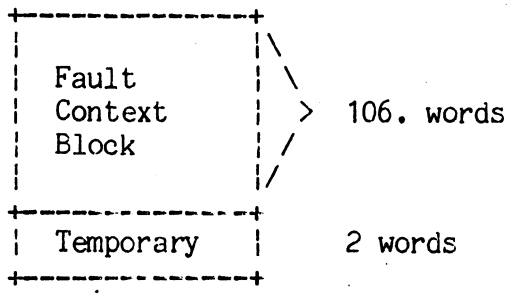
CDEs are collected in Control Directory Pages (CDPs). A count is maintained in the CME for a CDP of how many CDEs are currently defined on the page.

VTCB (Virtual Task Control Blocks)

VTCBs are located in ring 1 of each user. They contain the information that must be preserved during a page fault on the corresponding TCB. Therefore, there is the potential of 32 VTCBs. The first five VTCBs are located in the PEXTN page (in the area once occupied by the per process stack). The remaining VTCBs are allocated when needed in ring 1, 9. per page, thus requiring a possible 3 pages to store the complete VTCB group.

At the time of a page fault, the MV processor will store the current fault context block in the corresponding area of the VTCB (pointed to in page 0). In addition, the VTCB has a two word save area. The use of the area is discussed later in this chapter.

Each VTCB is structured as follows:



VTCXB.W	0	fault context block
VTSTK.W	162	stack block

VTCBLN.W is the length of the block (108. words)
MXVTCB is the maximum number of VTCB's (32.)
VTCBPPG is the number of VTCB's per page (9.)
SCVTCBEX is number of VTCB's in PT extender (5.)

System page zero

The following is a summary of important page 0 locations, with a brief description of their functions

PRSTK.W: points to the per processor stack
SUBSL: -320. -(# of PIT ticks in a sub-slice)
MAXRCB: 20 maximum # of resident control blocks

GARAY: The ring 0 gate array. These are the gates:

Entry point + high gate	function
RUNST+?ARING	schedule a task
SYST+?ARING	system call
PPNIC+?PRING	PMGR panic
UINTR+?URING	?IXIT
DEBBRX+?ARING	enter debugger from BKPT in ?URING
DEBBRZ+?ARING	debugger start up from ?URING
RUNIT+?ARING	?IMSG gate
RUNIS+?ARING	?RESCHED gate
SIGNL+?URING	?SIGNL gate
WTSIG+?URING	?WTSIG gate
SIGWT+?URING	?SIGWT gate
NSIGNL+?ARING	New ?SIGNAL gate for information to GURU
NWTSIG+?ARING	New ?WTSWIG gate for PMGR sched tuning (ring 3)
NSIGWT+?ARING	New ?SIGWT gate for PMGR sched tuning (ring 3)
GATENUM = 14.	number of gates

?ARING = ?PRING = 3 in the ring field
?URING = 7 in the ring field

COPYR: The Data General Copyright

NGHBT: Name of the AGENT file (AGENT.PR)
PCTBL: Unsafe panics table
INCHK: In checksum loop indicator
SYSIN: In system flage 1=in system/0=in user
INTLV: current interrupt level (0=base, n=level. define by the MV hardware as page 0, ring 0, offset 0.

TSPTB.W:Time slice end flag
TSLSV: Temporary time slice save area (for storage during interrupts)
CTSK.W: The current executing task
CMAP.W: Pointer to the address of the current user map
TODH.W: Time of day in seconds
RTIM.W: Page fault relative time (incremented at each page fault)

MPMGR: Master peripheral manager
DCTCH.W:Unit DCT chain

CC.W: Currently executing ELQUE entry
OCC.W: Old CC.W

SCPCL.W:SCP/HOST communication buffer list

Important offsets in STABLE.SR

PIDBT	Pid usage bit table (bit set indicates PID in use)
PIDTB.W	Pointer to the process table address table
PIDLN	Length of the process table table (initially 16.)
SKLTN	Skeleton (universal) system indicator (-1 -> not)
SHTDN	+1 system is shutting down
AOSBT	-1 -> abnormal shutdown / 0 -> normal shutdown
ESDSW	In ESD indicator
OVFAH.W	Overlay file logical disk address
RTCCB	31 words reserved for the system root CCB
PIFCID.W	The Process Information File (PIF) CID (channel ID)
PERCID.W	The :PER CID
NETCID.W	The :NET CID
HIFCID.W	The Host ID file CID
LHID	local host ID
LHNAM	local host name (16. words long)
PPBCH.W	protected file permission block page link(forward)
PPBCHB.W	protected file permission block page link(backward)
CXPBLK.W	connection table/protected shared file lock word
CNXTB.W	address of the connection table (0 -> not defined)
CPUID.W	cpu id and micro code word
SSBRTAB:	ring 0 SBR
SBRTAB=SBR1	ring 1 SBR (and start of the context dependant SBRs)
SBR2-SBR7	ring 2 - ring 7 SBR
REFSIZ	size of the referenced bit matrix (set up by SINIT)
.SFILE.W	pointer to the PAGE file table
HISLS.W	Base of the active histogram chain
CKSUM	Current checksum value
OVMIN	minimum number of overlays allowed in memory
CXFLAG	context block format flag initialized to MV/8 or MV/6 format
INITF	in initialization flag (1 -> still initializing)
DCHN.W	Base of the active delay chain
CRUN.W	currently running PTBL

N_T_RUN.W	highest priority entity ready to run
OLDNTRUN	last occupier of N_T_RUN.W
NTRUNDUP	indicator that there was a second structure which had the same PNQF as the curent structure in N_T_RUN.W
WTSCON	flag which shows weighted scheduling desired
WTSCINT	number of seconds to wait for examination to determine if weighted scheduling to be used
WTSCINTL	nuber of seconds in the time interval which elapses between checking cpu utilization
WTSCDCC	counter of the number of seconds of the duty cycle remaining
WTSCDCCL	duty cycle number of seconds of a time interval during which weighted scheduling is enabled.
WTSCCYC	number of times we will get to the end of ELQUE while looking for someone who is both ready to run and who has not run this interval when this counter goes to 0 shut off WTSGO
WTSCCYCL	number of times we will get to the end of ELQUE while looking for a PTBL which is both ready to run and not run this interval
WTSGO	tested after each structure has been selected via normal scheduling. If 1 then we are in the time interval after having found we are in cpu contention.
WTSCIDL	Simple counter inc'd in the idle loop used to register the amount of activity in the idle loop.
WTSCIDLL	If this number of counts is recorded in the idle loop during a time interval then weighted scheduling should begin.
SSTKCT	Number of free group 2 control blocks
SSTKQ	Free group 2 control block queue
RSTKCT	Number of free group 1 control blocks
RSTKQ	Free group 1 control block queue
G1	high priority for group 1 processes
G2	high priority for group 2 processes
G3	low priority for group 3 processes
G4	high priority for group 3 processes
QMIDDLE	PNQF at the 'middle' of group 2
QMIDCNT	counter of # of q's from head/tail if positive more q's from head if negative more q's from tail
G1RANGE	group 1 range set up at sinit time
G2RANGE	group 2 range set up at sinit time
PRANGE	values for entire priority range
HIBLK	highest block
LLOGICAL	last logical page in use
SCTBL.W	Address of system call count table
LSWPAD	logical address of the swaparea
SWQD.W	Physical address of the queue descriptor for VCME
SPQD.W	Physical address of the queue descriptor for VCME
SVQD.W	Physical address of the queue descriptor for VCME
SBQD.W	Physical address of the queue descriptor for VCME

SWPSI	global count of swapins
SWPSO	global count of swapouts
TOQUE	timeout request queue
BFLRU.W	list of free buffers in LRU order
BTAIL.W	
BUFCN	Number of buffers currently on the free buffer LRU
FC8	Free 8 word GSMEM queue (.W)
FC16	Free 16 word GSMEM queue (.W)
FC32	
FC64	
FC128	
FC256	
FC512	Free 512 word GSMEM queue (.W)
FC1024	Free 1024 word GSMEM queue (.W)
CBASE.W	address of the base of CMEs
LRUCH.W	Chain of CMEs describing shared pages with a use count of zero
UPSYS.W	Queue of high address system pages (overlays) in memory
SPHCH.W	Chain of SPH page descriptors
LPACH.W	Chain of LPA page descriptors
FCBCH.W	Chain of FCB page descriptors
IFCB	FCB/CCB unique ID (initially 1, incremented each time a FCB is created)
SMFLG	Core manager request flag
CMFLG	System manager request flag (this is not a typo)
MKEY	System memory key (incremented when memory is released)
BIAS	Minimum number of non-interactive processes in memory
HBIAS	Maximum number of non-interactive processes in memory
ELQUE	The major queue bases
IEBLK	
IEQUE	
IESWP	
IERES	
BLKQ	
MBLKQ	
DMTSK	The disk manager control block
CMTSK	The core manager control block
RTPTB	The root process table
BTBL	The interrupt vector dispatch table

MDCH1-MDCH8 The data channel map control slot (bit =1 -> free slot)

FLTBK.W Fault block for system

fXTNPG: PEXTN address in ring 1 (constant for all users)
PFDADR: Page file directory address in ring 1

VTCBTAB: Address of the VTCBs in ring 1

INUSER.W # of interrupts in non-ring 0 code
INSYST.W # of interrupts in non-checksum loop
IDL.W # of interrupts in idle loop (checksum loop)

DFSPCNT The default maximum per ring IPC messages spooled
GSWOTM.W Time stamp of the last process picked by COREM for swapout

These offsets are related to faulting information gathered by the system.

DIRFLT.W Count of faults run directly.

INFLT Count of faults using a CB.

LSFLT.W Count of logical faults on system pages.

PENDFLT.W Count of faults that pended.

The following symbol is used for tuning.

IAMTUNED This item is the flagword to signify that the tuning program was run.

The following symbols are used by the terminal utilization utility (GURU)

GURUBTYP This is the mask showing the conditions under which to begin timing.

GURUETYP This is the mask showing the conditions under which to end timing.

GURUFLAG If on then we wish to use the GURU utility

GURUTB.W The address of the GURU buffer.

GURUBUCK The number of ticks in each bucket

Major databases

Process tables and process table extenders

Each process in AOS/VS has associated with it two major databases one called the process table (PTBL) and the other called the process table extender (PEXTN). The PTBL is always available in memory, while the PEXTN will swap out when the process does. Therefore, the PEXTN cannot contain any information vital to process scheduling and swapping. Examination of the PEXTN offsets will bear this out. Both of these databases are defined in PARS.

Below are some of the more important PTBL offsets:

PLNK.W	forward link for connecting the process table to one of the major queues
PBLNK.W	backward link for major queues
PSTAT	contains the most important status bits for the process
PNQF	Priority eNQue Factor - determines the order in which this PTBL will be enqueued (see chapter 4 for formula)
PPC.W	address to begin processing when this PTBL is given control
PKEY.W	key on which to unpend this process
PDAD.W, PSONP.W PSONL.W	father, son, and brother PTBL addresses
PFLAG - PFLG4	additional process status words
PEXTN.W	pointer to the process table extender
PSFDF.W,PSFDB.W	Spool file directory chain (forward/backward link)
PSFRC.W	Spool file entry count (1 byte per rings 4 - 7)
PIORR.W	outstanding IREC receive chain
PIPCS.W	IPC spool file CCB address
PID	the PID
PPRV	assigned priviledges (superuser/superprocess...) flags
PCMLK.W	link word for processes waiting for swapin or swapout

PSLEX	time slice exponent (see chapter 4 for calculation)
PDLNK.W,PDINH.W	offsets used in managing ?WDELAY system calls
SWCCB.W,PGCCB.W	swap and page file CCB addresses
PWSET	working set size
PWSSH	number of shared pages in the working set
PSRNG	server ring bit map (bits 0-7)
PTRGC	target system call count (# of system calls currently targeted at this process

Some of the more important PEXTN offsets:

PSQCT	number of active system calls for this process
PSQMX	maximum allowed number of calls for this process
PWCCB.W,PDCCB.W	address of the current directory CCBs (working/default)
PCTSK.W	address of this processes current TCB
PSWD.W	chain of TCBs waiting for CBs and stacks
PSL	number of PIT ticks left in the current subslice
PSBR1.W - PSWB7.W	The ring 1-7 SBRs associated with this process
PDFR.W	chain of tasks with outstanding ?WDELAYs
PSRCH.W	eight addresses of the searchlist CCBs
PUNM	eight words for the username
PPNM	the first 15 letters of the program name
PWSMIN,PSWMAX	the minimum/maximum working set
PCAPT	time constant used in PFF algorithm (not used)
PRFIL1.W - PRFIL7.W	address of the CCBs associated with the .PR files for rings 1 - 7
PMEMDIS	7 sets of 12. words each used to describe the memory in each ring (see PARS page 19)
PUCCBS	7 words of counters indicating the number of user CCBs in each of 7 corresponding user CCB memory pages

PRMUNP, PRMSHP number of removable unshared/shared pages

PRCCB.W address of the breakfile CCB if one is requested

PFRING ring of father that issues the ?PROC call

PSWOTM.W swapout start time stamp

PSI (Page, Swap, IPC)

The PSI data base contains a linked list of CCB's for open (hot) but no longer needed page, swap, and ipc files. These files are not needed because the process that owned them has since terminated. When a new process takes an associated pid number this database is accessed to find psi CCB's associated with a pid. This process saves the system overhead of opening and closing of system files each time a proc is done.

The symbols dealing with Page/Swap/Ipc start with the letters PSI.

PSIBEG Hot PSI pool queue despriptor.

PSIEMP.W Count of the number of times the optimization of procs failed because the pool was empty.

PSIFLG Hot PSI locking word.

PSIMAX Absolute maximum hot PSI pool size (20)

PSINOT.W Count of optimiztion failure do to the associated pid not being in the pool.

PSIOPN Current number of open PSI's.

PSIREQ Requested pool size (default 5)

PSITAB.W Address of the first word in PSI pool.

PSITOT Current hot PSI pool size.

PIDBT - The PID's Bit Table

PIDBT is a bit table containing a bit for each possible PID. Bit number I will be zero if no process currently exists with PID = I. Bits 0 and 3 are preallocated for the root process and for CLIBT.

PIDTB - The Table of PID's

PIDTB is the process-identifier-to-process-table-address conversion table. Its current size is kept in location PIDLN in page zero. PIDTB entries are defined as follows :

PIDTB(PID) = 0 if no process exists with that PID

PIDTB(PID) = processor table address if that PID exists

PIDTB is initially (at SINIT time) allocated in a 32 word GSMEM chunk. The first time the PID count exceeds the length of the table, a table twice the size (32 .. 64 .. 128 .. 256) is allocated out of GSMEM, the old table is copied into the lower half of the new one. and the old one is released to GSMEM. This is called from the PROC2 module.

Control blocks

Control blocks are used for cases in which the system needs a stack to handle a code path, and there is a possibility that the path will pend. These stacks may be allocated when a user makes a system call which requires a stack, or they may be allocated for system use (daemons).

There are three types of control blocks; resident, or those associated with resident or preemptible processes, swappable, or those associated with swappable processes, and three special CBs associated with the disk manager, core manager, and system manager. Each control block has its own stack and these will be discussed later when we deal with the subject of stacks.

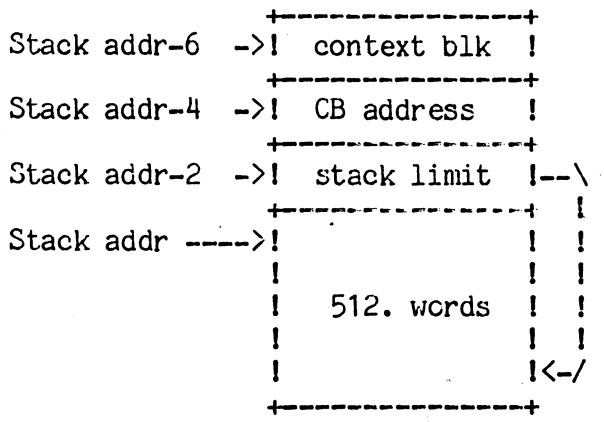
The first eight offsets of a control block (CB) matches those of a PTBL (thus allowing the two types of databases to be handled by the same queue search routines). It is through the first two links (PLNK.W and PBLNK.W) that CBs and PTBLs are linked onto the eligible queue. The other important offsets are:

CATCB.W	The address of the user TCB that the system call is running on behalf of (or 0 if a daemon or the core manager)
CBFEH.W	CB fatal error handler address- The address the system should pass control to on a trap within the system. If the CB can better handle the trap, it will ... otherwise we will panic
CMQWD.W	used only by the CMTSK CB. It is the start of the swap in / swapout queue.
CSTKC.W	The pointer to the stack base for the stack associated with this CB
CPTAD.W	The address of the PTBL that this CB is running on behalf of
CERWD	When a routine processing a CB request encounters an error, the error code is stored into this location. The CB dismissal routine will check this word, and if non zero, pass the value back in the TCBs AC0
CBDLS.W	Each CB has a DLS (dynamic logical slot) associated with it. This offset points to it.

Each control block has associated with it a fault context context block, a stack, and a dynamic logical slot. When the control block is selected as the entity to run, the MV's hardware registers are set up to point to the appropriate corresponding values for the control block. The

memory needed for a CB is allocated from GSMEM except for the first group 1 control block (CB000), the CMTSK CB, and the system manager task CB. The physical layout of the memory associated with a CB is as follows:

Each control block has associated with it a page fault context block, a stack, and a dynamic logical slot. When the control block is selected as the entity to give control to, the MV's hardware registers are set up to point to the appropriate corresponding values. The databases involved are linked together as follows:



Unallocated CBs (those not on ELQUE) are enqueued to either SSTKQ (if it is a swappable CB) or RSTKQ (if it is a resident CB) The maximum number of resident control blocks is specified in the page zero location MAXRCB (currently 16.)

Each time a resident process is proc'ed, the group 1 control block pool grows up to the maximum. Upon process termination, if the process terminating is resident, the pool will shrink.

The major AOS/VS scheduling queues

NOTE: Queues used by the AOS/VS diskworld are discussed in chapter 6.

AOS/VS maintains 7 major queues that are used in scheduling and scheduling related functions. These are:

ELQUE	the eligible non-blocked queue
BLKQ	the eligible blocked queue
MBLKQ	the eligible explicitly blocked queue
IEBLK	the ineligible blocked queue
IESWP	the ineligible swapped queue
IERES	the ineligible resident/preemptible queue
IEQUE	the queue of priority swapins for calls involving move bytes and IPCs

ELQUE is a special queue in that it contains both CBs and PTBLs. All other queues contain only PTBLs.

There are two offsets found in both the process table and the control block which are used to link items on the queues together, they are:

PLNK.W	which is the link to the next item on the queue.
PBLNK.W	which is the link to the previous item on the queue.

These queues (due to the common link word) are mutually exclusive, implying that a process can only be on one of the queues at a given time.

FLQUE - The Eligible Queue

FLQUE - This location, defined in STABLE, points to the head of the eligible queue, which is always the core manager control block address.

The eligible queue is a linked list containing the Core Manager, the system manager, any control blocks in use, and the process tables of any eligible processes. Subsequent links in the chain are linked both forward and backward through offsets in the process table, and all links are either control block or process table addresses.

The eligible queue is always headed by the Core Manager control block, and the last entry on the eligible queue is the address of the root process table. The order of entries on the queue is determined by the entries PNQF which will be discussed later.

ELINT - number of eligible interactive processes.

ELNON - number of eligible non-interactive processes.

These two counts are defined in STABLE. For AOS/VS, a non-interactive process is a swappable process having a time slice exponent of 6. An interactive process is any other swappable process.

BLKQ - The Blocked Queue

BLKQ - This location, defined in STABLE, contains the process table address of the process at the head of the blocked queue.

The blocked queue is a linked list of the process tables of all eligible processes that are currently blocked, but not explicitly blocked (waiting for son term, of by ?BLKPR). The chain is linked through offsets PLNK.W and PBLNK.W.

PTBLs on the BLKQ are ordered in FIFO order

The following locations are defined in STABLE :

SRBLC - number of blocked processes on the BLKQ

BLEND - the end of the BLKQ

MBLKQ -- The Explicitly Blocked Queue

MBLKQ - This location, defined in STABLE, contains the process table address of the process at the head of the explicitly blocked queue

The explicitly blocked queue is a linked list of the process tables of all processes that are explicitly blocked, and not swapped. A process is explicitly blocked if it was the target of a ?BLKPR or it executed a ?PROC and block. The chain is linked through offsets PLNK.W and PBLNK.W of the process tables in the queue.

PTBLs on the MBLKQ are in FIFO order

The following locations are defined in STABLE :

MBLKCN - number of blocked processes on the MBLKQ

IERES - The Ineligible Resident Queue

IERES - This location, defined in STABLE, contains the PTBL address of the process at the head of the ineligible resident queue. This queue links the process tables of any resident or preemptible type processes that have not been allocated memory (swapped out) and are not blocked. Resident type processes can only be on this queue when being created, or if they have just had their type changed to resident.

Because a process on this queue is swapped out, the process table extender and user status information in user's page zero as well as all the unshared portions of the user's process are not in memory.

PTBLs on the IERES queue are ordered by PNQF

IELRS - number of presently ineligible resident processes

IESFL - when non-zero, inhibits the scan of this queue.

These locations are defined in STABLE.

IESWP - The Ineligible Swappable Queue

IESWP - This location, defined in STABLE, contains the PTBL address of the process at the head of the ineligible swappable queue. The ineligible swappable queue is a doubly linked list of all the non-blocked swappable processes which are now swapped out

Because a process on this queue is swapped out, the process table extender and user status information in user's page zero as well as all the unshared portions of the user's process, are not in memory.

PTBLs on IESWP are ordered by PNQF

IELSW - number of presently ineligible swappable processes

IESFL - when non-zero, inhibits the scan of this queue.

These locations are defined in STABLE.

IEQUE -- high priority swapin queue

This location, defined in STABLE, contains the PTBL address of the first process on the IEQUE. The IEQUE is a queue of processes that must be swapped in in order to complete a byte move to a target. In AOS, the write is made directly to the swap file. In AOS/VS, we might have to write to the page file and not the swap file. In order to simplify and accelerate move bytes processing, the process that the system call is targeted at will be swapped in.

Processes on IEQUE will be temporarily given a PNQF of 0, thus accelerating the swapin processing

Processes on IEQUE are in FIFO order

IEBLK -- The ineligible blocked queue

IEBLK - This location, defined in STABLE, contains the PTBL address of the first process on the IEQUE. The IEQUE is a queue of processes that are blocked, and therefore not in need of swapping in. At the time that a process unblocks, it will be migrate to a different ineligible queue.

New processes are added to the front of the IEBLK queue.

IEBCN is a counter in page 0 of the number of processes on the IEBLK queue

Note : AOS/VS maintains four different queues of swapped processes, in order to better control the priority of certain swapins. The swapins on IEQUE are required to complete a system call (and therefore free up a CB) and are of the highest priority.

Next come the swapins of resident and preemptible processes, and finally those of swappable. Those processes on the IEBLK queue will not be swapped in until the become unblocked. The core manager (who is responsible for swapin) will not scan the IESWP queue unless the IERES queue is empty.

The minor AOS/VS scheduling queues

NOTE: Queues associated with the disk world are discussed in chapter 5.

There are three additional queues not discussed in the previous section that deal with processes or scheduling. These are:

DCHN.W	the chain of processes with outstanding delays
HISLS.W	the chain of processes with outstanding histograms
CMQWD.W	the chain of process that are waiting to be swapped (either in or out)

Unlike the case of the major queues, a process table can be on more than one of the minor queues at a given time. However, the process table must also be on one of the major queues if it is on a minor queue.

The minor queues are not linked through offsets PLNK.W and PBLNK.W (these are still used to link on the major queue). In the case of the DCHN.W queue, links are through offset PDLNK.W of the process table. In the case of the HISLS.W chain, links are through offset PHLNK.W of the process table extender. For the CMQWD.W chain, the link is offset PCMLK.W.

DCHN.W - The Delay Chain

DCHN.W This location, defined in STABLE, holds the PTBL address of the process at the head of the delay chain.

The delay chain is a queue used for keeping track of all of the tasks in the system currently doing a delay. A delay is the method by which a user task can pend for a specified time without tying up system resources.

The delay chain is actually two linked lists in one. The first one is pointed to by DCHN.W and is the linked list of process tables associated with processes that have one or more tasks doing delays. The second is a linked list of the process tables in the delay chain, including links to all the tasks in this given process that are currently doing delays.

The process tables in the delay chain queue are singly linked using offset PDLNK.W in the process table. The linked list is terminated by a minus one in the link word. Note that process tables found on the delay chain may also be found on various other chains such as the eligible queue or blocked queue.

Delaying tasks TCB's for each process are singly linked through offset ?TSYS of the TCB. The list is terminated by a minus one. The TCB's of each process are ordered by the amount of time left to delay. Those tasks with the least amount of time left to delay appear earlier on the chain. If two tasks delay for the same amount of time, the link word ?TSYS of the first task will have bit zero set (1B0).

The following five offsets are defined as process table offsets :

PDLNK.W forward process link offset.

PDINH.W number of real time clock ticks the first task is to delay

The following three offsets are defined in the process table extender :

PDFR.W start of the TCB delay chain.

?TSYS - forward task link offset, defined in the TCB of the task currently doing a delay.

HISLS.W - The Histogram Active Chain

HISLS.W This location, defined in STABLE, contains the PTBL address of the process at the head of the histogram queue.

The histogram queue is a linked list of all the processes which have initiated histogram creation via a ?IXHIST call. The histogram queue is a singly linked list off the associated processes process tables. The link offset is "PHLNK.W" and the link is terminated with a minus one. The information contained in the histogram is stored in the process table extender of the process making the "?IXHIST" call. Note that because of this the process making the call must be resident to insure that the process table extender is always in memory.

PHLNK.W forward link of the histogram queue, defined in the process table extender.

CMQWD.W - The Core Manager Request Queue

CMQWD.W This location is defined as a control block offset that only has meaning for the Core Manager control block, CMTSK. It is the beginning of the chain of processes that are enqueued to the Core Manager for swapping, either to be swapped in or swapped out. This offset contains a PTBL address. The queue is a singly linked list, through offset PCMLK.W of the process tables found on the queue.

Data Resources

We will also limit our description of data resources to a few items, the largest data resource in AOS/VS being the File System, subject of chapter 6. The data resources we chose to develop here the System Stacks and the system pageable pages.

The stacks

The ECLIPSE MV contains numerous predefined locations within page 0 ring 0 used to manipulate a stack. At any given point, one stack is defined as current. The following page 0 ring 0 locations are defined by the hardware:

Location	Use
0	the current interrupt level
4	vector stack pointer
6	vector stack limit
7	vector stack fault address
14	current stack fault address
20/21	WFP (current frame pointer)
22/23	WSP (current stack pointer)
24/25	WSL (current stack limit)
26/27	WSB (current stack base)
40	C/350 stack pointer
41	C/350 frame pointer
42	C/350 stack limit
43	C/350 stack fault address

In AOS/VS there are three types of stacks used by the system :

STACK TYPE	STACK BASE	USE
Interrupt stack	SS	when processing an interrupt, the current stack will be SS (set up by the XVCT instruction from base level)
Per processor stack	STK1	set up whenever we are in the normal (base level) AOS/VS codepath
Control block stack	pointed to by the CB	Set up when we run the control block

At any point in time one stack is current, and pointed to by the MV/8000 hardware stack registers defined in ring 0, page zero. The system always runs in ring 0, therefore all these stacks are defined in ring 0. These stacks are in effect only when the system is running. Before giving control to a user, the user's current stack is set up (this could be either a 32 bit stack or C/350 stack).

The interrupt stack

Location INTLV in SZERO is incremented at the start of interrupt handling, and is decremented by the interrupt dismissal code. When the system is at base level (not processing an interrupt), INTLV = 0. The hardware XVCT instruction will examine INTLV, and if the value is 0 at the time of the interrupt, it will save the current stack information in preassigned page 0 locations, and make the interrupt stack the current stack.

Interrupt stack definitions :

stack base: SS
stack limit: SSLMT
stack size: 512. words
stack is defined in module STKS
loc 0: INTLV, current interrupt level (0=base level)
loc 4: SS. vector stack pointer
loc 6: SSLMT. vector stack limit
loc 7: OVFLO. vector stack fault

The Per processor stack

The per processor stack acts as the universal stack for all system activity not related to one of the other system stacks. This stack being constantly re-initialized by code paths in the Scheduler and the Core Manager, it is necessary that all code using this stack be very careful that it will not depend on it in cases where it may be changed.

Per processor stack definitions :

stack base: STK1
stack limit: STK1N (stored at location STK1-1)

stack size: 384. words

stack is defined in module STKS
PRSTK: STK1 is a label defined in SZERO

System pageable pages

System pageable pages contain system code that is usually not permanently resident but is brought into memory at some point in time. These pages contain code similar to that found in the overlays of AOS. The system reserves enough pages to contain all of the overlays in the 33 megabyte of ring 0.

The only I/O operations to the system pageable pages are read operations. These pages contain pure code; they are never modified and thus never need to be written back to the system file.

The system maintains a queue called UPSYS which contains the CME for each physical page associated with a system pageable page.

For more information look at the modules SZERO.LS and STABLE.LS which are in the appendix of this manual.

The Kernel of AOS/VS comprises three major sections of the operating system. These three major sections are: (1) base system, (2) memory management. (3) process management.

The base system is made up of modules which handle various system services which no one process needs. It handles operator console I/O, scheduling of tasks and processes, interrupt dispatching, power failure detection and miscellaneous subroutines.

The following modules make up the basic system.

CONIO - console I/O routines for the master console
entry points:
OUTMES - print a message on op console
GETCHAR - read a character
PUT - print a character
BINAS1 - print a single binary word
BINAS2 - print a double binary word

CRESOLVE - resolves unloaded modules during link to -1

DUMMY - dummy module for the AGENT

DVRS - powerfail restart routine for MAGTAPE, MCA, IOP, IAC

INTS - interruprt service routines
entry points:
INTS - interrupt service entry
OVFLO - stack overflow
UDEX - set up for user device driver
IUD - undefined device service
UINTR - return from user interrupt
IRTC - RTC interrupt service
IUPSC - UPSC interrupt service
IPIT - PIT interrupt service
IWKUP - wake up a PTBL
PCWKUP - wake up a PTBL, don't check N_T_RUN.W
DISMISS - dismiss an interrupt
PFLTPIT - reschedule on a page fault at subslice end
ORWPAT1 - patch space for weighted scheduler
IRTSTL - weighted scheduler return
UTRAP.2 - pop to user that may have trapped

ORWELL - weighted scheduler
entry points:
ORWELL - entry for real time clock handler
ORWELL40 - scheduler entry to see if PTBL runnable
ORWELL60 - entry for ELQUE search on failure to find anyone ready-to-run

PANIC - system panic handler
entry points:
PNIC - panic the system
EPNIC - panic the system don't clear regs 5 - 8
PNIC2 - panic entry for the PMGR
PPNIC - panic entry for LPMGR
STRP - system validity panic entry
SHUTPR - OP CLI trap handler

PWRFAIL - power fail restart handler
entry points:
PWRFL - system power fail restart entry

```
                    UIPFL - user device restart return
RESOLVE  - resolve more unloaded modules from sysgen
SCHED    - system process scheduler
           entry points:
                    SMON - scan ELQUE from top setting SYSIN
                    SMON0 - scan ELQUE from the top
                    SMON2 - run same process again
                    SMON3 - run next_to_run process
                    SMOND - enter checksum loop
                    TACT - activate a control block
                    PENTR - schedule start a process
                    PCALL - same as PENTR
                    MAPCON - map ring 1 - 7 context
                    REMAPCON - remap ring 1 - 7 context
                    RUNEX - check if ready to run next
                    RUNST - start a reschedule of a task
                    PSCHD - find and start up a processes task
                    TSKEX - set up a stck fault block
                    EXVFL - flag the save for extended variable task
                    PEND - pend a process table for an event
                    UNPEND - unpend a process table
                    STKST - set up a process stack
                    RUNIT - special gate for ?IMSG to save state
                    RUNIS - special gate for ?RESCHED to save state
                    TSKSAV - save task state for all valid rings
                    EVENT - determine if PTABLE/CB really ready to
                            run
                    ORWPAT2 - patch for weighted scheduler
                    ORWPAT3 - patch for weighted scheduler
                    ORWPAT4 - patch for weighted scheduler
                    ORWPAT5 - patch for weighted scheduler
                    SMONDD - entry to idle loop
                    PCBST - entry to run in user space
                    RSTPR - set up runtime for direct page faults
                    UTRAP.1 - inward address trap user assumed to be
                              the cause
SCMOD    - routines to process direct system calls
           entry points:
                    TIMEQ - enqueue a ?WDELAY request
                    SIGNAL - signal the system from a process
                    RMBTU - move bytes to user from caller
                    RMBFU - move bytes from user to caller
                    SIGNL - signl a task or process
                    WTSIG - wait for a signl
                    SIGWT - signl a task and wait for a signl
                    UNLINK - remove a PTBL from a delay chain
                    LINKIN - insert a PTBL on a delay chain
                    TPID - check validity of a pid
                    NSIGNL - new signal call to pass info to
                             GURU
                    NWTSIG - new wtsig for pmgr I/O scheduling
                    NSIGWT - new sigwt for pmgr I/O scheduling
SCPER    - SCP error processor
           entry point:
                    SCPER.P - read out the error from the SCP
SCPRC    - system call processor
           entry points:
```

SYST - system entry point
FSYST - page fault handling entry
CALLS - bkpt for kernel syscalls
IOCRG - charge for I/O blks
RETER - system error returner
RETE2 - system error returner
GETERR - get current system error
TCBAD - roached TCB error handler
MAXSYS - highest sytem call number
MCCT.W - system call dispatch table
SNTIO - enque TCB to PTBL extender
TRTN - return from CBLK processing
DSQCT - decrement active call count
DIRRS - direct call handler
SYSMGRPV - check for sysmgr privlege
DBTB - direct call bit table
CWTB - parallel call bit table
TGRTN - good return from direct page faults

SGSUB1 - system subroutines
entry points:

MUIOPGR - mark user I/O pages for read rq
MUIOPGW - mark user I/O pages for write rq
LDHSC - load hi-speed channel map
SSOVF - system stack fault handler
MASK - add to acurrent mask
UNMASK - recover previous mask
CDLD - load DCH maps A and B
DLD - load DCH map C
EDLD - load DCH map D
SWAMP - set up data channel map
ALDCH - allocate and load PMGR slots
ALSLOT - all purpose I/O slot allocation
DEALS - all purpose I/O deallocation

SGSUB2 - memory management/scheduler part of system subroutines
entry points:

SGSUB2 - module name
INSHARED - verify address in shared area
SAVST - save process state
GMDESA - get memory descriptor address
CLEAR - clear core
IMCLR - clear user/system area
MCLR - clear user/system area
CHNSHFT - shift the channel for MV10000
OVCHNL - adjust frame size to fit frame of target
TCBRS - restart a task
NQTCB - enque a TCB to a PTBL
KCINT - update caller's page-sec integral
DBTRP - format a data base trap in PTBL
SSTRP - set up fatal termination for a process
PFITER - dequeue a PTBL and requeue with new priority
PTREE - block the entire process tree
PBITS - format PTBL flag to control termination
TCBCK - validate a TCB address
UPHSET - add a page to working set bit array
ARNGDES - inc working set descriptor count

SRNGDES - dec working set descriptor count
MODPRIO - modify priority using new structure
CHRPSWA - change resident/preemptable to
CHSWARP - change swapable process to resident/preemptable
MAPSWAIN - map swappable priority on input
MAPSWAOU - map swappable priority on output

STRAP - system trap routines entry points:
STRAP - module name
MAGIC - protection fault handler
BRKPT - breakpoint handler (panic 14)
FPFLT - floating point fault handler (panic 7)
COMFLT - commercial fault handler (panic 7)
FIXOV - fixed point overflow handler (panic 7)
SSOVFE - Eclipse stack fault handler

SYSER - system error handler
entry points:
UNERR.P - unit error reporter

SYSMGR - xyzzy system manager
entry points:
SYSMGR - init system manager's control block
SMINT - init sytem manager's control block
SWAKE - force a re-schedule of the system manager
SWAK1 - force a re-schedule of the system with interrupts disabled

UWART - entry point if system debugger in use

The AOS/VS scheduler

There are three distinct parts to the AOS/VS scheduler. These are:

1. The AOS/VS process scheduler (SMON in SCHED). Its responsibility is to determine if any process table (PTBL) or control block (CB) is ready to run, and if so, run it. If not, it will perform a checksum on various AOS/VS constants.

2. The AOS/VS task scheduler (PSCHD in SCHED). Its purpose in life is to decide which user TCB is to run when the process gets control of the CPU.

3. COREM (Core Manager), which in AOS/VS is not actually part of the scheduler, but does scheduler oriented activities. COREM handles swapping, some queue migration and resident process unpending if the process was waiting for memory.

Definitions

An ELIGIBLE process is a process that has at least one page of its context in memory.

An INELIGIBLE process is a process whose context exists only in the swap file. An INELIGIBLE process must be first made eligible (i.e. it must be swapped into memory) before it can be scheduled by the system to run.

A BLOCKED process is a process that has been pended for some period of time. A BLOCKED process may be either ELIGIBLE or INELIGIBLE

- A process becomes blocked as soon as the scheduler finds there are no ready TCBs for that process, or by the explicit ?BLKPR system call.

Queues

AOS/VS maintains 6 major queues. The databases on these queues are of the following type:

PROCESS TABLES - Contain enough information about a user process to allow the scheduler to both make a decision about scheduling the user, and then give control of the CPU to that user process (see PARS.SR).

CONTROL BLOCKS - Contain the static (non-stack) state of a path within the system. These paths are active to either process user system calls, or daemons created by the system. If the user system call is non-direct, or the daemon request

requires a stack, a system stack will contain the dynamic state of this path. The dynamic state of a path includes all subroutine return addresses, and temporary variable data used by the path (see PARS.SR). There are two basic types of control blocks, swappable (those started on behalf of a swappable process) or resident (those initiated on behalf of a resident or preemptible process)

The queues and the pointers to the queue head are:

ELQUE: The eligible queue of process tables and control blocks. This is the primary queue used by the scheduler.

* The DISK MANAGER Control Block is always first on this queue.

* The CORE MANAGER Control Block is always second on this queue.

* The SYSTEM MANAGER Control block is always third.

* The active resident Control Blocks are next. These are in FIFO order by time.

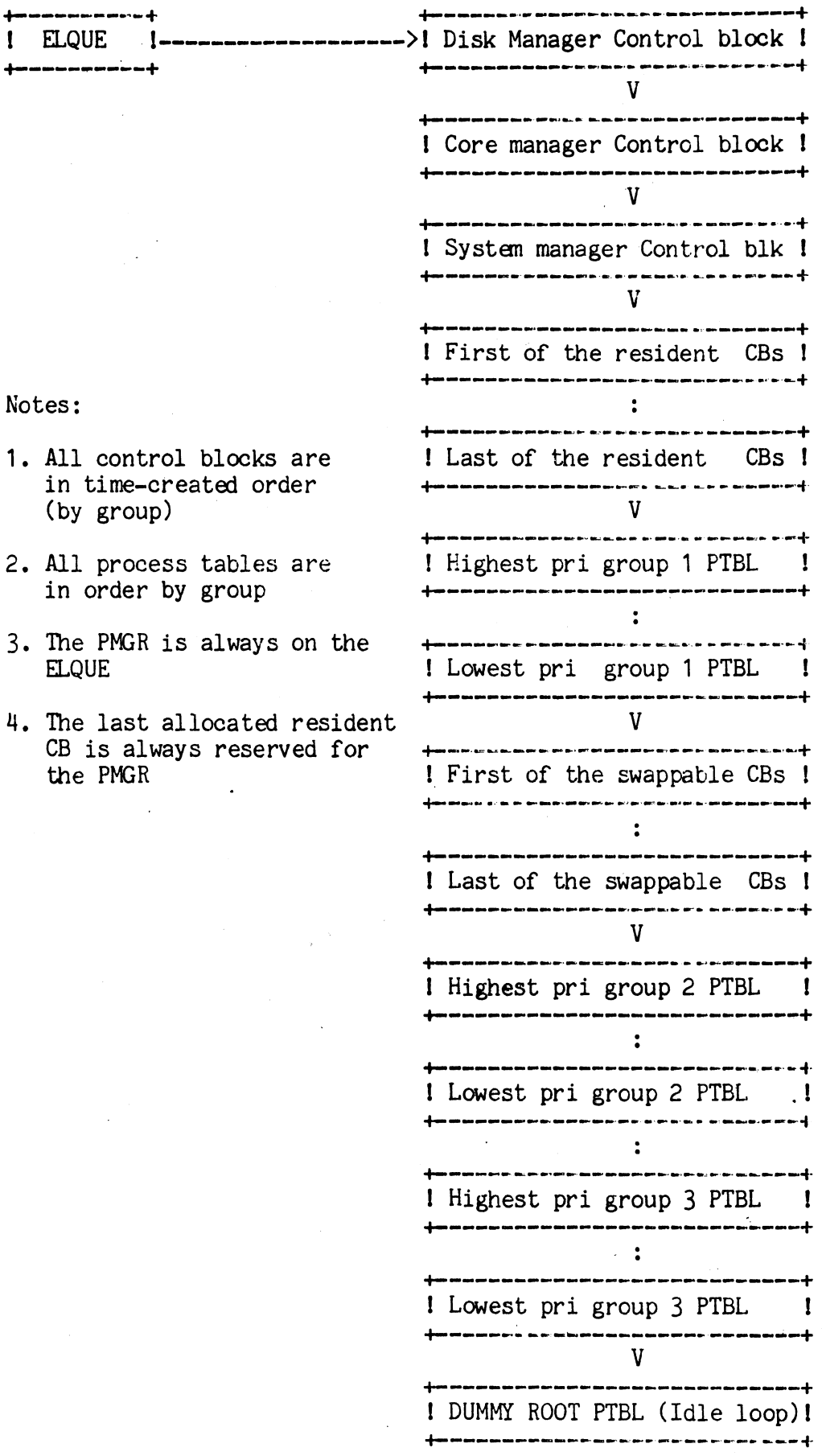
* The group 1 process tables are next. These are in order based on a process' priority. These include the PMGR process table which is permanently on the ELQUE

* Next comes the swappable Control Blocks, again FIFO by time

* The group 2 process tables are next ordered by PNQF

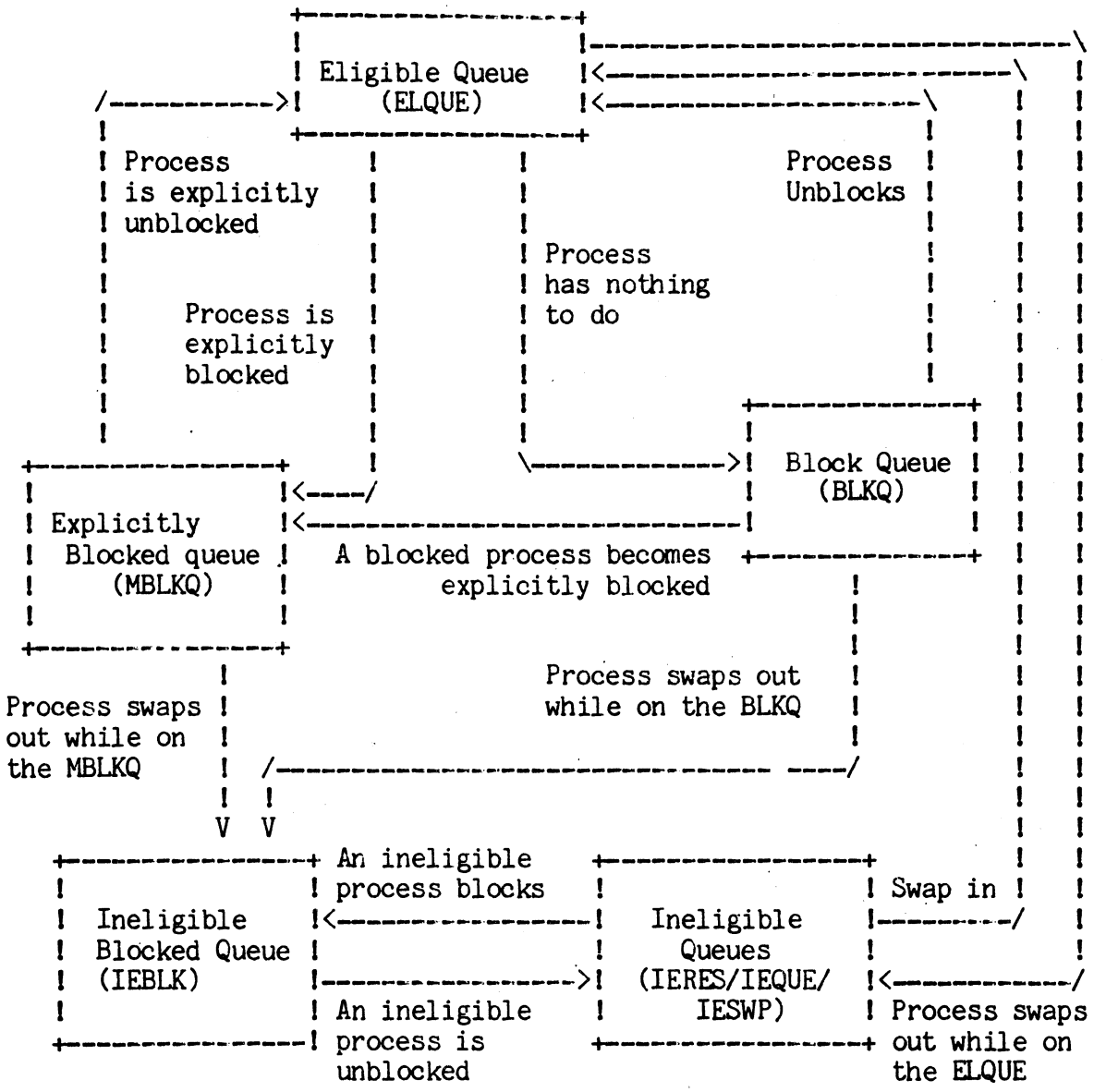
* The group 3 process tables are next ordered by priority

* Last on queue is a dummy process table, the root process table. This never requires CPU time, but is used to mark the end of the elque. The root process table has a PID of 0, and is the father of the PMGR and OP:CLI processes.



Notes:

1. All control blocks are in time-created order (by group)
2. All process tables are in order by group
3. The PMGR is always on the ELQUE
4. The last allocated resident CB is always reserved for the PMGR



Notes: A process has nothing to do if:

a. There are no task ready to run ... and ...
b. There are no outstanding system calls (?WDELAY/?IREC are not outstanding system calls) ... and ...
c. There are no TCBs enqueued to the PEXTN's system call chain

A process is explicitly blocked if:

a. A ?BLKPR is targeted at that process ... or ...
b. The process executes a ?PROC with the block option (wait until son termination)

Other queues:

IEQUE: Queue of ineligible processes involved in move bytes

IESWP: Queue of ineligible group 2 processes

IERES: Queue of ineligible group 1 processes

IEBLK: Queue of ineligible blocked processes

BLKQ: Queue of non-explicitly blocked processes

MBLKQ: Queue of explicitly blocked processes

One minor queue that involved in scheduling:

Core Manager queue: Queue of processes waiting for swap in or out

CPU time contention

In general, AOS/VS will allocate CPU time in the following overall priority structure.

Interrupt driven control functions that process events, these include:

Interrupts
Time Slice Completions (PIT interrupts)

group 1 control blocks, these include:

The Disk Manager -- permanently the highest priority.

The Core Manager -- permanently the second highest priority resident system daemons, and resident user system calls in a FIFO order.

Group 1 processes, ordered by PNQF

Group 1: PNQF = assigned priority

Swappable Control blocks, these include:

Swappable system daemons, and swappable user system calls in a FIFO order.

Group 2 processes, ordered by PNQF

Group 2: PNQF = dervied from priority and interactiveness

Group 3 processes, ordered by PNQF

Group 3: PNQF = assigned priority

Group1 processes receive control for a fixed time slice of 2.048 seconds. When this slice expires, the process is re-linked to the end of its priority group. If there is only one process at this priority level, then the same process is run again for another 2.048 seconds. Every 32 milliseconds (a sub-slice), the process's tasks are rescheduled. This is done to allow a round robin scheduling of multiple tasks at the same priority level.

Group 2 processes receive time slice based on past behavior and assigned priority.

T = time-slice = (32 millisec) * (2 ** S)

where: 1 <= S <= 6

The initial S is 1

The initial S will then be modified based upon the use of the allocated time slice by the process.

If the process blocks before the full slice expires, S will be given a new value based on the number of subslices used.

If process is still running when the time slice expires, S will be incremented by 1 the next time the process is scheduled. However, if the swappable process' priority is > 1, and the current S is 6, no change will be made to S. If the swappable process' priority is 1, S may reach an effective value of 7 (S will still = 6). In this case, a compute bound, or non-interactive, group 2 process of priority 1 can attain a time slice of 4.096 seconds. It will, therefore, receive twice as much CPU time as an equally compute bound group 2 process with a lower priority.

Actual time slicing is done on 32 ms. intervals. Hence, 2 ** S yields the number of sub-slices.

The scheduler maintains a count of the subslices for each process and the remainder of any incomplete sub-slice in the process's process table. This is done in case the process is pended due to:

1. Processing of an interrupt.
2. The scheduling of a system control block, or a higher priority process after an interrupt.

Control will return to the interrupted group 2 process when group 2 processes are again allowed CPU time.

Priority enque factor derivation for group 2 processes

- The basic equation:

PNQF = (slice-exponent) + 1B0 + (7*priority)

Notes:

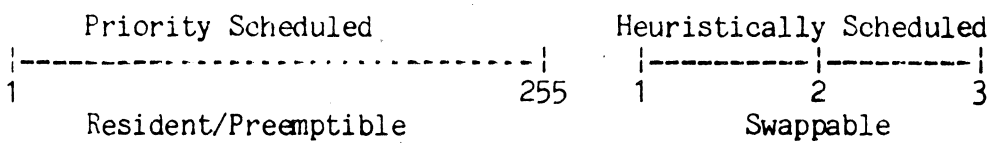
- PNQF = 100001 if the process is terminating.
- 1B0 is set to insure that all group 2 processes have a lower priority than group 1 processes.
- The 'slice-exponent' is the S explained earlier.
- PNQF values are read as unsigned values. The lowest PNQF values have the highest priority for CPU time.
- If the PNQFs for two processes are equal, they are managed on round-robin basis to insure all processes can get CPU time.
- The PNQF is updated whenever a process blocks, or a time slice expires.

The AOS/VS process scheduler

Prior to Revision 4.00 AOS/VS there were three process types with two independent types of fixed scheduling characteristics. Resident and preemptible processes are always of higher priority than swappable processes and may be assigned an external priority between 1 and 255. Swappable processes are always of a lower priority than resident and preemptible processes and may be assigned an external priority between 1 and 3.

There were two scheduling groups:

1) Resident/Preemptible Process priority range 1 to 255
2) Swappable Process priority range 1 to 3



One of the inherent problems with this strategy is that a user is always forced to have Resident/Preemptible processes of a higher priority than Swappable processes. This is a problem if a user wants to run a Resident process at a lower priority than a Swappable process. Another deficiency is the inability to define a process at a lower priority than that of a Swappable processes. With all Swappable processes being Heuristically scheduled regardless of the externally assigned priority, all Swappable processes compete with each other heuristically.

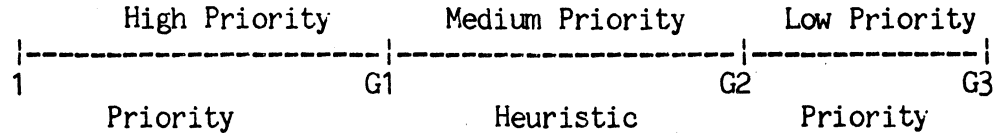
The first step taken is to define a new priority structure. This structure defines three separate scheduling groups which may be chosen regardless of process type. A result of this definition is to make the priority independent of the process memory requirements type. The other

factor which in AOS/VS used to be insignificant now comes back into use. That factor is the BIAS. The priority structure is now centered around three groups. The groups are:

Group 1:
High Priority
Priority Scheduled only
Priority Range 1 to G1

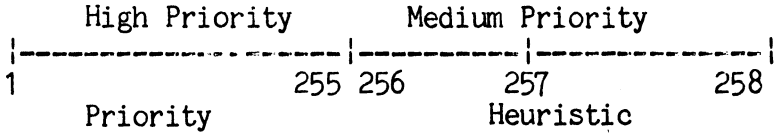
Group 2:
Medium Priority
Heuristically Scheduled only
Priority Range G1 + 1 to G2

Group 3:
Low Priority
Priority Scheduled only
Priority Range G2 + 1 to G3



Each of the Priority Group limits, (G1, G2, G3), are user selectable at VSGEN time. The limit for G3 is 511. In order for this new scheme tc be compatible with prior revisions of AOS/VS priority scheme the following values must be used:

G1 = 255
G2 >= G1 + 3
G3 = G2



If this structure is compared with the one drawn to show the current AOS/VS scheduling groups, some similarities can be seen. The only differences are, that the Heuristic scheduling priorities are 256, 257, and 258, instead of 1, 2, and 3, as previously defined.

Examples of how this scheme works:

1. When a Swappable process is created at Priority 1, it will be mapped by the system onto priority 256 in the Heuristic Range in order to meet the compatibility requirement. This means that a Swappable process cannot be created at priorities 1, 2, and 3 within the high priority group where there is no Heuristic algorithm, but Swappable processes can be created that are not Heuristically scheduled within the priority range 4 through 255.

2. Resident and Preemptible processes at priorities between 1 and 255 act as before.

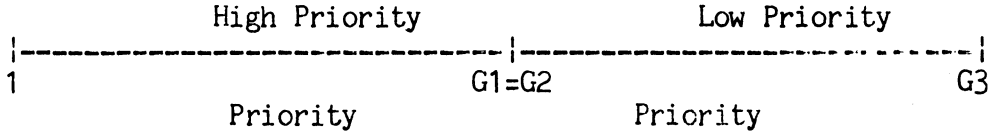
3. Resident and Preemptible processes created at priorities 256, 257, and 258, will be heuristically scheduled.

It should be apparent that this scheme is fully compatible with the old scheduler. A user is not required to change the priority structure of an installation to fit the new scheduler in AOS/VS revision 4.00. It also allows for more flexibility in that the user is no longer forced into the scheduling characteristics based on process memory type. This scheme allows the user to view the scheduling as a two-fold scheduler with High Priority Scheduling and Lower Priority Heuristic Scheduling.

A Simple Scheduling Scheme Without Heuristic Scheduling

G1 = G2

By specifying these Group Limits as equal, a simple Round Robin scheduler without heuristics is generated and no assumptions about process type versus scheduling characteristics exists.



Any process type may be selected at any priority and scheduling is strictly Round Robin.

An example of a system which could use a Round Robin Scheduler is a polling system.

The framework chosen for the new scheduler design provides a simple and fast scheduler with the underlying tools to build a more complicated structure. The compatible Scheduler algorithms are:

Decision Making Algorithms

Quantum Oriented Scheduling
This provides processor-sharing scheduling characteristics by defining Quantum expiration as a preemption point.

Preemptive Scheduling
Any process of higher priority than the currently executing process which becomes ready to run via external event completion will become the next process to be scheduled.

There are two independent Quantum times which are used for Preemption purposes.

Sub-Slice
This is the smallest Quantum which may be used for preemption purposes.

Time-Slice
This is the largest Quantum made up of some integral number of Sub-Slices.

By defining two different Quantums it is possible to use two Decision Making algorithms whose Decision Epocs occur at different frequencies. This allows for short (Subslice), and long (Timeslice) term Decision making algorithms.

Arbitration Rule Definition:

Any arbitration which must be done to resolve conflicts among competing processes is done via Round Robin Scheduling.

Round Robin Scheduling

When a process completes its time Quantum, it is considered as lower in priority than all processes of equivalent priority.(i.e. A process is shuffled to the end of its priority group at the end of a time Quantum.)

The current scheduler (pre rev 3.00) algorithm schedules processes via linear ordering by Priority eNQue Factor, selecting the highest priority process to run at any given time. The lower the PNQF, the higher the process' priority, and the sooner it was scheduled for execution. AOS/VS orders all processes by PNQF, and generates PNQF as given below:

Process Type	PNQF
Resident/Preemtible Control Blocks	0
Resident/Preemtible Processes	Process Priority (1-255)
Swappable Control Blocks	32768
Swappable Processes	32768 + (7 * Process priority)+E Where E is the Heuristic 0 < E < 7

The new scheduler for revision 4.00 maintains this type of linear ordering by using the following PNQF definitions.

Group Type	PNQF
Resident Control Blocks	0
Group 1 Processes	Process Priority
Swappable Control Blocks	G1 + 1
Group 2 Processes	G1+1 + (7*(Process Priority - G1))+E
Group 3 Control Blocks	7 * (G2 - G1 + 1) + G2
Group 3 Processes	7*(G2 - G1 + 1) + Process Priority

Given the above changes to the scheduler, below is an example of how the new scheduler can be made compatible with the old scheduler in terms of PNQF ordering of processes for execution.

It was stated earlier that in order to define a compatible scheduler that G1 = 255, G2 = 258, and G3 >= G2. By defining G1 = 255, G2 = 258, and G3 = 270 a compatible scheduler may be generated. By substituting into the PNQF definitions, one obtains the following PNQF equations.

Non-heuristic Group

1 < Priority <= 255

Group 1 Control Blocks	0
Group 1 Processes	Process Priority

Heuristic Group

255 < Priority <= 258

Group 2 Control BLocks	256
Group 2 Processes	256 + (7*(Process Priority-255)) + E

Non-heuristic Group

258 < Priority <= 270

Group 3 Control Blocks	28 + 258
Group 3 Processes	28 + Process Priority

This generates non-overlapping linearly ordered PNQF's given below.

Group 1

User selectable Priority Range 1,2, ..., 255

Control Blocks	0
Processes	1,2, ..., 255

Group 2

User selectable Priority Range 256, 257, 258

Control Blocks	256
Processes	256 + 7 + E
	256 + 14 + E
	256 + 21 + E

Group 3

User selectable Priority Range 259, 260, ..., 270

Control Blocks	258 + 28
Processes	Process Priority + 28

Another added feature of revision 4.00 scheduler is a secondary scheduler which can be used in CPU contention envircnments. This scheduler requires that the system be patched in five (5) locations to enable the secondary scheduler. In the real time clock handler is a routine to check to see if the CPU is under heavy contention by checking if an idle loop counter (WTSCIDL) has exceeded a cutoff limit (WTSCIDLL). If it has exceeded the cutoff limit there is a flag (WTSCGO) set so that the scheduler knows that weighted scheduling is enabled. This same routine sets a bit in the G2 and G3 process tables to show which one has run or not. After picking someone to run in the scheduler and that someone is a G2 or G3 then check if it has run before in this interval of time. If it has run before then it is passed over and the next who is both ready to run and has not run this interval is selected to run.

The weighted scheduler is turned off by either of two events:

The duty cycle completes (it can be specified that processes will run only N seconds out of an interval)

or

The number of times to cycle through the ELQUE has been used up.

In either case the flag (WTSCGO) which was set to turn on weighted scheduling gets turned off.

On the ELQUE there are three types of control blocks. They are Disk manager control block, Core manager control Block, and System manager control block. These control blocks have to have the highest priority on the queue to enable them to run and run quickly and frequently.

Disk manager

The disk manager runs as the highest priority control block on the ELQUE. It replaces the branch out of the scheduler which was in prior revisions which went to the same point. The change was made to speed up the scheduling process. The disk manager runs all IOCBs when they are ready to run. When the N_T_run.w is set the scheduler branches to RUNLC in DSKIO to shchedule IOCBs. All active IOCBs are run. As long as there are ready IOCBs they are run. When there are no more ready IOCBs the disk manager control block is changed by stting the 'not ready to ru bit and resetting the running bit. The disk manager is readied by a call to the routine DWAKE.

Core manager

The core manager runs as the second highest priority control block on ELQUE. It remains dormant until a code path calls the routine CWAKE which sets the ready to run bit in the status word for the CB in addition to the words or bits needed to indicate which action the core manager should process when it gets control of the CPU.

Special requests to core manager (flag word is SMFLG)

a. 1b0 -- Unpend resident processes waiting for memory release. This is done by loading the appropriate key and calling UNPEND.

b. 1b3 -- Scan BLKQ looking for anyone waiting to unblock. (call BSCAN in COREM2)

The SYSTEM MANAGER

The system manager code is found in SYSMGR. The system manager is currently used for five purposes. The first is to report unit errors, the second is to report over- subscribed memory, the third is to enable look-ahead faulting, the fourth is to enable look-ahead flushing, and fifth is to handle SCP error reporting. The unit errors are detected by the controllers, which set up error status words in the appropriate UDB (unit device block). The error routine then calls SWAKE, which will cause the system manager to wake up the next time a scan is made of the eligible queue (much as CWAKE does for the core manager). The oversubscribed memory condition (no memory available, no preemption possible) is detected by the preemption code, which then call SWAKE.

The requests to the system manager are indicated in CMFLG.(The core manager equivalent is SMFLG)

The AOS/VS task scheduler

The AOS/VS scheduler checks to see if the PTBL just readied for running is a control block, a PTBL or a TCB. If it is a TCB then it causes the user routines to be executed. If it is a PTBL it examines the active TCB chain for a ready to run TCB. The task scheduler has the ability to check to see if a more significant event has occured to force a rescheduling to the other event. If there are tasks to be run then ready the tasks and run them. If there is no other task other than the current one then run it.

Event Synchronization

It is unusual for AOS/VS to process for an extended period of time. Usually, many paths require short periods of CPU time. When a path gives up control, it is generally because it's waiting for an event.

Typical events are:

1. Waiting for a disk block to be read into AOS/VS buffer (e.g. opening a file requires the reading of a directory entry.
2. Waiting to access a database being used by another active path.
3. Waiting for an AOS/VS global resource.

The general procedure for pending/unpending is as follows:

1. The path calls the subroutine 'PEND', passing to 'PEND' the key it wishes to wait on.
2. The path becomes quiescent (i.e. it is not given any CPU time).
3. Another path calls the subroutine UNPEND, passing to 'UNPEND' the key that the first path is waiting on.
4. The waiting path now becomes ready and resumes execution.

All paths waiting on given key are readied when 'UNPEND' is called.

PEND keys

SKTRM = 1	Wait for son proc term
SKTRG = 2	Wait for target call completion
SKOOM = 4	AOS/VS needs memory
SKSWP = 5	Wait for swap of proc to complete
SKSIO = 6	Shared read wait
SKBUF = 7	Base level AOS/VS needs buffer

SKDED = 8.	Target call waiting for special unpend after ELQUE scan
SKNWU = -1	General wait (no wakeup key)
= database	An AOS/VS code path can pend waiting for a specific database. In this case, the key is the database address

AOS/VS uses hierarchical event locking. A majority of the paths only lock a single database. If two databases must be locked, then a path requiring the locking of database 'A' and then database 'B' will require that any path that uses these two databases must lock them in the same sequence.

The interrupt world

When an I/O device completes its operation and is ready to receive/send more data, it requests an interrupt. As soon as the CPU is at an interruptable point in its processing, and has finished servicing data channel requests, it takes care of the interrupt.

Upon servicing an interrupt, the ECLIPSE MV CPU does the following (with interrupts left off):

- if the ATU is enabled (in VS it will be), Fetch the pointer to the interrupt handler from ring 0, location 1
- If the ring at the time of the interrupt is not zero, then store the current stack registers in the current ring's page zero locations, load ring 0's stack information into the hardware stack registers, and cross to ring 0
- resolve any indirection in the pointer to the interrupt handler address
- process the first word of the handler (which in AOS/VS is a XVCT instruction, which, in addition to many other things, will reenable interrupts, and vector to the appropriate interrupt handler

After servicing the device, the interrupt service routine will jump to the routine DISMIS, which will return program execution to the point at which we were interrupted. Note that one of the last last things that the XVCT instruction does is to reenable interrupts, thus allowing other device to interrupt us.

If interrupts are disabled throughout the interrupt service routine, the CPU can no longer be interrupted until this device servicing is finished, and all other devices requesting interrupts must wait. This might lead to losing data on fast unbuffered devices. Therefore more sophisticated hardware instructions are available to implement a system of interrupt priorities which will permit some devices to interrupt others.

Every I/O device is assigned an interrupt mask bit by the hardware, and the interrupt service routines can control interrupt priorities by setting interrupt masks : any device which should not interrupt the device being serviced is masked out (prevented from requesting an interrupt) if its mask bit is set. The mask bits corresponding to devices which can interrupt are zeroed. By changing the priority mask, an interrupt service routine can mask out those devices whose interrupts are undesirable, without disabling interrupts
for the duration of the service.

AOS/VS handling of interrupts

Location 1 of a machine running AOS/VS will normally point to INTS, the interrupt dispatching routine. When the ECLIPSE MV is

interrupted, it examines the effective address pointed to by location 1. The effective address in AOS/VS resolves to INTS, which contains a XVCT instruction. When the MV encounters the XVCT instruction, the following occurs:

```
Fetch the level count from location 0, segment 0

If the count is 0, then

    Increment the level count
    Save location 14 (stack fault address) and the stack info
      internally
    Load location 14 and the stack registers with the vector info
      (from page 0, loc 4-7)
    Push saved data onto the new stack
    Push wide return block onto the new stack

If the count is not 0, then

    Increment the level count
    Push wide return block onto current (seg 0) stack

Calculate the effective address of the XVCT instruction

Index into the vector table (effective address from XVCT address)
  by device code.  Value will point to DCT for the device

Push current mask (VCT table address-2) onto stack

OR current mask with contents of 2,3 of DCT into VCT addresss-2

Do MSKO with the ORed mask

Load ZEX device code into AC1

Load PC with first two words of DCT

Load PSR with word 4 from DCT

If a stack overflow has occured:

    Transfer control to the stack fault handler (loc 14)
    Fetch and execute first instruction of fault handler

If no stack overflow has occured:

    Fetch and execute instruction pointed to by PC

 Enable interrupts
```

BTBL (Vector table)

BTBL is the vector table that the XVCT instruction will dispatch through. The table is a series of double word pointers to interrupt service routines. The table is indexed by device code. The table is initialized so that all pointers point to the routine IUD, which is the undefined interrupt handler. During SINIT, the table pointers are filled

in for each sysgened device so that they point to the appropriate DCT, which in turn contains the address of he interrupt service routine. When a user IDEFs a device, the pointer corresponding with the IDEF'ed device code is set to UDEX, which is the dispatch routine to the user interrupt processor.

DISMISS (Interrupt dismissal)

Interrupts in AOS/VS are dismissed by the routine DISMISS (in the module INTS). After the appropriate routine processes the interrupt, it does a XJMP to DISMISS (The exception is the routine UINTR [user interrupt dismiss] which jumps to a separate entry in DISMISS). DISMISS will return based as follows:

1. Decrement INTLV

2. Disable interrupts

3. If we are not at level 0, then perform a WPOPB to get back to previous level (note that UINTR jumps to DISMISS at this step)

4. If the ring field of the PC at the time of the interrupt is 0, and INCHK is not 1, enable interrupts and do a WRSTR to restore the pre-interrupt state. Increment INSYST.W, the number of interrupts that occured while the system (non-idle) was active.

5. If the ring field of the PC at the time of the interrupt is 0, and INCHK is 1 then the system was interrupted while in the idle loop so increment IDL.W, the counter of such occurances. Then check RESCH and UPQUE. If the are both 0, enable interrupts and do a WRSTR. If either RESCH or UPQUE is not 0, then clear INCHK and jump to the top of the scheduler

6. To reach this point. the processor was in the user (ring<>0) at the time of the interrupt, so increment INUSER.W. If both RESCH or UPQUE are 0, and we have not reached a sub-slice end, then start up the PIT, enable interrupts, and WRSTR

7. If the current subslice is done, add the slice value to the PEXTN offset that contains CPU time (PRUNH.W). If the CPU time exceeds the limit (limiting requested), format the user's process table so that the process will terminate. If the subslice end is also a full slice end, set the flag to indicate so. Store the time slice residue in the PEXTN, save the current task's information, and jump to the scheduler.

8. Finally, to reach this point. the processor was in the user

(ring<>0), this is not a subslice end, and either RESCH or UPQUE is non zero. Reset the running bit, store the sub-slice residue, save the current task's state, and jump to the top of the scheduler.

INTS logic

Simply execute a XVCT instruction on the location BTBL. This will result in the functions described above, with control eventually passing to the interrupt service routine appropriate for the device that interrupted.

IRTC logic

This is the real time clock interrupt handler. Every clock tick it performs the following :

- read and halt the PIT, and save the time slice in TSLSV if the interrupt occured at base level
- startup the RTC
- check to see if PFF can be turned off, if it can be turned off then turn off PFF and update counters.
- if a second has elapsed, update the time of day, and if necessary, the date
- if in state 3 or 4 check to see if switching states will help system performance (see PFF and PSTEAL in memory management)
- if a second has elapsed, check for a device timeout, and process any that occur. (Each device's timeout block will be decremented and if any go to 0, the system will dispatch through the timeout routine for that device)
- if there are any processes with histograms, the histograms are updated.
- the time remaining for any process on the delay chain is decremented; if it goes to zero and the process is eligible, the task making the delay call is unpended. If the process is not eligible, set the BPFDU bit to tell the SWAPIN code to unpend the first task with delay when the swapin is completed. If the process is blocked, set the appropriate flags to have the Core manager unblock it.

UDEX and UINTR logic

UDEX is dispatched to on any interrupt from a user device. If the interrupt occured at base level, UDEX will first halt and read the PIT. UDEX will then save the interrupted PTBL address on the stack, get the PTBL for the user that defined the device, and turn I/O allowed on and LEF mode off for the ring that contains the service routine. UDEX then loads the hardware SBRs (1-7) with the SBR words for that user (from the PEXTN).

Finally, a fake return is built on the stack to set up the user's AC, and a WPOPB is executed to go to the user.

The user returns from the user interrupt handler via a LCALL instruction to UINTR. UINTR will restore the SBRs to the state before the interrupt occured and jump to the common interrupt dismissal code (DISMISS).

IWKUP and UIWKUP logic

IWKUP readies a specified process. If the process is not pended on a page fault, the the not ready to run bit is reset. If the process is a resident process other than the PMGR, IWKUP will force a reschedule of the system by incrementing RESCH, and of the user's TCBs by setting the BPFRS (reschedule) bit in the process table. Before returning to the calling code path (IWKUP is called with a XPSHJ), enable interrupts.

UIWKUP is a special version of IWKUP called from UINTR. Certain assumptions are made because we know that UIWKUP is called from the interrupt world. UIWKUP will reset the not ready to run bit if the process is not pended during a fault, XNISZ RESCH to force a reschedule, set the BPFRS bit to force the processes tasks to also reschedule, and return (UIWKUP is also called via a WPSHJ). Note that UIWKUP does not enable interrupts before returning.

Power Failure Handling

AOS/VS now detects power failure if enabled at VSGEN time. The STKS module checks for power failure and branches to the power failure routine. The sytem does one of three things based on configuration. If full battery back-up then system will restart if user selected the option otherwise the system will be set up to run ESD on return of power. If partial battery back-up then ESD will be readied to run upon restoration of power. If no battery back up then ESD will be set up but system will die and FIXUP must be run to recover disks.

The machine state is saved (i.e. floating point.stack pointer, frame pointer, and device code). It tells the PMGR devices to save their state and halt. It tells any user devices to do their thing in event of power failure if programmed for power failure.

Upon restoration of power the power failure routine tries to resart the system by printing the power failure message on the console device. It tries to restart the sysgenned devices followed by the user defined devices. Upon successful completion of the restart another message appears indicating restart was completed.

PFL logic

This routine handles powerfails and spurious interrupts on device code 0. The routine performs a SKPDZ CPU, which will skip on a real powerfail. If not a real powerfail (spurious interrupt on device code 0), then ISZ a counter, and if the counter overflows, panic (4001), else jump to the DISMISS code. If a powerfail has occured, goto the powerfail routines (if defined) else panic (15001) when the system recovers.

SYSTEM CALL PROCESSING

General flow of a system call

Entry into AOS/VS is through the ring 0 gate array, which points to the entry point SYST in SCPRC. The user's system call is processed as a LCALL to the AGENT ring, which, after identifing the call as one that needs the system, will make an LCALL to the system ring. (the LCALL in the user space is found in the module SCALL which is bound into each user program. There is a version for 16 bit processes, and one for 32 bit processes. The code can be found in URT16.LB or URT32.LB. Calls from 16 bit processes are converted into 32 bit calls by the AGENT, which will recursively call itself. It is the 32 bit call that reaches the kernel.

Kernel system call processing

Entry to the AOS/VS kernel is through the ring 0 gate array. The standard system calls are referred to as TCB calls, (in that they run on TCBs). Only TCB calls go through SYST

Some system calls have specific attributes that are classified by tables found in SCPRC

1. Direct calls -- Processing of the system call will never pend, although the task making the call can pend. (No stack or CB is needed)

   ?RDB ?WRB ?SPAGE ?WDELAY ?MBTU ?MBFU ?RPAGE ?SGNL ?GPORT ?SIGNL ?WTSIG ?SIGWT ?DVSTT ?NSIGNL ?NWTSIG ?NSIGWT

2. Parallel calls -- A multitask process can have no other system calls active while this call is active.

   ?CTYPE ?GCHN ?GPROC ?MEMI ?SSHPT ?TABT ?RPAGE ?SPAGE ?SRDB ?IHIST ?WIRE

3. Expensive calls - Calls to the file system. This table is used to determine the CPU charge for a call

?CREATE ?GPROC ?DELETE ?RENAME ?GCHN ?ILKUP
?GOPEN ?GCLOSE ?SOPEN ?DIR ?INIT ?FSTAT
?RELEASE ?SLIST ?GLIST ?GNAME ?GACL ?SACL
?GNFN ?RDUDA ?WRUDA ?CRUDA ?SATR ?BRKFL
?SCLOSE ?CGNAM ?DACL ?GFNAME ?UPDAT ?ROPEN
?RCLOSE ?SOPFF ?RINGLD ?RNGPR ?WIRE ?GTRUNC
?ESFF

AOS/VS Modules involved in system call processing

SCALL -- bound into the user program

AGENT -- runs in ring 3 and preprocesses all system calls (converts 16 bit to 32 bit) (see chapter 8)

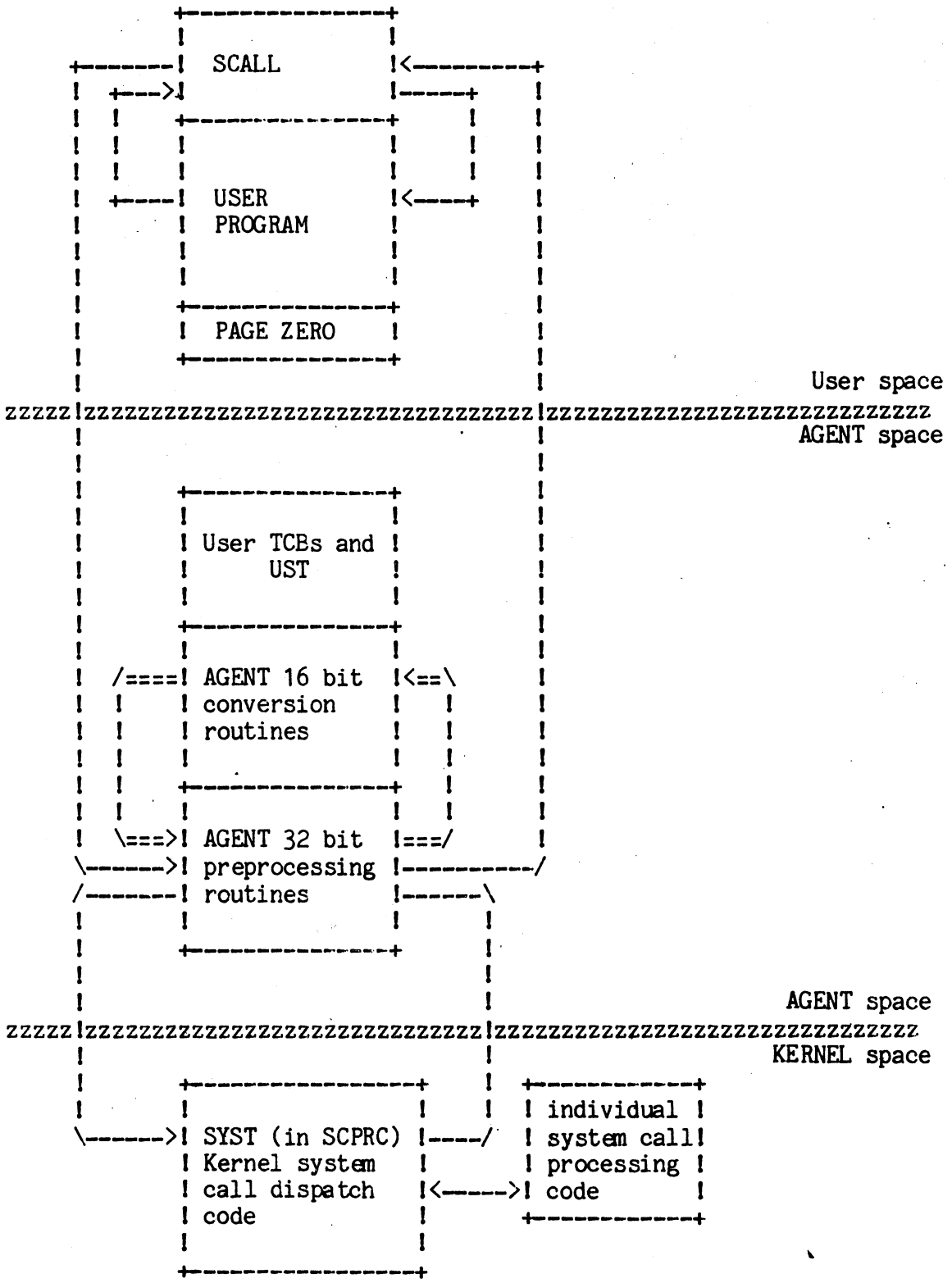
SCPRC -- system call processor for TCB requests

SCHED -- The AOS/VS scheduler. This picks up the enqueued requests from the PEXTNs, associates them with a CB and stack, and then jumps into the appropriate overlay for processing.

SCMOD -- This routine is involved in miscellaneous direct calls (those that cannot pend) and will not be included in this chapter.

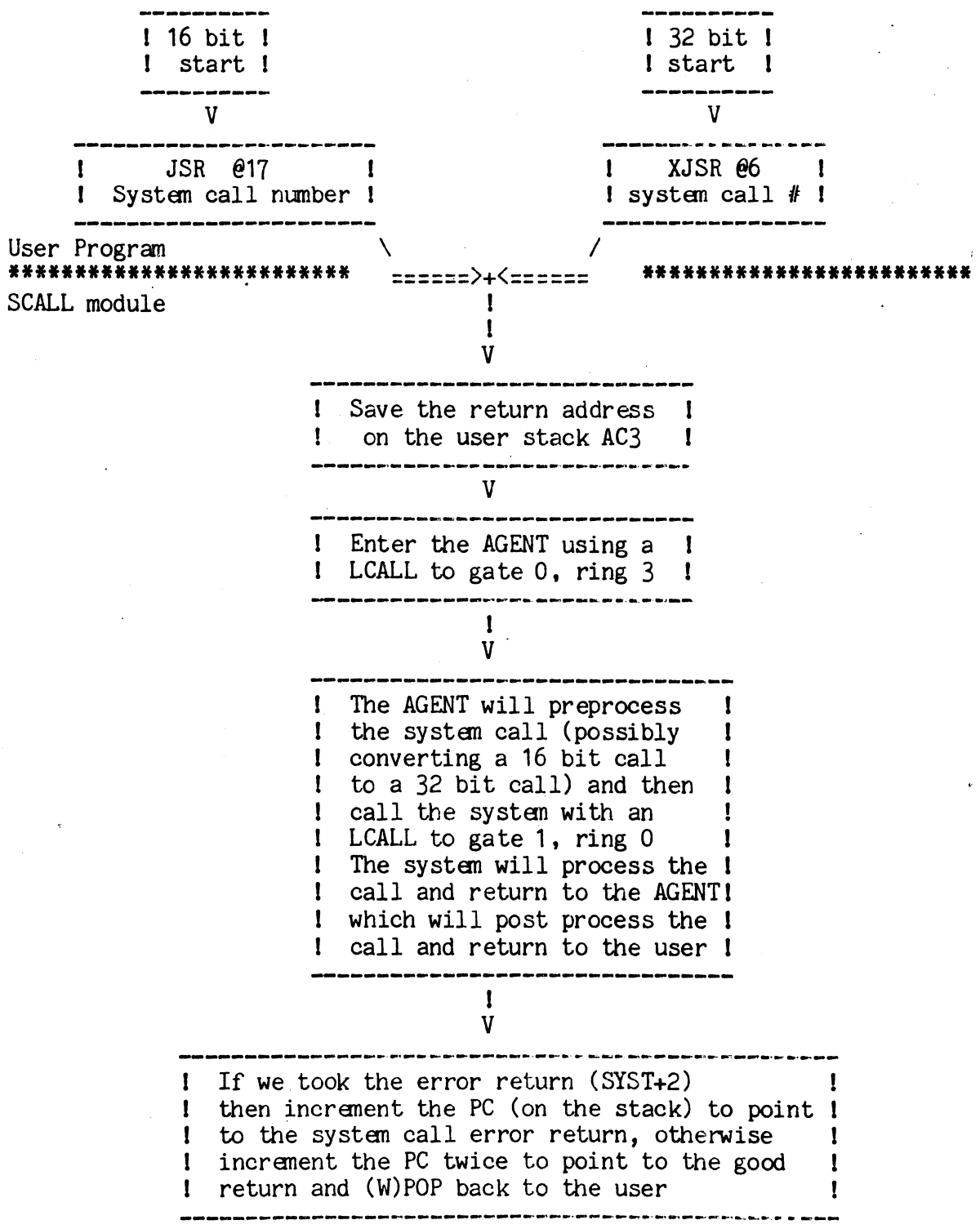
The diagrams provided on the following pages graphically represent the overall flow of a system call in AOS/VS

SYSTEM CALL PROCESSING INTERFACE



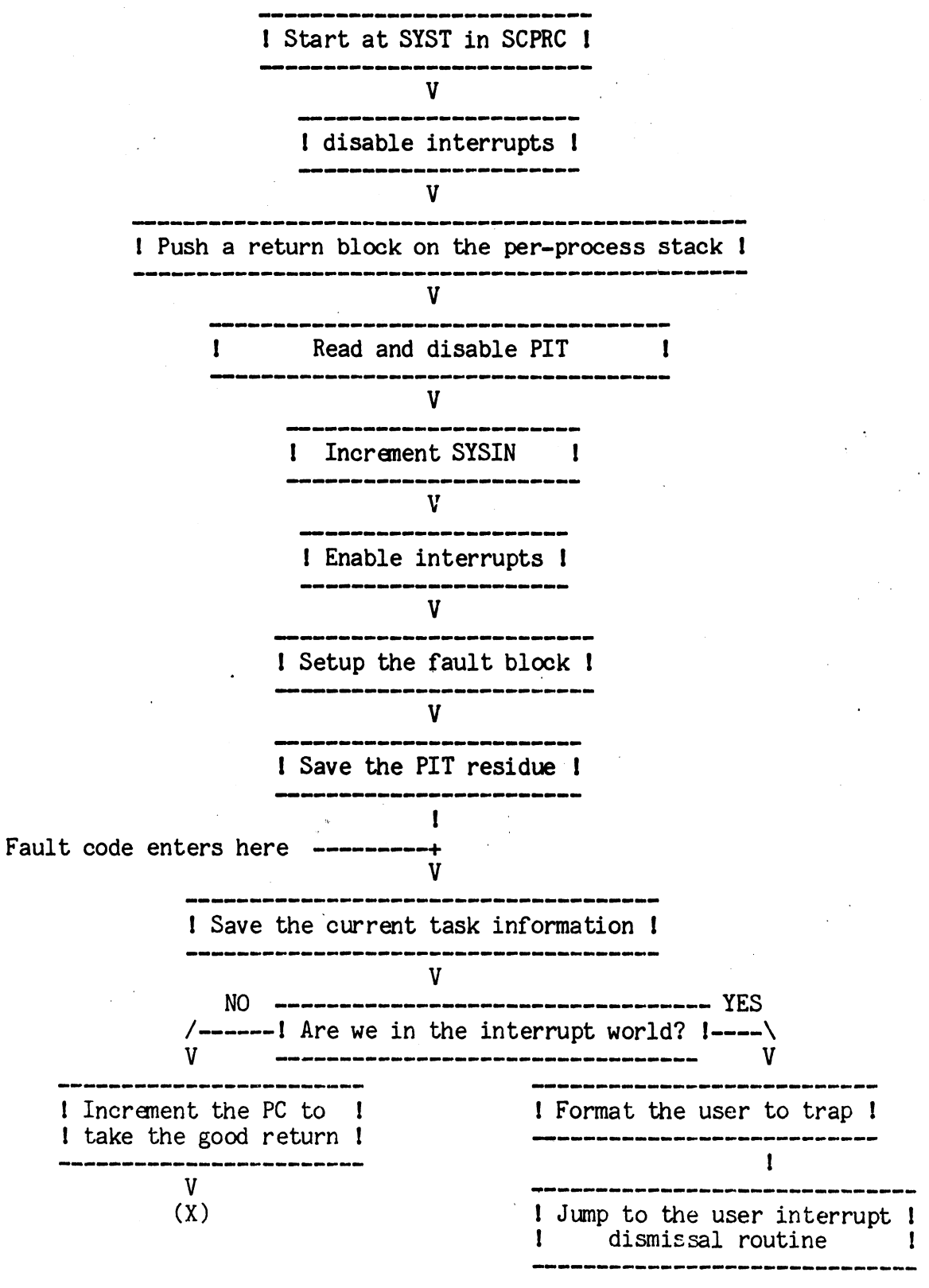
Module: SCALL

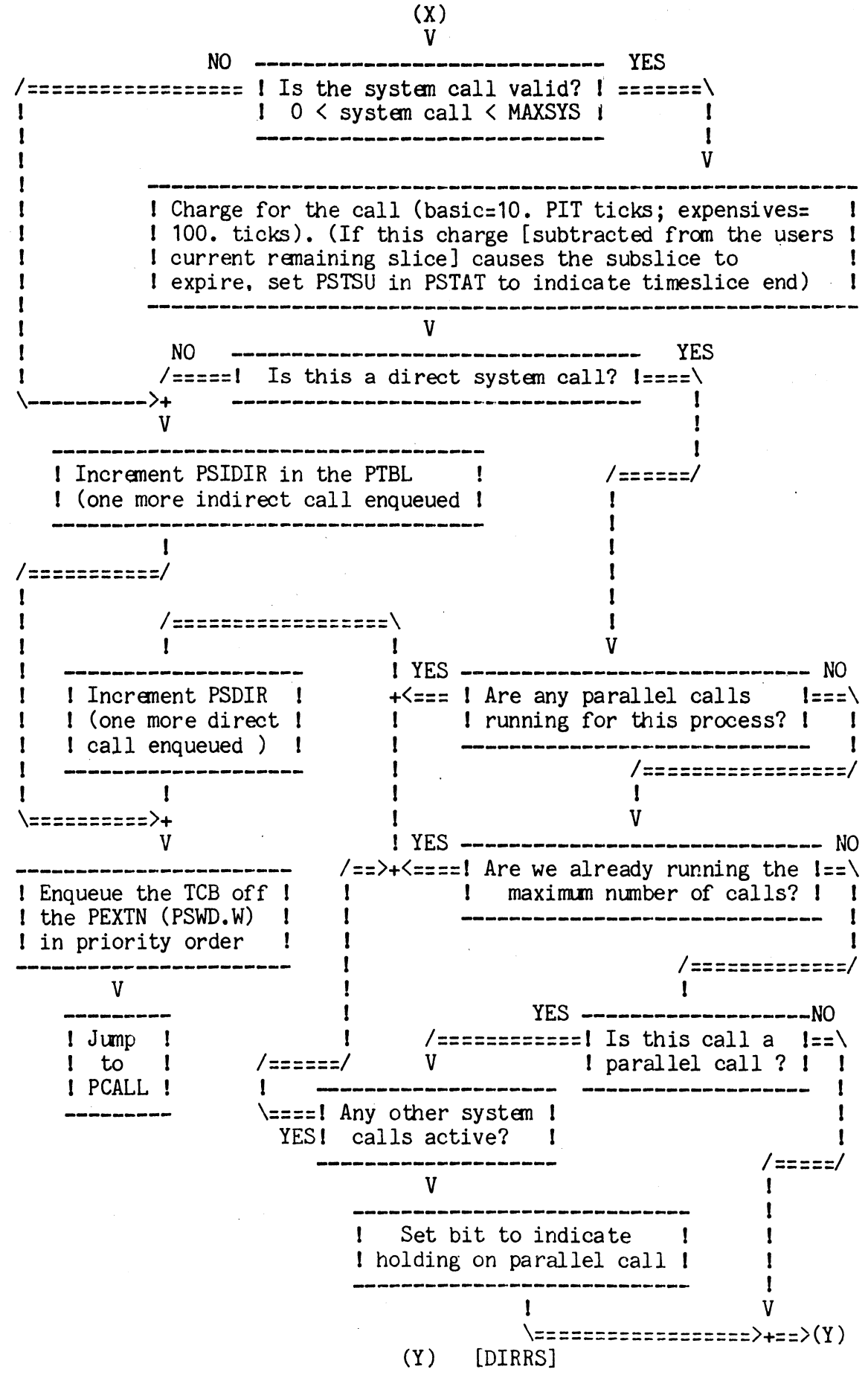
This module is automatically added into a user program at LINK time. The primary purpose of SCALL is to get the user into AOS/VS AGENT code. There are two different SCALL modules, one for 16 bit processes, and one for 32 bit. Though some of the instructions are different the general idea is the same.

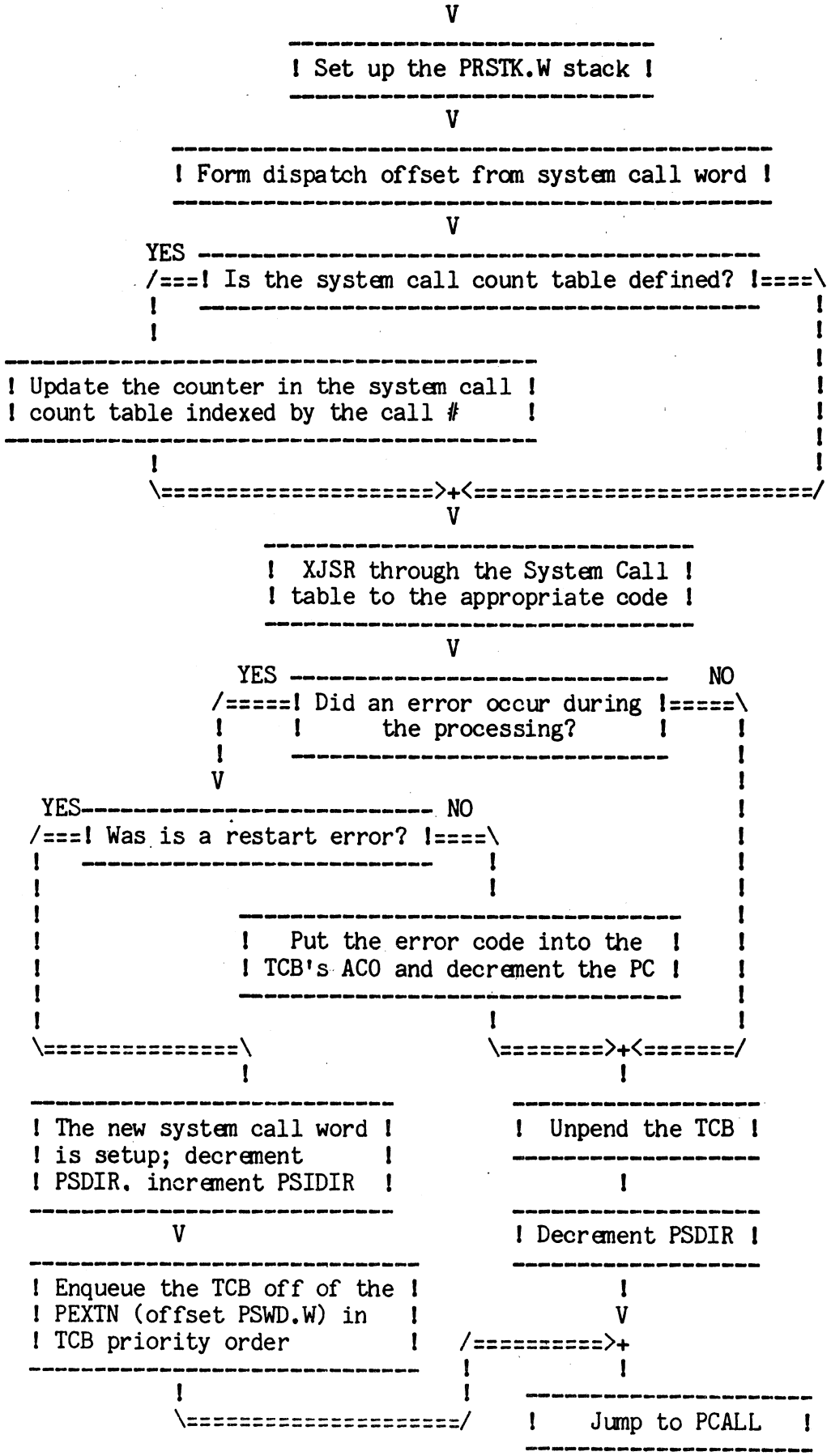


Module: SCPRC

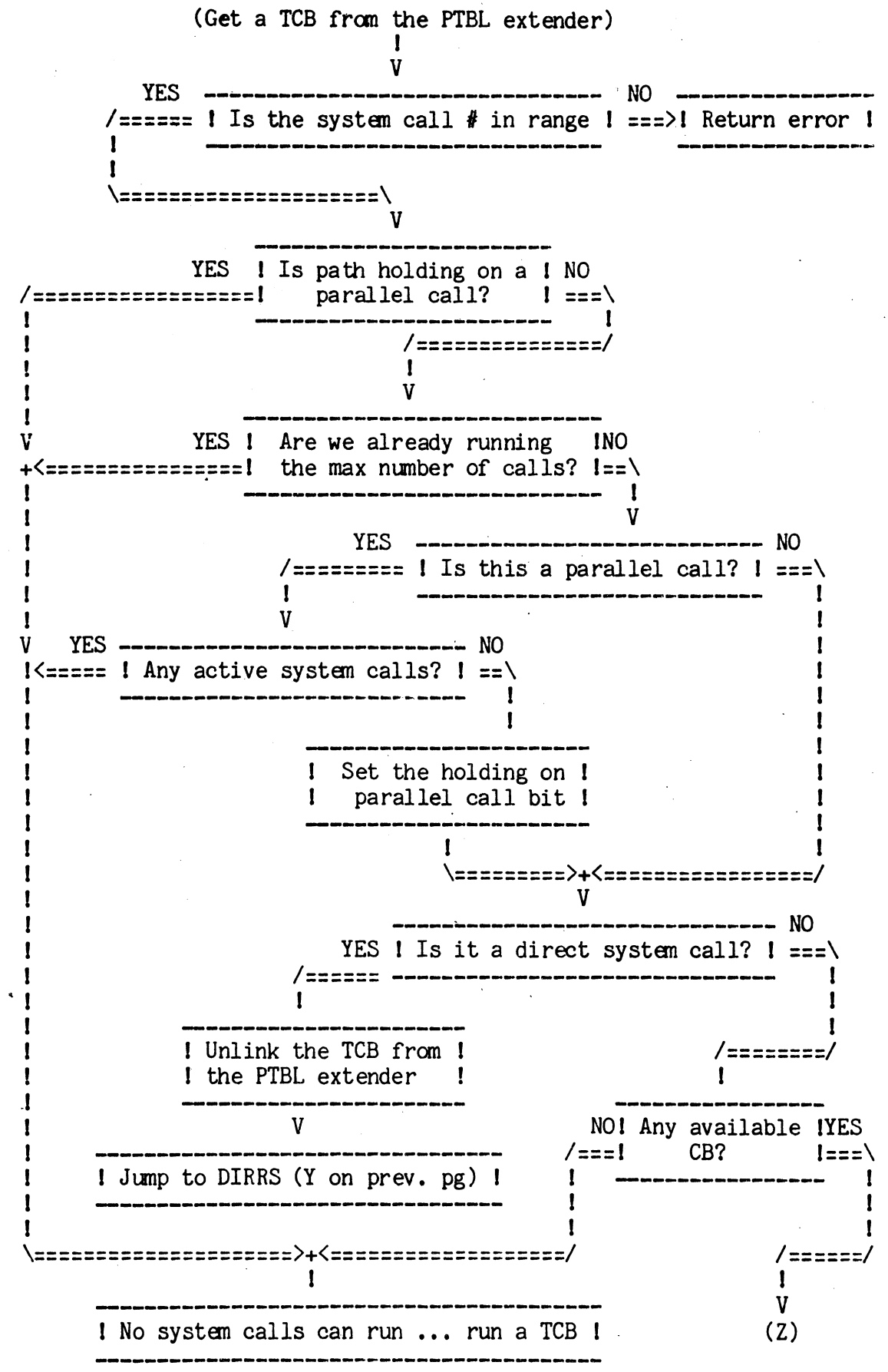
SCPRC is part of the kernel.

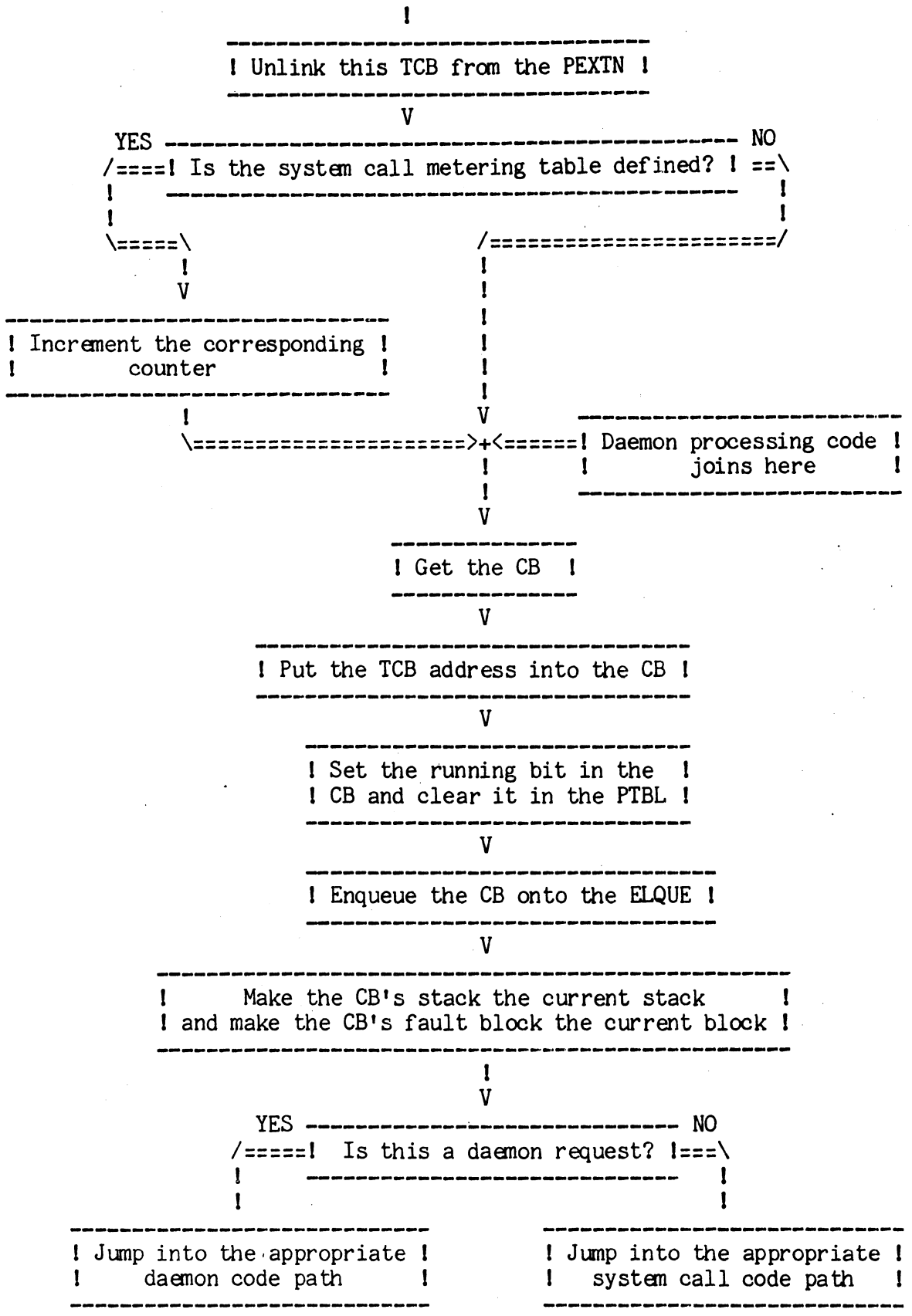



(X)
NO
Is the system call valid?
0 < system call < MAXSYS
YES
Charge for the call (basic=10. PIT ticks; expensives= 100. ticks). (If this charge [subtracted from the users current remaining slice] causes the subslice to expire, set PSTSU in PSTAT to indicate timeslice end)
NO
Is this a direct system call?
YES
Increment PSIDIR in the PTBL (one more indirect call enqueued
Increment PSDIR (one more direct call enqueued)
YES
Are any parallel calls running for this process?
NO
YES
Are we already running the maximum number of calls?
NO
Enqueue the TCB off the PEXTN (PSWD.W) in priority order
Jump to PCALL
YES
Is this call a parallel call ?
NO
Any other system calls active?
YES
Set bit to indicate holding on parallel call
(Y)
(Y) [DIRRS]



Module: SCHED


(Get a TCB from the PTBL extender)
YES
Is the system call # in range
NO
Return error
YES
Is path holding on a parallel call?
NO
YES
Are we already running the max number of calls?
NO
YES
Is this a parallel call?
NO
YES
Any active system calls?
NO
Set the holding on parallel call bit
YES
Is it a direct system call?
NO
Unlink the TCB from the PTBL extender
NO
Any available CB?
YES
Jump to DIRRS (Y on prev. pg)
No system calls can run ... run a TCB
(Z)
(Z)


Unlink this TCB from the PEXTN
Is the system call metering table defined?
YES
NO
Increment the corresponding counter
Daemon processing code joins here
Get the CB
Put the TCB address into the CB
Set the running bit in the CB and clear it in the PTBL
Enqueue the CB onto the ELQUE
Make the CB's stack the current stack and make the CB's fault block the current block
Is this a daemon request?
YES
NO
Jump into the appropriate daemon code path
Jump into the appropriate system call code path

System call dispatch locations (MCCT.W)

```
MCCT.W: CREATE.P                ; 0-  CREATE
        DELETE.P                ; 1-  DELETE
        RENAME.P                ; 2-  RENAME
        MEM.P                   ; 3-  AVAILABLE MEMORY
        GCHAIN.P                ; 4- CHAIN
        PRSTAT.P                ; 5- PROCESS STATUS
        DPMGR.P                 ; 6- DEFINE PERPH PROC
      ; PAIRED CALL
        RRDB                    ; 7- PRIMARY READ BLOCK
        RDB.P                   ; 10- SECONDARY READ BLOCK
      ; PAIRED CALL
        RWRB                    ; 11- PRIMARY WRITE BLOCK
        WRB.P                   ; 12- SECONDARY WRITE BLOCK
        PROC.P                  ; 13- CREATE A PROC
        MEMI.P                  ; 14- MEM INC
        -1                      ; 15- QUEUE A TIME REQ (AOS-16)
        INTAD.P                 ; 16- DEFINE INT PROC ADDR
        -1                      ; 17- MOVE BYTES TO GHOST
        -1                      ; 20- MOVE BYTS FROM GHOST
      ; PAIRED CALL
        RMBTU                   ; 21- PRIMARY CALL TO MOVE BYTES
        MBTU.P                  ; 22- SECONDARY WITH STACK
      ; PAIRED CALL
        RMBFU                   ; 23- PRIMARY CALL TO MOVE FROM
        MBFU.P                  ; 24- SECONDARY WITH STACK
        ISEND.P                 ; 25- IPC SEND
        IREC.P                  ; 26- IPC RECEIVE
        ILKUP.P                 ; 27- IPC LOOKUP
        GRUNT.P                 ; 30- GET RUNTIME STATS
        TABT.P                  ; 31- ABORT CALL
        -1                      ; 32- TRANSLATE A TO A MAPPED ADDR
        CTYPE.P                 ; 33- CHANGE PROC TYPE
PAIRED CALL
        RBLKIO                  ; 34- PRIMARY BLKIO
        BLKIO.P                 ; 35- SECONDARY BLKIO
        GTIME.P                 ; 36- GET TOD
        STIME.P                 ; 37- SET TOD
        SDAY.P                  ; 40- SET DAY
        GDAY.P                  ; 41- GET DAY
        IDEF.P                  ; 42- DEV INTERRUPT DEFINE
        IRMV.P                  ; 43- INTERRUPT REMOVE
        SSHPT.P                 ; 44- SET SHARED PARTITION
        RPAGE                   ; 45- REL SHARED SLOT
        ODIS.P                  ; 46- DISABLE PROG CONT A
        OEBL.P                  ; 47- ENABLE CONT A
        DEBL.P                  ; 50- ENABLE MAPPED DEV
        DDIS.P                  ; 51- DISABLE MAPPED DEV
        STMAP.P                 ; 52- SET MAP FOR USER DCH
        SUPROC.P                ; 53- CHANGE SUPERPROCESS MODE
        TABT.P                  ; 54- TASK ABORT
        TRMPR.P                 ; 55- TERMINATE A PROC
        GOPEN.P                 ; 56- OPEN
        GCLOSE.P                ; 57- CLOSE A USER CHANNEL
```

```
SPAGE.P                 ; 60- SHARED PAGE READ
CXINFO.P                ; 61 -AGEN/KERNEL PAGE FAULT INTERFACE
CISND.P                 ; 62- SEND A KEYBOARD INTERRUPT TO SON
SOPEN.P                 ; 63- SHARED OPEN
-1                      ; 64- GET A PORT OWNER AGENT RESERVED
TPORT.P                 ; 65- TRANSLATE A PORT NUMBER
BLK.P                   ; 66- BLOCK A PROCESS
UNBLK.P                 ; 67- UNBLOCK PROC
PRIPR.P                 ; 70- CHANGE PROC PRI
SIGNAL                  ; 71- SIGNAL THE SYSTEM
GUNM.P                  ; 72- GET A PROCESS'S USER NAME
GSHPT.P                 ; 73- GET SHARED PARTITION VALUES
GHRZ.P                  ; 74- GET CLOCK FREQ
DIR.P                   ; 75- DIR
DINIT.P                 ; 76- INIT AN LDU OR MTV
FSTAT.P                 ; 77- GET FILE STATUS
DRLSE.P                 ; 100- RELEASE AN LDU OR MTV
SLIST.P                 ; 101- SET SEARCH LIST
GLIST.P                 ; 102- GET SEARCH LIST
LOGGR.P                 ; 103- MANIPULATE SYSTEM LOG
GBIAS.P                 ; 104- GET BIAS FACTOR
SBIAS.P                 ; 105- SET BIAS
WHIST.P                 ; 106- INIT HISTOGRAM
KHIST.P                 ; 107- KILL HISTOGRAM
-1                      ; 110- GHOST SHARED OPEN
GNAME.P                 ; 111- GET FULL PATHNAME
GNCP.P                  ; 112- GET A CONSOLE PORT NUMBER
SUSER.P                 ; 113- CHANGE SUPERUSER STATUS
SACL.P                  ; 114- SET A FILE'S ACL
GACL.P                  ; 115- GET A FILE'S ACL
PNAME.P                 ; 116- PROCESS NAME <-> PID
-1                      ; 117- RESERVED
FLUSH.P                 ; 120- FLUSH A SHARED PAGE TO DISK
-1                      ; 121- GET A FILE'S ALIAS
GTACP.P                 ; 122- GET ACP'S FOR A FILE
-1                      ; 123- DELETE FILE AND ALL NAMES
GLINK.P                 ; 124- GET LINK CONTENTS
GPRNM.P                 ; 125- GET PROGRAM NAME
LOGEV.P                 ; 126- LOG EVENT IN SYSTEM LOG
DADID.P                 ; 127- GET FATHER'S PID
CPMAX.P                 ; 130- SET CONTROL POINT DIR MAX SIZE
GNFN.P                  ; 131- GET DIR'S NEXT FILE NAME
-1                      ; 132- MAP PERFORMANCE DATA TO USER
RDUDA.P                 ; 133- READ USER DATA AREA
WRUDA.P                 ; 134- WRITE USER DATA AREA
CRUDA.P                 ; 135- CREATE USER DATA AREA
-1                      ; 136- ASSOCIATE A FILE
-1                      ; 137- DISASSOCIATE A FILE
ENBRK.P                 ; 140- ENABLE A BREAKFILE
SATR.P                  ; 141- SET FILE ATTRIBUTES
IS.R.P                  ; 142- IPC SEND/RECEIVE
BRKFL.P                 ; 143- BREAK FILE
-1                      ; 144- ENABLE A SYNC LINE
-1                      ; 145- DISABLE A SYNC LINE
-1                      ; 146- SEND DATA
-1                      ; 147- RECEIVE DATA
-1                      ; 150- DEFINE A POLLING LIST
```

```
-1                      ; 151- ENAB. A M.D.TERMINAL FOR POLLING
-1                      ; 152- DISA. A M.D.TERMINAL FROM POLL.
-1                      ; 153- GET SYNC LINE ERROR STATISTICS
-1                      ; 154- DEFINE EXTENDED CONTEXT
-1                      ; 155- INITIALIZE EXTENDED CONTEXT
-1                      ; 156- RELEASE EXTENDED CONTEXT
SINFO.P                 ; 157- GET SYSTEM INFORMATION
-1                      ; 160- LOGICAL TO PHYSICAL MAP
SCLOSE.P                ; 161- CLOSE A SHARED FILE
MAPR.P                  ; 162- MAP A REGION OF ADDRESS SPACE
CGNAM.P                 ; 163- GET PATHNAME FROM CHANNEL #
SSID.P                  ; 164- SET SYSTEM ID
GSID.P                  ; 165- GET SYSTEM ID
DACL.P                  ; 166- USER DEFAULT ACL (ON/OFF/SET)
CONX.P                  ; 167- CONNECT
DRCONX.P                ; 170- DISCONNECT
SERVE.P                 ; 171- BECOME A SERVER
RESIGN.P                ; 172- STOP BEING A SERVER
MBTC.P                  ; 173- MOVE BYTES TO CUSTOMER
MBFC.P                  ; 174- MOVE BYTES FROM CUSTOMER
PRCONX.P                ; 175- PASS A CONNECTION
DRCONX.P                ; 176- RING SPECIFIC DISCONNECT
VRCUST.P                ; 177- VERIFY RING CONNECT
PRCONX.P                ; 200- PASS A RING CONNECTION
-1                      ; 201- ACCESS THE NETWORK
RNAME.P                 ; 202- HOST ID FROM PATHNAME
RSEND.P                 ; 203- SERVER ?ISEND
ITIME.P                 ; 204- RETURN TIME IN INTERNAL FMT
FNAME.P                 ; 205- SERVER PATHNAME RESOLUTION
HNAME.P                 ; 206- HOST ID<->HOSTNAME
RPORT.P                 ; 207- NETWORK ?TPORT
-1                      ; 210- RESERVED
-1                      ; 211- RESERVED
KIOFF.P                 ; 212- DISABLE KEYBOARD INTERRUPTS
KION.P                  ; 213- ENABLE KEYBOARD INTERRUPTS
KWAIT.P                 ; 214- WAIT FOR A KEYBD INTERRUPT
KINTR.P                 ; 215- SERVER KEYDB INTERRUPT FNCT
VRCUST.P                ; 216- VERIFY CUSTOMER RELATIONSHIP
CTERM.P                 ; 217- TERM CUSTOMER PROCESS
-1                      ; 220- ENCRYPT/DECRYPT
-1                      ; 221- ENABLE SYNC LINE FOR HDLC
-1                      ; 222- DISABLE SYNC LINE USING HDLC
-1                      ; 223- SEND USING HDLC PROTOCOL
-1                      ; 224- RECEIVE USING HDLC PROTOCOL
BNAME.P                 ; 225- NETWORK PROCESS NAMES
PRDB.P                  ; 226- PHYSICAL READ BLOCK
PWRB.P                  ; 227- PHYSICAL WRITE BLOCK
-1                      ; 230- RESERVED
-1                      ; 231- RESERVED
UPDATE.P                ; 232- FLUSH FILE DESCRIPTOR
ROPEN.P                 ; 233- RESERVED OPEN (AGENT ONLY)
RCLOSE.P                ; 234- RESERVED CLOSE (AGENT ONLY)
SWST.P                  ; 235- START WORKING SET TRACE
KWST.P                  ; 236- KILL WORKING SET TRACE
ALLO.P                  ; 237- ALLOCATE DISK FILE ELEMENTS
SOPPF.P                 ; 240- OPEN A PROT. SHARED FILE
PMTPF.P                 ; 241- PERMIT ACCESS TO PROT SHAR FILE
```

```
        -1                      ; 242- RESERVED
        -1                      ; 243- RESERVED
        -1                      ; 244- RESERVED
        -1                      ; 245- ?SIGNAL ( GATE CALL # RESERVED)
        -1                      ; 246- ?WTSIG ( GATE CALL # RESERVED)
        -1                      ; 247- ?SIGWT (GATE CALL # RESERVED)
        RNGST.P                 ; 250- RESTRICT RING LOADING
        RNGPR.P                 ; 251- GET PATHNAME FROM LOGICAL ADDR
        VALAD.P                 ; 252- INNER RING SERVER VALIDATE
        FLTSC                   ; 253- PAGE FAULT PSEUDO CALL
        GMEM.P                  ; 254- GET SYSTEM AVAILABLE PAGES
        LMAP.P                  ; 255- MAP A SYSTEM/USER PAGE
        EXPO.P                  ; 256- SET/CLEAR/CHECK EXECUTE PROTECT
        POKE.P                  ; 257- USER DEB CHANGE C(LOCATION)
        PEEK.P                  ; 260- USER DEB EXAMINE C(LOCATION)
        WIRE.P                  ; 261- WIRE A PAGE INTO WS
        UNWIRE.P                ; 262- UNWIRE A PAGE
        TIMEQ                   ; 263- QUEUE A TIME REQ (MAGIC)
        RINGLD.P                ; 264- LOAD RESTRICTED RINGS
        LEFE.P                  ; 265- ENABLE LEF MODE
        LEFD.P                  ; 266- DISABLE LEF MODE
        LEFS.P                  ; 267- SAMPLE LEF MODE
        -1                      ; 270- RESERVED
        -1                      ; 271- RESERVED
        -1                      ; 272- RESERVED
        -1                      ; 273- RESERVED
        -1                      ; 274- RESERVED
        GTRUNC.P                ; 275- TRUNCATE A DISK FILE
        ESFF.P                  ; 276- FLUSH SHARED FILE
        DVSTT                   ; 277- GET DEVICE STATISTICS

MAXSYS= (.-MCCT.W-1)/2          ; HIGHEST SYSTEM CALL NUMBER (=277)
```

Note: <entry>.P indicates that the call is pendable, that it is found in a system pageable page and therefore requires a CB to execute. However some entries not having the .P suffix also require a CB (example - FLTSC)

The following list of modules handle process management. They are listed alphabetically by module and the entry points are listed as they occur in the listing.

CHAIN - chain to another program
entry points:
GCHAIN.P - chain to another process
MAPR.P - stack initialization

CLNUP - clean up a process upon termination
entry points:
CLNUP.P - cleanup a process for chain/term
WRBRK.P - write a breakfile image
RELMF.P - let go of channels,pages for chain/term
RETCN.P - return process console
PLLRU.P - pull unshared pages off LRU
RESET.P - abort I/O
A.BRKF.P- patch location for agent breakfile

PRCNG - terminate a process
entry points:
RET.P - term demon
FPTRM.P - fatal process termination
TRMPR.P - ?GTERM system call
TERM.P - terminate a process (without subordinate)
PRNCI.P - trap ^C^B entry

PROC - proc a process
entry points:
PROC.P - proc a process
PROPEN.P- open a .PR file
RDUST.P - read ust from .PR file
RD2BLK.P- read first 2 blocks of .PR file
USTVAL.P- checks for valid ust
TCBVAL.P- checks for valid tcb

PROC2 - proc code continued
entry points:
PROC2.P - second half of PROC code
PTRAP - trap from user proc call
CLWS.P - initial working set calculation

PROC3 - proc error handler
entry points:
PRCER.P - common error processor
SETTL.P - set time limit on proc
PERR.P - error handler for memory error in PROC2
PDLSW.P - creates a hot PSI pool entry if both page and swap exist. If one does not exist a panic 14465 occurs.
PSIPE.P - clean out page + swap to prevent a panic
PSWBLD.P- build hot PSI pool entry

RINGLD - load a lower ring
entry points:
RINGLD.P- load a lower ring
VALAD.P - inner ring server address validation
RNGST.P - restrict ring loading

```
                        CTYPE.P - change process type
SOV11       - break file processing
              entry points:
                        BRKFL.P - create a break file ?brkfl system
                                  call
                        IBRK.P  - create a break file with user and
                                  agent
                        ENBRK.P - enable a break file to be taken
SOV17       - initial load of a process and ring 1 initialization.
              entry points:
                        PPRLD.P  - proc initial load deamon
                        R1INIT.P - ring 1 initialization
SSOV3       - routines used in proc
              entry points:
                        RXINIT.P - init ring from the .pr file
                        NUHIAD.P - change ring 1 to get more logical
                                   addresses in the user ring
SWAPFILES   - swap file manager
              entry points:
                        PROC1.P  - continuation of proc code
```

Program Load Option

Every process starts with a working set large enough to accommodate Page 0 (the first 2K bytes of the logical address space) and the program counter (PC) page. The PC points to the instruction which is currently executing in a program.

You have the option, however, of loading part or all of the unshared address space in your initial program file into physical memory. This program load option is useful when the program which you're executing

- is short
- runs briefly
- frequently references a large unshared area.

By loading in pages initially, you save the time incurred by multiple, sequential page faults.

Before you can use the program load option, your system manager must enable the initial program load option during the VSGEN dialogue by indicating the number of pages a process can have at initial load time. You must then use the SPRED utility to edit the preamble of your program file to indicate the address range of the area you want loaded. For details, refer to "How to Generate and Run AOS/VS" (093-000243).

Variable Swapfiles

Memory contention occurs on a system when currently active processes all desire total working sets larger than the memory available. When contention is light, AOS/VS removes inactive pages from processes and keeps them in a "page file" dedicated to that process. If the process later demands the page(s), the system restores them to the working set.

When heavy memory contention occurs, the system picks a process to swap out to disk via a "swap file". Each process has its own swap file in the SWAP directory. By default, these files have a fixed size.

The fixed size, however, can be a disadvantage for certain processes. For example, if the swapfile is 124 pages and the system decides to swap out a process whose working set size is 250 pages, the system has to break up the working set to fit it in the swapfile. When this same process is later swapped back into memory, the process must incur a series of page faults to restore its working set back to 250. For processes with large working sets, this paging can be costly.

To incur less cost, you can set up a system to allow swapfiles which vary in size from process to process.

To allow the use of variable swapfiles,

- During the VSGEN dialogue, the system manager indicates that he

wants variable swapfiles and specifies a default and a maximum swapfile size.

- The system manager gives certain users (those who run programs with large working sets) the privilege of changing their working set size.

- The privileged users edit the preamble of their program files with the SPRED utility. In doing so, they specify a size for the swapfile equal to the typical size of the working set of the program.

Process Types

To manage the multiprocess environment, AOS/VS allocates main memory to processes based on their priorities and scheduling characteristics. Processes fall into two main categories:

- Those which always reside in memory (these are called resident). In general, only the most critical processes in your system environment should be resident.

- Those which the memory manager moves back and forth between disk and memory (these are called preemptible and swappable).

NOTE: Under AOS/VS preemptible and swappable processes are almost identical; for differences, see the section on "Priority Changes" in this chapter. Under AOS, however, a preemptible process ALWAYS has a higher priority than a swappable process.

When you create a process with the ?PROC system call, by default the process is the same type as its father. You may, however, give it another type, if you wish.

Any process can issue the ?WIRE system call to bind pages to its working set. Remember, however, that if you start wiring a lot of pages to a resident process, you'll degrade the performance of the system because of the increased number of pages the system will be unable to swap out when contention occurs.

In addition to any pages you may wire with ?WIRE, AOS/VS automatically wires the Agent of a resident process to its working set (the Agent is that part of AOS/VS which pre-processes system calls and serves as an interface to the operating system.) You may, however, issue an ?AWIRE system call to unwire all the Agent pages from a resident process, except for those needed to support user devices. As a result, you free up some pages of memory and improve the efficiency of the system as a whole. Your resident process, however, may seem less efficient.

As a general rule, AOS/VS keeps interactive swappable processes in memory longer than non-interactive swappable processes. You may change this, however, by setting the bias factors.

Priority Numbers

Eligible processes compete with each other for CPU time, based on their individual priority numbers. AOS/VS uses priority numbers to determine each process's priority. When you create a process, you may assign it a priority number.

Priority numbers range from 1 (the highest priority) through 511 (the lowest). These numbers span three scheduling groups (with no overlap and no gaps), whose boundaries are determined during VSGEN.

Priority Changes

If a process wants to change its own priority, it may issue the ?PRIPR system call. To change the priority of another process, however, the calling process must be in Superprocess mode. (See "Superuser Mode/Superprocess Mode" in this chapter.)

Changing Type

The priority of a process may also change when you change its type with either ?CTYPE or ?PROC. Given that the boundaries of the 3 scheduling groups are

Group 1 = 1 - G1
Group 2 = G1+1 - G2
Group 3 = G2+1 - 511

then Tables 2-1 and 2-2 summarize the changes in priority which occur when a process changes type. Notice that a swappable process can never assume a priority of 1, 2, or 3, but it may APPEAR to do so because of the way priority numbers get mapped (see the discussion of "Mapping" below.)

Priority Changes Going from a Resident or Preemptible to Swappable Type

Priority Before Change	Priority After Change
1 - 3	1 - 3 * **
4 - G1	2 * **
G1+1 - G1+3	1 - 3 **
G1+4 - G2	G1+4 - G2
G2+1 - 511	G2+1 - 511

* This parallels what happens under AOS.

** Although you would see these numbers if you displayed the priority of a process with the CLI PRIORITY command, the actual priorities would be G1+1 - G1+3. See "Mapping" below.

Priority Changes Going from a Swappable to a Resident or

Preemptible Type

Priority Before Change	Priority After Change
1 - 3 **	1 - 3 *
4 - G1	4 - G1
G1+4 - 511	G1+4 - 511

* This parallels what happens under AOS.

** Although you would see these numbers if you displayed the priority of a process with the CLI PRIORITY command, the actual priorities would be G1+1 - G1+3. See "Mapping" below.

Mapping

A resident or preemptible process can assume any of the priority numbers 1 through 511. The system uses this number in gauging the importance of the process during scheduling and displays this same number if you request the process's priority.

To maintain compatibility with AOS, however, AOS/VS has to map priority numbers for swappable processes. As a result, the actual number the system uses in its scheduling calculations and the number it displays when you request the process's priority may differ.

The discrepancy between actual and displayed priority numbers occurs in three cases:

1) If you assign a swappable process a priority of 1, 2, or 3.

2) If you assign a swappable process a priority of G1+1 - G1+3.

3) If a resident/preemptible process with a priority of 1, 2, or 3 changes its type to swappable.

In all three cases, AOS/VS uses a priority number of G1+1 - G1+3 when scheduling the process because a swappable process cannot have a priority of 1, 2, or 3. The system cannot, however, display the numbers G1+1 - G1+3 for a swappable process, and so displays 1 - 3.

In all other cases (4 - G1 and G1+4 - 511), the actual number is the same as the displayed number.

Remember, however, that if you do assign a swappable process a priority of 1 and then it changes type to resident (or preemptible), the resident process WILL have an actual priority of 1, even though the swappable process could not.

Examples of Mapping

1) If a resident process with a priority of 2 changes its type to swappable, the system displays a priority of 2, but it actually uses

G1+2 when scheduling the swappable process.

2) If a resident process with a priority of 3 changes its type to preemptible, the system displays and uses a priority of 3 for the preemptible process.

3) If a preemptible process with a priority of G1+3 changes its type to swappable, the system displays a priority of 3, but uses G1+3 in scheduling the swappable process.

4) If a preemptible process with a priority of G2+44 changes its type to swappable, the system displays and uses a priority of G2+44 for the swappable process.

5) If a swappable process with a displayable priority of 3 (meaning its real priority is G1+3) changes its type to resident, the system displays and uses a priority of 3 for the resident process.

6) If a swappable process with a priority of 5 changes its type to preemptible, the system displays and uses a priority of 5 for the preemptible process.

Process Scheduling

AOS/VS schedules eligible processes based on their priority numbers and scheduling characteristic. As you may recall, the range of process priority numbers (1 through 511) spans three scheduling groups.

Group 1 ranges from 1 to a number, "G1", which is set during VSGEN. AOS/VS schedules any process whose priority number places it in Group 1 on a round-robin basis. Under this scheme, each process is allocated a uniform slice of time during which it may execute. Once a process of a specified priority temporarily stops executing (having used up its time slice), it is not chosen to execute again until all other processes of that priority have been chosen to execute.

Group 2 ranges from G1+1 to a number, "G2", which is also set during VSGEN. AOS/VS schedules any Group 2 process heuristically, which means that the system takes the process's past behaviour into account when alloting it an interval of time during which it may execute.

Group 3 ranges from G2+1 to 511. AOS/VS handles processes in this group on a round-robin basis.

NOTE: If you need to maintain compatibility with AOS, G1 and G2 must be set to 255 and 258, respectively.

Group 1 processes are always more important (that is, more likely to be chosen for execution) than those in Group 2 or 3, and Group 2 processes are always more important than those in Group 3. Within each group, the lower the priority number, the greater the importance of the process. The importance of a process may, however, alter as a result of a change in type.

If an executing process cannot proceed, you can issue the ?RESCHED

system call, which allows the calling process to give up control of the CPU and forces AOS/VS to immediately schedule another process for execution.

Process creation

Process creation is one of those AOS/VS functions that is not the responsibility of anyone module or routine. It involves system call call processing, COREM, daemons and other such things. The following is an attempt to follow a process creation from the time at which a process performs a ?PROC until the new process is healthy and strong, ready to assume its place in the AOS/VS world.

?PROC Create a New Process

The ?PROC is first pre-processed by the AGENT, which builds an initial IPC message from the user packet that contains the names of the generic files @LIST, @INPUT, @OUTPUT, @DATA. This IPC message is sent to the new process to be picked up when the new process's AGENT starts up. The AGENT then enters into the kernel through the normal system call path. AOS/VS does the actual processing and then control is returned to the AGENT for some post processing.

System call trace:

1. Meter the number of PROC requests (PROCRQ.W)

2. Set up a temporary CB fault handler (may fault when reading the user's packet)

3. Copy the user's packet onto the CB stack

4. If the caller does not have the unlimited sons privilege, count up the number of son and make sure the caller will not exceed the assigned limit

5. If the caller wants to create with block, return an error if the caller is resident. If the caller is creating without block, return an error if the caller does not have the privilege to do so.

6. Allocate a PTBL and PEXTN from GSMEM (72. and 1K words long)

7. Set up some initial values (point the PTBL at itself and the PEXTN, set the initial load, the swap in progress, and the daemon start bits). If a full breakfile (MDUMP) is requested, set the bit.

8. Check process priority and check for proper ranges and ability to change priority.

9. Set up the working set minimum and maximum offsets in the PEXTN

10. Open the .PR file, and insure that it is an executable file type, that the user has execute access and that it is at least 1K long.

11. Store the CID (Channel ID or the address of the CCB) in the PEXTN.

12. check for extensible swapfiles if so set up new swapsize = to smallest of preamble, swapsize or new WSMAX.

13. Allocate a 256 word chunk of GSMEM and read in the second block (block 1) of the .PR file using the NQCRQ routine.

14. Verify that the TCB address pointed to in USTCT of the UST is 446

15. Validate the active TCB queue (must also be 446)

16. Store away the initial user PC and task count for the AGENT and make sure the task count is valid (1<=taskcount<=32)

17. If the user is procing up a process of a different type (16 vs 32) insure that the user has the privilege

18. Validate the processes address space (shared does not overlap the unshared, initial PC is in the valid address space, etc.)

19. Release the GSMEM chunk allocated in step 13.

Call to PROC1.P in SWAPFILES.

20. Setup the PSMEMQ, PWSMQ, PFRMQ, PRBQUE, ILAQUE, and PIORR/PIORB chain pointers

21. Check for pre-paging + initial load. If set load it in.

22. Set up single or multi level .PR

23. Set up the concurrent system call number

24. Allocate the swap and page files Note that this will return a PID number

25. Set up the new processes username

(At this point, the code chains from PROC1 to PROC2)

26. Set up default and working directories

27. Store the subslice length (32 ms) in the PSL offset of PEXTN

28. Expand the PIDTB if necessary (if PID#>PIDLN it must expand)

29. Update the appropriate son and brother pointers

30. Set up the process name in the PIF

31. Create and initialize the IPC spool file, initialize the spool file directory chain (on the PTBL), and send the initial user and AGENT IPCs

32. Set up the new processes searchlist (same as fathers)

33. Put the ring 7 .PR file name into the PIF.

34. If the user specified a max CPU time, set up the values in the PEXTN

35. If a console is being assigned to the new process, connect the new process to the console controller (PMGR or SVTA), and send an IPC to the console controller. Then wait for the PMGR or SVTA to signal the completion of the console assignment.

36. If the proc'er has received a ^C^B during the proc, abort.

37. If requested, pass along the default ACL

38. Set up the initial working set requirement in the PTBL extender as follows (estimate initial process memory requirement)

    a. Calculate the number of pages required in the AGENT for the TCBs and the number of pages required in ring 1 for the virtual TCBs (call this sum A)

    b. For a 16 bit process, we will need pages for the following:

        1 low level PTP for ring 7
        1 high level PTP for ring 1
        1 low level PTP for ring 1
        1 data page for ring 1 (page file directory)
        1 high level PTP for ring 3
        1 low level PTP for ring 3
        A+1 data pages for ring 3 (A from step a above +1 for the initial AGENT PC page)
        If resident, all the shared and unshared pages, otherwise 2, the initial PC page, and page 0

    For a 32 bit single level process:

        1 low level PTP for ring 7
        1 low level PTP for ring 1
        1 data page for ring 1 (page file directory)
        2 low level PTP for ring 3
        A+1 data pages for ring 3
        1 or 2 data pages for ring 7 (only 1 if the initial PC is in page 0
        1 high level PTP for ring 1
        1 high level PTP for ring 3

For a 32 bit two level process, we allocate as for a single level process plus:

1 high level PTP for ring 7
(1) low level PTP for ring 7 if necessary

39. If the caller is to block, set a bit to tell the scheduler to block the caller when all system calls are completed (BPFEB)

40. If the new process is resident attempt to grow the resident CB pool (if this fails, abort the proc)

41. Set up the connect time and day, and the initial PNQF

42. If the new process is to be blocked after initial load, set the appropriate bit

43. Enqueue the new PTBL to the appropriate ineligible queue

44. Meter the number of completed procs (PROCFN.W)

45. The system call is complete (the caller can now continue if this was not a proc/block call)

Time passes ... Eventually the process will get the initial required memory (which will be enqueued off of the processes memory queue, and the PTBL will be moved onto the ELQUE. The PTBL will get control of the CPU and it the scheduler will start up the initial load daemon. This will then take us to IPRLD.P in SSOV3. Then ...

1. Set up the PTPs. The memory for these pages will have been allocated by GCORE and linked of the process' memory chain

2. Set up the SBRs to point at the new PTPs

3. If the process is narrow (16 bit) and resident, read and wire in the entire working set. Otherwise, read in the user's page 0 and initial PC page (stored by the PROC code at step 16) and initial load area if set.

4. If the process is to come up in the debugger, set the appropriate flag in DEBFLAG

5. Chain to the initialize AGENT code (still called IGHOST.P)

The IGHOST.P code is located in SOV17 and does the following:

1. If the PMGR is running (MPMGR<>0) then open up AGENT.PR, and put the CID into the ring 3 .PR file CID location in the PEXTN. If the PMGR is not running (MPMGR=0) then open LPMGR.PR (IOP or IAC) and the CID into the ring 3 .PR file CID location in the PEXTN. (This will initialize the PMGR if PID 1 or the AGENT if not PID 1)

2. Build the PTPs for ring 3, and store the information in the PEXTN

3. Get a data page for page 0 (note that the page should be on the process' memory chain and VS will panic if the get page call fails)

4. Read into a system buffer the UST for the AGENT so we can obtain the information about the AGENT's shared area

5. Read AGENT block 0 into the ring 3 address space

6. Allocate (from the process' memory queue) enough pages to hold the TCBs and add the pages to the ring 3 PTP structure

7. Read in the TCB pages from the preamble

8. Set up the initial starting address (either AINIT or ADEBUG), and store the user's starting address in AGAC2.W

9. Fault in the AGENT PC page (which cannot already be there because the page is shared)

10. Wire in the TCB pages

11. Set up the initial (primary) AGENT stack

12. Set up the user's stack (based on whether the user is 16 or 32 bit)

13. Reset the initial load bit, and flag that the initial load is complete

14. Update the working set to reflect the additional pages added

15. Clear the process sched action bit (so that the process will now run)

16. Dismiss the CB

When the scheduler next scans down the ELQUE, it will find the new process ready to run, and will give control to the one TCB that exist. If this is PID 1, the PC will point to the PMGR init code, else the PC of that TCB will point to either AINIT or ADEBUG.

1. If the entry point is ADEBUG, set a flag to indicate so.

2. Build a fake return block on the current stack, with all ACs=0 and the PC = the user's starting address (passed by the kernel)

3. Initialize the memory manager

4. Build the TCB free chain

5. Allocate memory for the memory database (store the address in AMEMDB)

6. Get the memory for the AGENT stacks (192. words per task).

7. Point each TCBs ring 3 SP and FP at the allocated stacks

8. Initialize the ring 3 and ring 7 memory allocation tables using information obtained from the kernel via ?MEM and ?GSHPT

9. Copy the first 21. words of the AGENTs UST into the user's UST

10. If the user is 16 bits, and an overlay descriptor table is defined, call AINIT16. which will allocate the memory needed to hold an overlay descriptor table within the AGENT space, and will copy the ring 7 table into the ring 3 table

11. Read in the initial IPC

12. If the user did not specify the PROC/DEBUG option, then perform a WRTN which will pass control to the user. Otherwise, jump to the debugger (LJMP XDEBUG)

Process Termination

There are basically five ways that a process can terminate.

1. Direct termination -- (self termination and forced termination by a different process)

   ?RETURN

   ?TERM

2. Console interrupt -- (user forced by typing interrupt key)

   ^C ^B

   ^C ^E

   MODEM DISCONNECT

3. Trap

   See the section on traps (there are 11 different hardware traps defined by the MV series ECLIPSEs.)

4. Father termination

5. Fatal process error - (While processing a system call or internal routine, AOS/VS has taken an unrecoverable error path and must terminate the process

Code paths:

Hardware Traps while not in ring 0, ^C^B, or ^C^E (PRNCI.P)

1. Save away the current task information

2. Join the common code below

Fatal process errors (FPTRM.P)

1. Store the error code (from the CB) into the process table extender, and set the fatal term bit in the process table

2. Save the TCB at the time of the error

3. Zero the CB's TCB pointer to prevent the unnecessary awakening of the TCB when the call CB completes

4. Jump to common code below

Self termination / forced termination (TERM.P)

1. If a forced termination (i.e. TERM 12), validate the target PID of the command.

2. If self-termination, set the self term bit (BPFST)

3. Zero the CB's TCB pointer to prevent the unnecessary awakening of the TCB when the call CB completes

4. Jump to common code below

Common code (module PRCNG)

1. If the process is already terming ignore this termination, otherwise set the first term bit.

2. Call PTREE to block the process' entire inferior process tree.

Begin main termination loop:

3. Look for a process that does not have a son, and call PBITS passing the found PID as the parameter.

PRCNG

SGSB2

4. PBITS will:

   a. If the process is faulting, set the term after fault bit.

   b. If the process is not swapping, clear sched action (allow things to happen)

   c. Set the break (or interrupt) bit in PSTAT

d. Set the terming bit.
e. Reset the not ready bit.
f. If the process is blocked, unblock it (or have the coremanager unblock it if at interrupt level)
g. If the process is not in core. regenerate the PNQF to speed things up, and flag the COREM to swap in the process

SGSB2

PRCNG

5. We then go back to the scheduler, which will find the process we just PBITed ready to run, and start a termination daemon for that process. The daemon will start at RET.P in PRCNG

6. Increment TERMRQ.W (number of term requests)

7. If the terminating process now has a son (which was proced during the termination cycle, go back to step 2 above)

8. Change the WSMAX for the terming process to the system default.

9. Give control to the processes AGENT to allow it time to clean up databases, and flush AGENT buffers. Control will be transfered unless:

   a. This is a self term. The ?RETURN or ?TERM self system call will have started in the AGENT which will have flushed its buffers before passing control to the kernel.

   b. The terminating process has trapped in the AGENT. If the trap was in the AGENT, the system is uncertain about the state of the databases and buffers and therefore will not allow them to be flushed.

   c. The terminating process was initially loading. Since the AGENT has never run, and in fact does not even exist for this process, we can not give control to it.

   The system will force the current TCB for the user to execute the AGENT cleanup code. It will also disable task resheduling for the terminating process, prevent the process from ever blocking, and prevent the running of outstanding system calls enqueued off of the PEXTN. The code will also tell the AGENT to perform a full breakfile (MDUMP) if this option was requested at PROC time.

10. If the father has resource limiting on, decrement the time limit by the time used by this process

11. If this is a trap, then:

a. If this is the PMGR trapping, then PANIC 12010
b. Create the breakfile CCB (delete, create, open the file)
c. Notify the PMGR about the termination (via IPC)
d. Call CLNUP.P (see documentation below)
e. Call SIPCD.P (which will format the termination IPC message)

If this is a ^C ^B or ^C ^E, then:

a. If this is a ^C ^E, then create the breakfile CCB
b. If this was a modem disconnect, process as a FATAL ERROR (see below)
c. Notify the PMGR via IPC about the termination
d. Call CLNUP.P
e. Call SIPCD.P

If this is a FATAL error, then:

a. Notify the PMGR
b. Call CLNUP.P
b. Call SIPCD.P (termination code = 4, terminated by the system)
d. If this is a modem disconnect, then put the error (175) in the IPC to be sent; otherwise. put the error from the process table extender into the IPC message.

If this is a self termination, then:

a. Set up the termination IPC header and if a user termination message was specified, copy it onto the stack.
b. Notify the PMGR about the termination
c. If there is a user message, send it (SIPC.P).
d. Call CLNUP.P
e. Join the code path below at step 13.

If this is a term by AOS/VS (because the father termed), then:

a. Notify the PMGR about the term
b. Call CLNUP.P
c. Call SIPCD.P

12. If SIPCD was sucessful, send the IPC message (SIPC)

13. Release the system CCBs associated with this process' directories

14. Release the swap and page files associated with the process.

15. Unlink us from our fathers son list.

16. If we are PID 2, jump to DEATH, the routine to shut the system down

17. Release the process' unshared memory area (including the PEXTN)

18. Unpend any processes waiting for terminations (call UNPEND)

19. If the father is not the root. and the father is not terminating, then unblock him if he is waiting for son termination.

20. Call the core manager (memory is now available)

21. Update PIDTB and PIDBT (zero the double word pointer in PIDTB and clear the appropriate bit in PIDBT)

22. Unpend processes waiting on SKTR (son termination during system shutdown

23. If this is a resident process terminating, return one resident CB to GSMEM

24. Release the process table to GSMEM

25. If our father is the root, or not terminating, we are done Otherwise, if the father has another son (we are survived by a brother), terminate that son (and its sons and...) otherwise, term the father. (Jump to step 3 above)

CLNUP.P

(Note that this routine is also used by the ?CHAIN code)

1. If this is not a ?CHAIN, build and post a logfile termination message.

2. Release user devices (IDEF)

3. If a breakfile CID exists (i.e. we are creating a breakfile) then do the following in this order:

   a. Copy PFLAG words and the trap code from the kernel space into the AGENT page 0
   b. Write out the AGENT page 0 and TCB pages
   c. Close the breakfile

4. Delete any user created IPC entries for this process.

5. Dequeue outstanding IRECs

6. Dequeue outstanding spoolfile directory entries from the PTBL chain

7. Close IPC spool file.

8. Inform the Connection manager about the termination (TBC in CONX)

9. If this is a chain, terminate any sons that were proc'ed from rings 3 - 6.

10. Remove this process from the delay and histogram chains if appropriate.

11. If a WS trace is in effect for this process, stop it.

12. Wait for all system calls targeted at this process to complete.

13. Release the shared areas for rings 3-7. This is done by examining the CME for each page in the WS and removing the page (via FREL) if the 'shared' bit is set.

14. Close any open files (call the routine RESET.P)

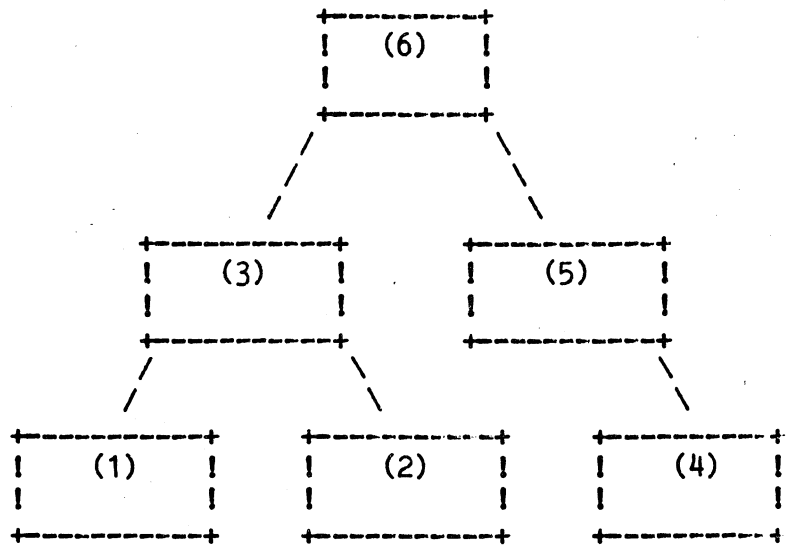
15. Search the LRU for unshared pages belonging to this process and put the found pages onto the FC1024 chain.

16. Close each .PR file open for ring 1-7

17. Release the searchlist (if not CHAIN)

18. Dequeue process from ELQUE. (end of CLNUP)

Process termination diagram -- the numbers indicate order in which the processes will terminate.



System shutdown (DEATH)

1. Term all processes (set term bits in all processes)
2. Pend root process until all others are gone.
3. Turn off system log.
4. Close PIF
5. Release :PER directory CCB
6. Release the :SWAP and :PAGE directory CCBs to the hot PSI pool if the pool is not full.
7. Release all LDUs still initialized (the root [:] is last to go)
8. Tell the world and halt the processor.

The following modules handle memory management.

CLUSTER - prepage at load and fault time
entry points
ILCLSTR - initial load prepaging
CLUSTER - fault time prepaging

COREM - core manager
entry points
COREM - core manager
CMINT - core manager task
CWAKE - force a reschedule of core manager
CMENQ - enqueue a process to core manager
EXPTB - exponent's elque counter table

COREM2 - part 2 of core manager
entry points
GPNQF - generate priority enque factor
CTBLK - block a process
IUNBLK - interrupt unblock a process
CTUNBLK - unblock a process
PDEQ - remove a process tabe from a queue
PENQ - enque process table to end of queue
TSPRC - process end of time slice
TSUP - process end of time slice
PRBAG - mark process for swapout
FITER - enque a process to one of the ineligible swapped queues
NOSYS - check for any outstanding system calls

FAULT - fault in pages
entry points
FAULT - fault a page into the working set
FAULTS - special entry into fault for ?SPAGE
FLTWR - fault and wire a page into working set
FLTWRS - special entry to FLTWR for ?SPAGE
UNWIRE - unwire a page in a working set
UNWIRS - special unwire entry for ?SPAGE

MAPR - memory mapping routines
entry points
CKSWIP - check for eligible to have page swiped
LDPTBL - force load a PTBL
RLPTBL - release a PTBL
GPTBL - convert a PID to PTBL
FUCCB - find user CCB
DFAULT - find and fault in user CCB
MAPU/
MAPUNW - read in user pages with full access
RMAPU/
RMAPUNW - read in user pages with read/execute access
MAPCU/
MAPCUNW - map in a user page
RMAPCU/
RMAPCUNW- map in a user page
UNMAP - unmap a user page

MEMRY - memory management routines
entry points
RPMEM - release memory on the PSMEMQ

RMVWSB - update WS bit and ref bit arrays PLPHYS, and PHPHYS when removing a page
PGSMEM - GSMEM that looks on the PSMEMQ
PGSMEMNW- GSMEM that looks on the PSMEMQ (no wait)
CKWMAX - check for exceeding WSMAX of process
PGMBLK - GMBLK that looks at PSMEMQ
GMBNW - GMBLK with no wait
GSMNW - GSMEM with no wait
GMCNB - get a block from only the candidate chain
GSMRS - get a page and wait as well as restart TCB
GSMEM - get a page or pages from memory
RSMEM - release a piece of system memory
GMBLK - get a block of memory
GMBRS - get a block and restart TCB
RFBLK - release a 1k page to free chain
RFBLKN - release a 1k page do not change MKEY
RSBLK - release a shared 1k block of memory
RUSBLK - release an unshared 1k page to LRU

MEMRY2 - rest of memory management entry points

GETSPH - allocate an SPH
RELPMEM - release an entry from PSMEMQ
TGCHK - verify validity of target process
PGLPA - get an LPA from PSMEMQ
GLPA - get an LPA
PSMFREE - release memeory to it proper chain
GFCB - find a free FCB
RFCB - release an FCB
KPFCB - find and mark as allocated a free FCB
SHFLS - flush (write) shared pages reset modified
SHFLSNR - flush (write) shared pages no reset
DQCHD - dequeue a core map header from LRU
GCORE - allocate ws bit array ref bits process mem
RCORE - release ws bit array ref bits ptbl ext mem

MEMRY3 - logical ring 0 memory management entry points

DALCB - deallocate a control block
ALCB - allocate a control block
ALOPTES - allocate pte's for ring-0 logical space
ALOMEG - allocate ring-0 low level ptp
DLSAL/
DLSREL - allocate/free dynamic logical slot
ESAL/
ESREL - allocate/free extender logical slot
GETXSPC - get space in process' extender slot
FREXSPC - free space in process' extender slot

PFF - page fault algorithms entry points

CLRFAY - clear reference array

```
                    CREF    - copy reference bit array
                    PFF     - page frequency algorithm
                    PFFUFT  - page frequency algorithm
PGFLT   - hardware page fault handler
          entry points
                    CPDESC  - decrement control page use count
                    CPINC   - increment control page use count
                    PTPDEC  - decremint page table use count
                    PTPINC  - increment page table use count
                    FLTSC   - page fault pseudo system call
                    PGFLT   - hardware fault handler
                    NPGFLT  - hardware fault handler
                    FIXCB   - converts direct to indirect call
PGREL   - release memory pages
          entry points
                    FREL    - release all logical pages
                    LKLPA   - find a logical page associator
                    LPAPREL - release a logical page associator
                    PRELS   - special entry into prel for ?SPAGE
                    SYSREL  - release ring '0' overlay page
                    WSQREL  - release all of a process' removeable
                              pages
PRMPT   - preemption for memory contention
          entry points
                    PRPR    - preempt a process for core mgr
                    MSOLV   - need a page frame can wait for it
                    PRPG    - need a page frame can't wait
PRSUB   - page file management routines
          entry points
                    APGFRD  - find a region descriptor backed up by
                              the page file in a process's address
                              space
                    CPGFFBLK- get a disk page for an unshared page
                    READPG  - read a page from page file
                    READPR  - read a page from the .PR file
                    UNFLUSH - flush an unshared page to page file
                    UNFLSH1 - flush unshared page but also pass in
                              PTEL
                    STARTIO - queue I/O for a page fault
                    READSY  - read system page
PTE     - routines for examining and modifying PTE's
          entry points
                    GETPHYS - get physical address for logicals
                              address
                    RGETPHYS- same as GETPHYS but panic if not
                              resident
                    GETPTE  - get contents of PTE
                    GETPTE.O- get PTE but wire calling overlay first
                    GPTEA   - get the address of a PTE
                    GPTEANW - no wait version of GPTEA
                    GETPTENW- get contents of PTE but never pend
                    GPTEA.O - get PTE address but wire calling
                              overlay
                    SETPTE  - set the contents of a PTE
                    GETSBR  - get contents of SBR
                    GETLEV1 - get level 1 page table
                    GETLEV2 - get level 2 page table
```

SPH - routines for shared page headers
entry points
DQSPH - deque a SPH from the FCB SPH chain
FSPHDQ - searches for a sph on system wide sph chains. if found then dequeues it
FSHPNDQ - searches for a sph on system wide sph chains but does not deque
SPHCHINT- init the FCB SPH chain QD entries
READSH - read a shared page

STEAL - page stealing routines
entry points
DCMEPWSC- deque a CME from the removable chain
NEWPS - steal a physical page using removable array
PSTEAL - steal a physical page from the WS array
QCMEPWSC- enqueue a CME to removable chain
RPGFWS - steal a logical page from WS array

SWAPIN - swap in a process
entry points
SWAPIN - swap in a process from swapfile
SWAPIO - perform swap file I/O
BLDSWAP - build ring 0 swaparea PTP from WS bit array

SWAPOUT - swap out a process
entry points
SWAPOUT - swap out a process image
CLNLRU - flush LRU of any pages being swapped
SWSINIT - set up for a scan of working set array

TRACE - working set trace
entry points
TRACE - working set trace maintenance
STRACE - system WS trace maintenance
KTRACE - rip down WS trace
KSTRACE - rip down a system WS trace
RCHEX - get an extender to examine
WCHEX - modify an extender

VMSUB - virtual memeory management routines
entry points
GENCCBAD- convert user channel to logical ring 1 CCB
VALPAGE - validate a requested page
RESPAGE - check if page resident
WORKPAGE- validate page part of working set
LRUCK - check if requested page on LRU
GENPTADD- generate a ring 1 address of PTE
GENCDADD- generate a ring 1 address of CDE
PXTNPG - process table extender page address
CCBTAB - ccb table address
VTCBS - address of start of vtcb
VTCBTAB - address of start of vtcb area
SVTCBS - address of start of svtcb area
BUILPT - build 1 level page table
HBUILDPT- build high level page table (level 2 translation)
LBUILDPT- build low level PTP (level 2 translation)

Demand Paging

AOS/VS is a demand-paged, virtual-memory operating system. Virtual memory means that memory is a composite of main memory and disk memory. Demand paging is the AOS/VS method of adding logical pages to the working set of a process as the process "demands" (refers to) those pages. The working set is that subset of a process's logical address space which is currently in memory. The working set changes in size and content as the process references pages.

The pages outside the working set make up the process' virtual address space.

Pre-Paging at Fault Time Option

By default, when a page fault occurs (that is, a process demands a page), the system adds one page to the working set. You have the option, however, of requesting that the system add the faulting page, plus a cluster of logically contiguous virtual pages, to the working set at fault time. This option is known as "pre-paging at fault time". Pre-paging is the process of adding unreferenced virtual pages to a working set.

The pre-paging option is useful when

- a program includes large array-like structures (large meaning that the virtual addresses of the structure exceed the main memory available)
- the algorithm which processes the structures tends to reference the entire structure or parts of it sequentially
- the area in which you want pre-paging to occur is in unshared or unused memory

If your program has such characteristics and you understand its page referencing patterns, pre-paging can speed up execution considerably. The system is far more efficient when it moves a cluster of contiguous pages into main memory than when it moves them in one by one.

Before you can use the pre-paging option, your system manager must set the pre-paging parameter during the VSGEN dialogue (if the pre-paging parameter is either 0 or 1, pre-paging is off system-wide; otherwise, this parameter indicates the maximum number of pages you can add to the working set during one fault).

Assuming pre-paging is enabled, you must then use the SPRED utility to edit the preamble of your program file. When you do so, you indicate:

- the starting and ending addresses for the cluster area (remember this must be an unshared or unused portion of memory)
- the cluster size in pages

Getting a page frame

When attempting to get a page frame, ie. a physical memory page, AOS/VS will first attempt to grab a free frame, and failing this, will attempt to preempt a lower priority process. The two basic routines to do this are:

GMBLK -- get a page frame
PGMBLK -- get a page frame, but first examine the per process chain of assigned memory

The basic algorithm is as follows:

1. Attempt to get the memory from GSMEM by examining the FC1024 chain

2. Attempt to get a page from the LRU

When these preferred methods fail, no free page frame is available, and it is time for:

Preemption

The preemption code will release pages to the LRU. PRPR, located in PREEMPTS.SR is responsible for all preemption, and can be called under a number of different circumstances. The routine is outlined below:

Note: In order to prevent certain race conditions, a time stamp has been added to revision 1.60. When a process is picked to be preempted, the time is recorded in the PEXTN (in PSWOTM.W) and in the global last swapout time flag GSWOTM.W). When the process sucessfully swaps out, these two locations are cleared. When the Core manager again tries to preempt, and finds the same process as the best candidate to swap out, it examines the time stamp. If not 0, and more than 3 seconds have elapsed, it is assumed that the process can not swap out, and a second choice is found. The scheduler before entering the checksum loop will examine the Global last swapout word, and if more than 3 seconds have elapsed since the last process was selected for swapout, it will wake up the coremanager so that the alternate can be found.

1. If this is a CMTSK or SMTSK request, attempt to use an extraneous free system pageable page (overlay), and return an error if none exist (CMTSK and SMTSK will handle the error)

CMTSK will call PRPR attempting to get memory for a PTBL extender
SMTSK will call PRPR attempting to fault in UNERR.P

2. If the process the preemption is on behalf of has been marked for swapout, just return.

3. Turn on PFF call PFF on all processes found on the MBLKQ and BLKQ (PFF will actually release pages to the LRU). If this adds enough pages to the LRU to satisfy the request, return.

4. If this is a single page request:

   a. If we are faulting on a system pageable page, attempt to reuse a free system pageable page.

   b. If there is currently a process enqueued for swapout return with the PTBL address of that process as a key.

   c. From the back of the MBLKQ, attempt to find a process that can be swapped out (not the target of a system call, and not resident [resident processes will end up on MBLKQ if they were blocked as swappable and then a change process type was aimed at the process]). If one is found, return the PTBL address as a key.

   d. From the back of the BLKQ, attempt to find a process that can be swapped out (not the target of a system call, and not resident [resident processes will end up on BLKQ if they were blocked as swappable and then a change process type was aimed at the process]). If one is found, return with the PTBL address as a key.

   e. Attempt to steal a page from a lower or equal priority process on the ELQUE.

   f. Attempt to steal a page from the process that needs the preemption.

   g. No preemption is possible

5. If this is a multiple page request:

   a. See if the request can be satisfied simply by releasing the excessive system pageable pages (OVCNT>OVMIN)

   b. If there is a process enqueued to swapout that will release enough pages to satisfy the request, return the PTBL address as a key.

   c. Scan first the MBLKQ, then the BLKQ, and finally the ELQUE

from the back looking to see if enough processes can be swapped out to satisfy the request. If so, mark the first process found for swapout. (When the swapout completes, the CORE MANAGER will rescan for the next process to swapout).

By starting at the back of the MBLKQ and BLKQ, we will find the processes that have been on the specific queue the longest.

By starting at the back of the ELQUE, we will find the lowest priority process. However, special rules apply.

Do not violate the BIAS factors

If the preempting process is resident, preemptible, or swappable completing a move bytes, only non-resident processes of lower or equal priority can be preempted.

If the preempting process is swappable, do not preempt:

1) a resident or preemptible process
2) a swappable process of lower PNQF
3) a swappable process of equal priority if it has a lower timeslice exponent
4) a process of equal priority, equal exponent, that has never completed a complete timeslice
5) a process of equal priority, higher exponent unless it has run enough subslices to equal the number of subslices that the preempting process would have run

d. Attempt to steal enough pages from the lower priority processes on the ELQUE that will not fit into the swap file (WS>124) and has enough excess pages(>124) to satisfy the request.

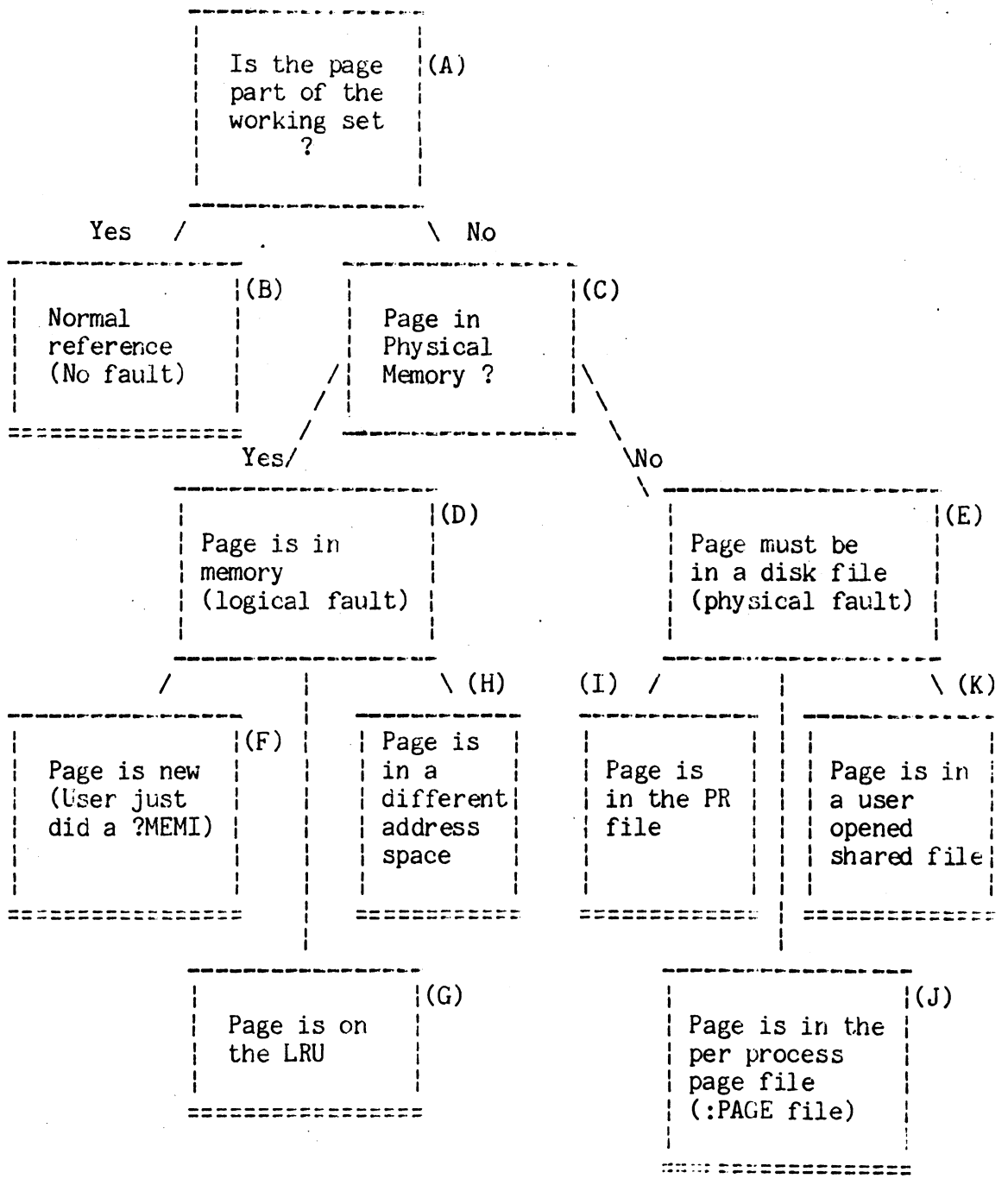
e. No preemption possible, take error return

Accessing memory and page faulting

The following section describes the AOS/VS fault processing mechanism. This includes both the hardware and software processing, in addition to the Page fault frequency (PFF) algorithm and time stamping. The first section is an overview of the entire process.

Overview

When a user, or the system accesses a page of memory, the following can happen:

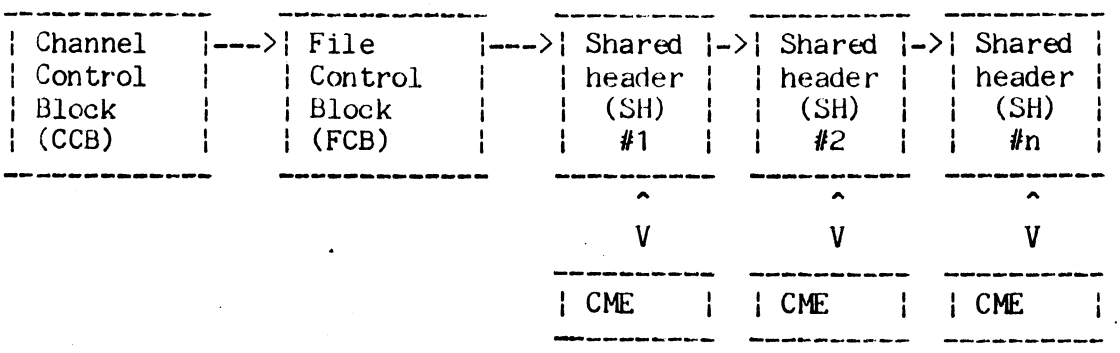


The following list compares a specific type of memory reference to the logic path involved. The letter sequence corresponds to the boxes in the previous diagram.

Normal access to a page within the W.S in either the shared or unshared area.	AB
The user does a ?MEMI, and then accesses the page for the first time	ACDF
The user accesses an unshared page that has been removed from the W.S, but has remained in physical memory (on the LRU)	ACDG
The user accesses a shared page that is currently in use by someone else.	ACDH
The user accesses a shared page that was in use at sometime before, and is still in physical memory (on the LRU)	ACDG
The user accesses an unshared page that was never modified during this .PR execution and is no longer in the w.s or on the LRU	ACEI
The user references an unshared page that has been previously modified and is no longer part of the w.s or on the LRU	ACEJ
The user references a shared page within a .PR file, and that page is not on the LRU or in another processes working set)	ACEI
The user references a share page within a data file (opened via ?SOPEN or the equivalent)	ACEK

***** NOTE: With pre-paging enabled there are more pages read in at fault time. The routines are primed for this prior to performing the actual faults. This is the only change in the following logic.

There are a number of databases used by AOS/VS (and the MV hardware) in determining the current state of a logical page of memory. Most of these are discussed earlier in this chapter. For completeness, the following diagram shows the other relevant databases and their relationships to one another. These database will be discussed in more detail later in this chapter.



There is one CCB per open user channel. These channels are the one that a user references in a ?RDB or ?WRB system call. If two users have the same file open, each will have their own CCB.

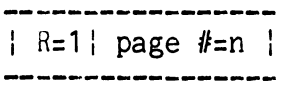
There is one FCB per open file system wide. The FCB is pointed to by the CCB and contains information about the file (i.e. where it is on disk, and an open count). There is one FCB per file regardless of how many processes have opened it. If two users open the same file, the use count in the FCB will be 2, and both CCBs will point at the same FCB.

There is one SH per shared page in physical memory. A shared page is associated with a file, not a user. Therefore the SHs are linked off of the 'per file' database, or the FCB. The SH contains, among other data items, the physical page number of the shared page.

The following diagrams illustrate the states of the databases during each type of memory reference (AB, ACDF above). Note that N = x indicates a don't care state for N.

AB - normal reference

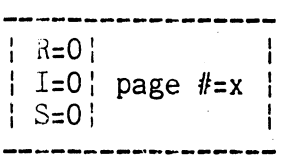
PTE



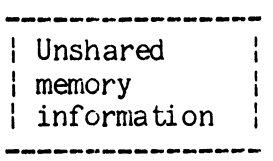
This reference is simple, the R=1 indicates that the residency bit is set, therefore the page is part of the W.S. The ATU knows what physical page the instruction is referencing because it is in the PTE (page #).

ACDF - reference to a ?MEMIed page never before referenced

PTE



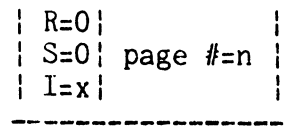
PEXTN



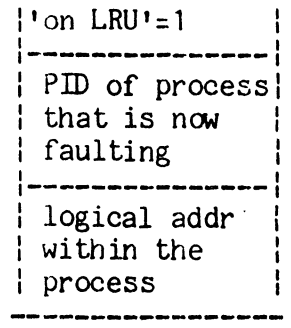
The ATU knows that the page is not in the WS because R=0. The fault occurs transfering control to AOS/VS. The OS examines the PTE software bits and knows that the page is not shared (S=0), and has not been referenced and modified before (I=0). Therefore the page must be either in the .PR file on the disk, or be a new ?MEMIed page. The fact that it is a ?MEMIed page can be calculated using data within the PEXTN, specifcally the highest unshared .PR page within the file. If the logical address of the faulting page is higher than the value in the PEXTN, the page must have been added via ?MEMI. AOS/VS will make a call to GSMEM for one page, put the physical page number into the PTE, and set the residency bit within the PTE.

ACDG -- Logical fault, page is unshared

PTE



CME for page n

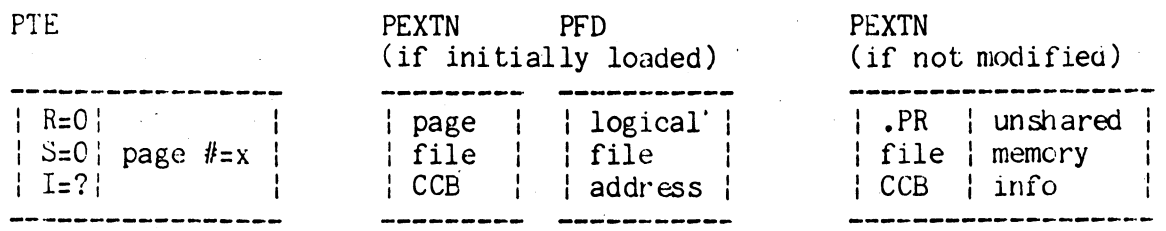


After AOS/VS receives control from the MV fault hardware, the OS sees that the page is unshared (S=0). It then examines the page number in the PTE, and looks in the corresponding CME to see if the 'on LRU' bit is set. If so, AOS/VS will then look to see if the logical address pair within the CME matches the PID/logical address pair of the current fault. If it does, the page is removed from the LRU, the 'on LRU' bit is cleared, and the residency bit in the PTE is set.

ACEI -- page is unshared, not on the LRU, initially loaded (modified)

or

ACEJ -- page is unshared, not on the LRU, not initially loaded

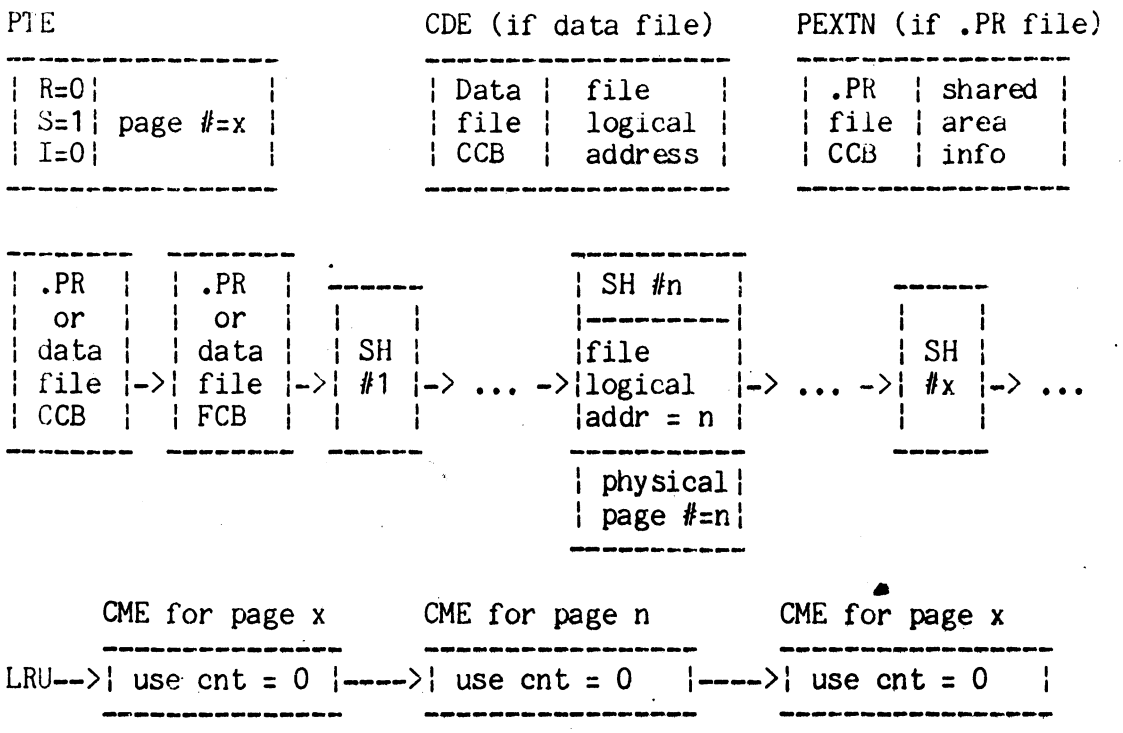


After AOS/VS receives control from the MV fault hardware, the OS sees that the page is unshared (S=0). It then examines the page number in the PTE, and looks in the corresponding CME to see if the 'on LRU' bit is set and that the logical address/PID in the CME matches the logical address/PID for this fault. Since we are assuming a fault, one of the two conditions will fail. If the page has been initially loaded (or modified), AOS/VS will obtain the CCB address of the page file from the PTBL, and the address within the file from the page file directory located in the process' ring 1. If the page has not been modified, AOS/VS will obtain the CCB address of the .PR file from the PEXTN, and calculate the address within the file from information found in the memory descriptor also located in the PEXTN. Since we now have a CCB, we must also have a FCB, and therefore the location on disk of the file. AOS/VS will allocate a page from GSMEM, and four blocks will be read from the appropriate file. The physical page number of the new page is put into the PTE, and the residency bit is turned on.

ACDG -- Logical fault, page is shared, part of a .PR file, on the LRU

or

ACDG -- Logical fault, page is shared, part of a data file, on the LRU

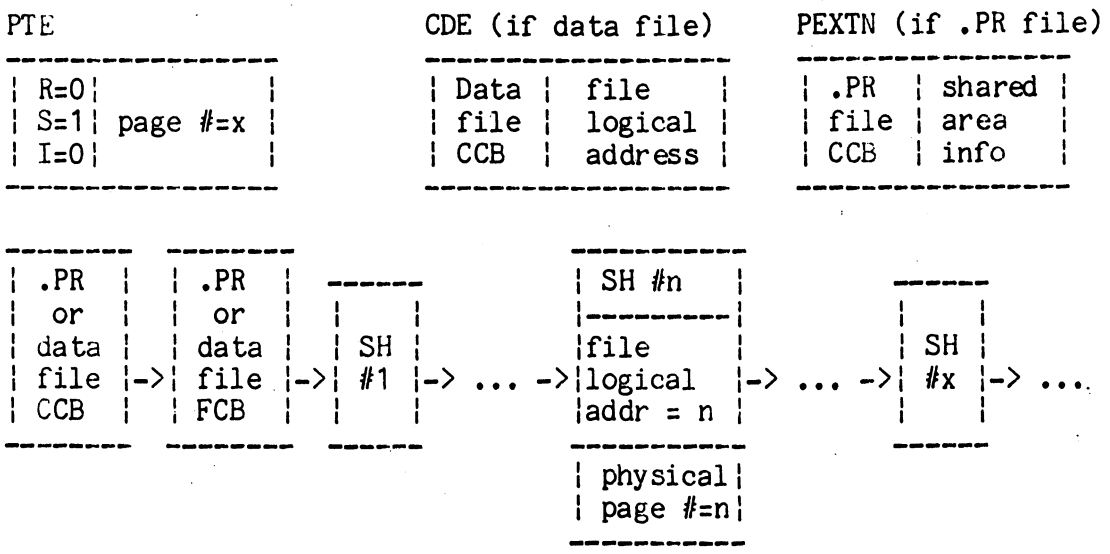


After identifying the page as a shared page, AOS/VS looks at the code/data bit in the PTE. If the bit indicates code (part of a .PR file), the address of the PR file's CCB is obtained from the PEXTN, and the file logical address is calculated from the shared memory descriptor also in the PEXTN. If the bit indicates data, the CCB address and file logical address is obtained from the correct CDE. Following the CCB pointer to the FCB, AOS/VS then searches down the FCB's SH chain until a match is found for the file logical address. The SH will also contain the physical memory page holding the correct shared page. Since the use count on the shared page is 0, the page must be on the LRU. Therefore, AOS/VS removes the CME from the LRU chain, puts the physical page number (from the SH) into the PTE, sets the residency bit in the PTE, and increments the use count in the CME.

ACDH -- Logical fault, page is shared, part of a .PR file, in use by someone else

or

ACDH -- Logical fault, page is shared, part of a data file, in use by someone else



CME for page n

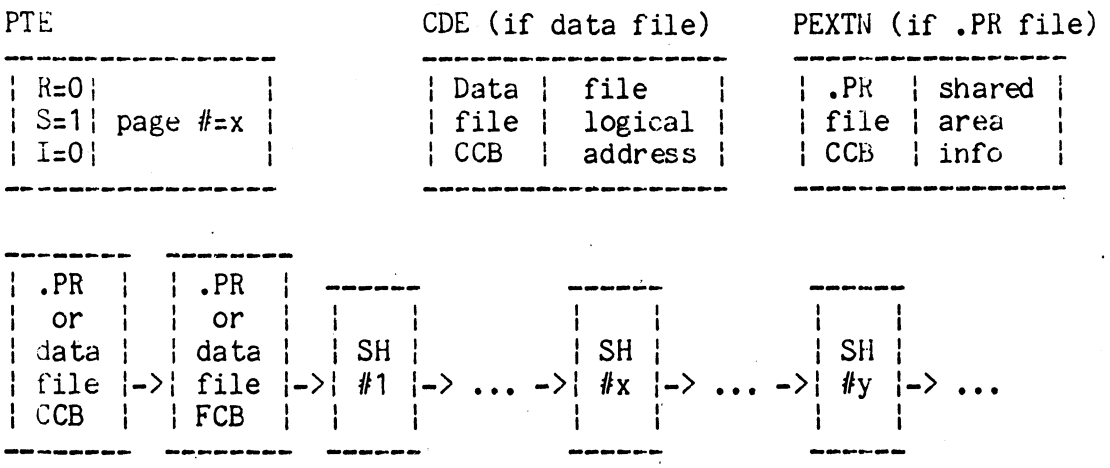
use cnt <>0

The logic for this path is the same as the one immediately above, except that the use cnt<>0 indicates that the page is not on the LRU, and therefore does not have to be removed from it. The use count will be incremented to indicate that one more process is using the page.

ACEI -- Physical fault, page is shared, part of a .PR file

or

ACEK -- Physical fault, page is shared, part of a data file



After identifying the request as shared, .PR file, the data/code bit is interogated. If the page is to come from a code file (.PR), the .PR's CCB address is obtained from the PEXTN, and the file logical address is calculate from the shared file information also found in the PEXTN. If the page is to come from a data file, the files CCB address and file logical address is obtained from the CDE.Since the page will not found on the appropriate FCB's SH chain, AOS/VS must read it from the disk. The address on the disk is calculated from the file location (in the FCB) and the logical file address in the CDE or PEXTN. A physical page is aquired from GSMEM, the four blocks are read into the page, and a shared header is built. The physical page number is put in the PTE, and the PTE's residency bit is set.

Time stamping

AOS/VS maintains a global counter RTIM.W which is incremented and assigned to each physical page as it faults in. The value is stored into the CMTIM.W offset of the physical pages's CME.

PFF (the page fault frequency algorithm)

The PFF flag which indicates PFF is on is PFFEN. It is turned on and off as time advances depending on the memory usage. If the machine gets into memory contention then PFF is turned on.

PFF and PFFUFT are two routines (found in PFF.SR) used to determine which pages, if any, should be removed from a processes working set. The routines will remove each selected page from the WS, and if it is an unshared page, or a shared page with a use count of 0, it will be put on the LRU. PFFUFT will, in addition, always update the process's time since last fault (PFTOT.W in the PEXTN), while PFF will only update PFTOT.W if the WS will be changed. Below is a general summary of the code path involved:

1. If PFF is disabled (it will not be turned on until the first call to PRPR for preemption) just return

2. Compare the current WS size with the minimum, and if they are equal, just return (we cannot release any pages)

3. Calculate the number of CPU ticks used by this process since it started:

   (320-time_slice_residue) + CPU time previous to this slice (PRUNH.W)

   If this is a PFFUFT call, store the result in PFTOT.W (number of ticks at last fault)

4. Calculate the number of ticks used since last fault:

   (results from step 3) - (old PFTOT.W)

   and if the value is less than CAPT (currently 2000) just return. CAPT, therefore, is the minumum number of ticks that a process can run before getting it WS adjusted.

5. Since we are going to adjust the WS, update PFTOT.W

6. Update the WS-mills integral (call KCINT)

7. Release all of the unreferenced pages until WSMIN is reached.

   For each unreferenced, unwired, unshared page:

   a. Clear the PTE resident bit
   b. Decrement the use count on the PTP that holds the PTE

c. Decrement the WS size (PWSET)
d. If the page has been modified, set the initially loaded bit (which will cause the page to flush to the page file, and be reread from it instead of from the .PR file)
e. Purge the ATU (PATU)
f. Release the page to the LRU
g. Update the WS and referenced arrays
h. If the page being released is a PT page, reset the resident bit and set the initially loaded bit in the 1st level PTE, and decrement the use count of the 1st level PTP.

For each unreferenced, shared page:

a. Scan down the page's LPA (pointed to by the page's CME), looking for a match on the PID. If no match, we are done.
b. If a match, release each logical page
c. Purge the ATU
d. If the page is wired, call UNWIRE (decrement the wire count)
e. If it is a shared data page (vs code), decrement the corresponding CDE use count
f. Decrement the working set size (PWSET)
g. Release the shared page
h. Release the LPA
i. Update the working set and referenced arrays
j. Go back to a (a PID can have more than one LPA for a given shared page)

Handling a hardware fault]

The ECLIPSE MV hardware, upon detecting a fault, will dispatch through ring 0 locations 30 and 31. In AOS/VS, this double word pointer contains the address PGFLT (in PGFLT.SR).

1. Read out and stop the PIT (the user could be faulting)

2. If we faulted on the system fault block (never should happen), or in the interrupt world, PANIC 14340.

3. If SYSIN = 1, then we faulted in the system (on a CB) So...

    a. If the hardware detected a page fault depth violation, and the process that this CB is running for does not have the privilege to change page table levels, PANIC 14340.

    b. If the address that caused the fault was not in ring 0, then we credit the user for indirectly causing the fault. First we call PFFUFT to adjust the working set, then we call FAULT (see below) to fault in the page, we clear the referenced bit array, and return control to the CB with a WDPOP.

    c. If the address that caused the fault was in ring 0, then the system faulted in attempting to access a pageable system page. Call ASPGTWS (see below) to add the system pageable page to the system working set, meter the fault (SFAULT.W), and return to the control block with a WDPOP

4. If SYSIN = 0, then we faulted in the user ... so ...

    a. Set SYSIN = 1 (we are now in the system)

    b. Save the faulting PC (from the context block of the VTCB) in the TCB
    c. Store away the PIT value into the PEXTN (at PSL)

    d. Set the task is faulting bit in the current task

    e. If we are at the end of a time slice (the PIT value was 0 at the time of the fault), jump to the timeslice end processing in the interrupt service code.

    f. Fetch the VTCB address, and store the faulting task's TSYS word in VTSTK.W in the VTCB.
    g. Put a pseudo system call value (253) in TSYS.W to indicate a fault. (The fault will run similar to a system call)
    h. Jump into the system call processing code ... which will:

1. Enqueue the TCB to the system call queue for the process
2. Allocate a CB (this could pend ... CBs are limited)
3. Enqueue the CB to the ELQUE to be found later by the scheduler
4. Give control to the CB which will jump to the second stage of the fault code (in FAULT.SR)

i. Check the type of fault. If it is a page depth fault (attempt to do a two level translation on a one level table), grow the number of levels if allowed (bit in PFLG3, else signal an error.
j. Copy the hardware referenced bits into the process's referenced array
k. Call PFFUFT
l. Call FAULT
m. Clear the referenced bit array
n. Restore the TSYS word from the VTSTK offset in the VTCB
o. Take the good return path back to the system call dismissal routine, which will:

1. Unpend the TCB
2. Dequeue the CB from the ELQUE and return it to the appropriate free CB queue.
3. Jump to the scheduler (which will find this process with at least the task that just faulted ready to run)

FAULT

1. Get the page table address for the address that caused the fault

2. Update the WS-mill integral, and add the page to the working set (call APGTWS below)

3. Time stamp the page (RTIM.W + 1 -> RTIM.W, RTIM.W -> CMTIM.W)

4. Return

Adding a page to a working set

*	ASPGTWS	Add a pageable system page to the system WS
*	APTTWS	Add a page table page to a WS
*	APGTWS	Add a data page to a user's WS

1. If the page is already in the working set (PTE resident bit is set) just return

2. If the page is already faulting in (PTE faulting bit is set), pend waiting for the faulting (key is PTE address), and when unpended, go to step 1.

3. Set the fault in progress bit.

4. Wire in the PTPs required to complete the fault.

5. If the requested page is shared, call READSH to read in the page, and goto step 10.

6. If the page is unshared and on the LRU:

a. If adding this page causes the process to exceed the working set maximum, remove one page.

b. Meter the global and logical fault counters (per PID)

c. Dequeue the page form the LRU.

d. Increment the removable page counter.

e. Goto step 10.

7. Read in the appropriate page:

If the page is a system page (ring 0) request, read in the system page via a call to READSY.
If the page is an initially loaded user page, read the page

from the page file (READPG).
If the page is a non-initially loaded .PR file page, read the page form the .PR file (READPR)
If the page is a non-initially loaded user page that was ?MEMIed into the address space, allocate a page with a call to PGMBLK
If the page is a PTP, build it (BUILDPT)

8. Set up the usage descriptor offset in the CME (logical addr + PID) (PID = 0 for the system)

9. Add the frame number (passed back by READSY, READPG, READPR, PGMBLK, or BUILDPT) into the PTE

10. Set the resident bit in the PTE, unpend anyone waiting for the page, and pass back the modified PTE in AC0

11. If not a system pageable page, increment PWSET, and set the corresponding bit in the process's WS array. If the WS change is to the currently mapped process, the ATU must be purged to implement the change

12. If a system page, time stamp the fault (from RTIM.W), and enqueue the page to UPSYS.W (at the end of the queue, UPSYS.W is a LRU). Increment OVCNT (number of system pageable pages in memory) and write protect the page

13. Return

Reading a page into memory

The read requests from the add page to WS code above can be of four types, READSH (read a shared page), READPG (read a page from the page file), READPR (read a page from the PR file), and READSY (read a system pageable page). All of the routines will have to request one page frame of physical memory, and therefore might have to preempt another process. Therefore these routines can pend. Note that these pages are often called overlays, and for ease of use, they will be here.

READSY -- read a system pageable page (module PRSUB.SR)

1. Calculate the overlay number:

   (logical_address-start_of_pageable_area)/1024.

2. Get a page frame (call GMBLK). If none are available, call SYSREL which will release an old overlay page and repeat this step.

3. Build a buffer header on the stack including the logical address of the overlay on the master LDU (overlay# *4)+OVFAH.W. OVFAH.W contains the logical address of the base of the overlay area on disk.

4. NQ the buffer header (NQBHR) and wait for completion (BWAIT-> PEND this is an overlay load, so we must have a CB, so we can pend)

. Return with the frame #

READSH -- read a shared page (module PRSUB.SR)

1. If the maximum working set will be exceeded by the new page, remove one page from the working set.

2. The shared page will be either a data page or code page:

   a. If the shared page is a code page (from a .PR file), get the .PR CCB for the requested ring, and calculate the disk block offset of the page in the shared area (using the ring's memory descriptor offsets PSPRST.W and PSHSH.W)

   b. If the shared page is a data page, fault in the page's CDE, increment the corresponding CDP's use count, and fault in and wire the user CCB page holding the data file's CCB (obtained from the CDE).

3. Get and initialize an LPA

4. Scan the FCB chain (pointed to by the CCB) to see if a shared header (SH) for the page is already in memory. If so, but it is locked (indicating I/O in progress), pend the current task using the SH address as the key.

5. If the page is not in memory:

   a. Allocate a shared header (from GSMEM) and initialize the disk block number (so it will be found by others wanting this page).

   b. Lock the shared page header (see 4 above).

   c. Enqueue the new shared header onto the FCB SH chain.

   d. Allocate a page of memory (PGMBLK)

   e. Enqueue a shared read request to the diskworld (from the appropriate file found in step 2, to the page allocated in step 5d).

   f. Unlock the shared header, and unpend anybody pended on this headers (see 4 above).

6. Increment the use count for the shared page (in the CME)

7. If this is the first copy of the shared page, increment PRMSHP (# of shared pages in the WS with a use count =1) for the faulting process. If this is the second copy, decrement PRMSHP for the process that requested the page first. (The first requesting process can be found by examing the LPA chain)

8. Enqueue the new LPA to the Shared page header's (SPH) LPA chain. (the chain is ordered by descending PID)

9. If we now have one process with multiple copies of the same shared page in the process's address space, set bit BPFMS to indicate so.

10. Increment the shared page count in the working set

11. Return

READPR -- read a page from the .PR file (module PRSUB.SR)

1. Get a page frame (PGMBLK)
2. Get the .PR file CCB address for the requesting ring (from PEXTN)
3. Convert the logical address requested into a disk block number

   (page # + preamble) * 4
4. Enqueue the request as an unshared read
5. Increment PRMUNP, the number of unshared, unwired pages
6. Return

READPG -- read a page from the process's page file (module PRSUB.SR)

1. Get a page frame (PGMBLK)
2. If this is a PTP, convert the address to a ring 1 address
3. Convert the logical address into a pagefile address
4. Enqueue the IO request as an unshared read
5. Increment PRMUNP, the number of unshared, unwired pages
6. Return

Note: PGMBLK will first scan down the process memory chain before looking at free general memory. GMBLK will just look at free general memory

Removing a page from a working set

There are two basic routines involved with removing pages from a working set. These are PSTEAL, which is called by the preemption code in the core manager, and RPGFWS, which is called from the SWAPOUT routine to shrink the current working set down to 124. pages. The code path is, for the most part, common between the two routines.

- Calculate the bounds of the workings set (lowest page and highest page in working set)

2. Examine each unreferenced page in the working set to see if it is the best candidate for removal, and if so, can be removed. The best candidate for removal is the page with the lowest CMTIM.W. (relative time last faulted) The page can be removed if:

    a. The page is not wired ... and ...

    b. The page is unshared ... or ...

    c. The page is shared with a use count of 1

3. If a page was found in step 2, call FREL to release it (if it is unshared, the page's PTE residency bit will be reset, the use count on the corresponding PTP will be decremented, the page modified bit in the PTE will be set if the LMRF says it has been, the page is release to the LRU, and the WS and referenced array are updated. If the page is shared data, the corresponding CDP's use count will be decremented, the page will be released (if last user it will be enqueued to the tail of the LRU and marked modified if so), the LPA will be released, and the referenced and WS arrays will be updated.

4. If the required number of pages have been released, take the good return

5. All pages have been checked, and we have not found the required number of unreferenced pages. Perform step 2 and 3 on pages in the working set that have been referenced.

6. If we have satisfied the request, take the good return. If not, put the number of pages removed in AC1, and take the bad return.

Swap and page file

Allocation

SFILE.W is a table of bytes that is indexed by PID and contains the file type and 'in-use' bit for each page file.

File type number	Element size	Page file type
1	32	1 mbyte (single level page table user including a 16 bit user)
2	64	10 mbyte (not used)
3	64	512 mbyte (double level page table user)

1. Identify the page file type

2. Scan the table looking for a SFILE.W entry not in use, and the PID number not reserved by the connection manager. When one is found, and it is of the type we are looking for, join the common code below. Otherwise, save the PID # in one of 4 temporary locations indexed by type (0 = free) for later. If a PID # for a type is already saved, just continue the scan

3. If no free match is found, then find a page file based on the following:

   If we need a 1 mbyte file:

   a) look first at any unused page file (PID # will be in STFREE.W) If a free page file is found, then create and open the file and join the common code below

   b) If no free page file is available, then look at 10 mbyte files If one is available, then delete it, and create and open a 1 mbyte file, and join the common code below

   c) If no 10 mbyte file is available, then look for a 512 mbyte free page file. If one exists, delete it, and create and open the new file, and join the common code below

   d) Return an error, no available PID

   If we need a 512 mbyte file, do as above but in the following order:

   a) unused 10 mbyte file

   b) unused page file

   c) unused 1 mbyte file

. (common code) We now have a page file, and a PID number. Create

(if necessary) and open the associated swap file (all swap files are the same size)

Deallocation

1. Close the swap file
2. Close the page file
3. Clear the in use bit in SFILE.W
4. Done

Swapping

The swap file

Each swap file is 128 pages long. There are four pages reserved for holding various queues of VCME (virtual core map entries) that are used to describe the CMEs they hold. Therefore, only 124. pages can be swapped out. As in AOS, only unshared pages will be swapped out.

There are four VCME queues in the swapfile. Each holds a specific type of VCME

SPQD.W	holds the shared page VCMEs
SBQD.W	holds the high level page table page VCMEs
SVQD.W	holds the low level page table page VCMEs
SWQD.W	holds the unshared data page VCMEs

The VCME for a page contains the wire count for that page. Pages that were wired at the time of the swapout will be enqueued to the front of the appropriate chain, while those that were not wired will be enqueued to the back.

SWAPOUT

1. If we are aborting an initial load, goto step 8
2. Meter the number of swapouts (SWPSO) and the number of times that this process has swapped out (PSWPSO in the PTBL)
3. If the WS is larger than size in usable swapfile size in PTBL Extender, release the overflow pages by calling RPGFWS
4. Set the swap in progress bit
5. Initialize the swap file queues

6. If there are any shared pages in the working set, build one VCME for each page and enqueue it to SPQD.W. Then release the page (decrement the use count and process if the count goes to 0)

7. If the process is terminating, release any unshared unwired pages.

8. Remove all unshared pages associated with this process from the LRU and enqueue the pages to the FC1024 chain.
(This step will flush any modified page to the page file before it is removed)

9. For each page in the working set, copy the CME into the appropriate type of VCME and enqueue the VCME to its proper swapfile queue.

10. Remap the process image into the swaparea (a contiguous logical area)

11. Call SWAPIO, requesting a write

12. If the Global swapout timestamp matches that found in the PEXTN, clear the global stamp.

13. Release the process's page frames (call RCORE)

14. Reset the swap in progress, one slice completed & enqueued to CM.

15. Dequeue the process table from the CMTSK swap queue.

16. Dequeue the process from the ELQUE, BLKQ, or MBLKQ and enqueue it to the appropriate ineligible queue

17. Unpend anyone waiting for this process to swapout

18. Return

SWAPIN

At the time the Core manager calls the routine SWAPIN, the following has been done:

GCORE has been called to allocate the needed memory pages and PTBL extender (PEXTN)

All of the needed pages, LPAs, and SPHs have been enqueued to the process's memory queue (PSMEMQ)

The WS and Reference bit arrays have been allocated

1. If the process is swapping in to complete a target system call, goto step 3

2. If the process has been preempted or blocked, abort the swapin (release the memory assigned, and enqueue the process to the appropriate ineligible queue)

3. Increment SWPSI (global number of swapins)

4. Dequeue the number of needed pages to complete the swapin request from PSMEMQ, set up the ring 0 swap file PTEs to point at the appropriate pages, and read the process image in from the appropriate swapfile.

5. Rebuild the process table's SBRs from the high level PTP VCMEs

6. Rebuild the hi level PTP to point at the low level PTP

7. Restore each CME based on the VCME enqueued to SVQD.W and update the ring 1 PTE that will point to the page (SVQD.W for high level PT)

8. Do the same with SBQD.W (SBQD.W for low level PTs)

9. Restore each CME based on the VCME enqueued to SWQD (unshared data page), and update the low level PTE that points to it

10. Fault (and wire based on the wire count) each VCME on the SPQD chain (shared pages)

11. Clear the PSWOTM.W (time of last swapout) time stamp.

12. If we are swapping in the process to complete a target system call, call UNPEND with the PTBL address as the key.

13. Dequeue the process table from the CMTSK swap queue.

14. Process any ?SIGNL and ?SIGWT requests aimed at this process. There is a double word bit map (PTUP.W) in the process table that will have a bit set for each TCB that must be unpended. (A check is made to make sure that the TCB is indeed pended on a ?WTSIG)

15. All done, return to the CMTSK main code path.

Miscellaneous

This section will discuss some of the AOS/VS routines and concepts that can not be classified as exclusively part of one module.

AOS/VS Timeslices

There are two types of timeslices under AOS/VS. The first, the subslice (Ss), is always 32 ms long and is defined by the interrupting of the PIT. The second, the user's timeslice (Ts), is defined as:

Ts = Ss * (2 ^ S)

where S (varying between 1 and 6) is determined by the interactiveness of the user.

Subslice (Ss) end (PIT interrupt)

At the end of a subslice (PIT interrupt), the following occurs:

1. Bump the number of subslices run since last made eligible (PSSEL in the PTBL)
2. Increment the user's CPU usage by -SUBSL (SUBSL is a negative number, so this will increase the CPU usage value)
3. If the user has resource limiting (max CPU time) check to see if it has been exceeded. If so set the appropriate bits to force termination (TIME LIMIT EXCEEDED).
4. Decrement the subslice count and put the PTBL in TSPTB.W If the count is not 0, then we are not at a full timeslice end, so set the high order bit of TSPTB.W (time slice end flag) to indicate PTBL shuffle only.
5. Reset the subslice (SUBSL -> PSL)
6. Save the user's task state.
7. Rescan the eligible queue (jump to the scheduler).
8. The scheduler will see that the flag (TSPTB.W) is set and will call TSPRC below.

TSPRC - Time- or Sub- slice end (as handled by the scheduler)

1. The scheduler checks TSPTB.W It will contain either 0 (no slice end, a PTBL (timeslice end), or 1SO + the PTBL (subslice end). We are concerned with the latter two possibilities: the scheduler will jump to TSPRC.
2. If this is a timeslice end (0S0):

    a. If the process is swappable then increment the exponent if not already max (6), calculate the new PNQF for the process, and calculate the number of subslices to make up the next full slice

    b. Put the new number of subslices that make up a full slice into the process table extender (constant for non-swappable processes)

c. Call PFF to adjust the process' working set.

d. If any processes are on IEQUE, IERES or IESWP, call CWAKE to wake up the core manager

3. Move this PTBL to the end of its priority group on ELQUE.

4. Jump back to the scheduler

The BIAS factors

The locations BIAS and HBIAS, in STABLE, define the AOS/VS bias value. The bias factor is used to balance the number of non-interactive vs interactive swappable processes on the ELQUE. BIAS contains the minumum number of non-interactive processes that AOS/VS attempts to keep on ELQUE, while HBIAS represents the maximum number. A non-interactive process in the AOS/VS sense is a swappable swappable having a time slice exponent of 6.

The following system modules reference the bias values :

SSOV5 - Implements the ?GBIAS / ?SBIAS calls.

COREM - The HBIAS factor is used in deciding which swappable processes can be marked for swapout. If the number of non-interactive processes on the ELQUE (ELNON)= HBIAS, then only a different non-interactive process can be preempted off of the ELQUE.
In the preemption code, PRPR, AOS/VS will not preempt a a non-interactive process if the current number of non-interactive processes equals BIAS.

CORM2 - When AOS/VS scans the IESWP queue for processes to swap in, and the minimum BIAS has not been met, it will not swap in an interactive process if any non-interactive processes have become unblocked, regardless of the fact that the non-interactive process might not fit.

Daemons

A daemon can be considered an AOS/VS initiated system call (as opposed to the user oriented or standard call). When AOS/VS needs something done, and the code path required might pend, AOS/VS will use a daemon for the processing.

Daemons are currently used for the following:

1. Process terminations (4 types: normal, trap, fatal error, ^C^B)

2. Process initial load. (The path can pend waiting for the disk)

3. Process a 16 bit process changing to/from resident (we will wire in a the pages of a resident 16 bit process)

4. Process keyboard interrupts (other than ^C^A)

Daemons are started by setting the request daemon bit in PSTAT, and run off of control blocks. They can be identified by examining offset CATCB.W (12) of the CB. It will contain a 0. The code for dispatching to the specific daemon code paths is located in SCHED.

The System Memory Key MKEY

MKEY is a location defined in the system upper page zero to keep track of the significant changes in available system memory (GSMEM). When a process has to wait for memory, the current value of MKEY is stored into its process table in offset PMKEY. Subsequent comparisons of PMKEY to MKEY allow the scheduler to decide if the memory situation is now such that the process could in fact run.

Besides STABLE, MKEY is referenced by the following modules :

MEMRY - Routines GSENQ and RFBLK ISZ'es MKEY when adding an area of GSMEM to a significant chain, i.e. when the element is added to an empty chain and all chains of larger size elements and the LRU are empty.

MEMR2 - Routine RCORE will ISZ MKEY independent of the state of the FC1024 chain or the LRU. RCORE release a processes memory on termination or swapin abort

SCHED - Routine PMWT, dispatched on by the process scheduled start up code (PENTR) when bit PSMWT is set, checks if PMKEY is different from MKEY. If not, we just run the next process on the ELQUE because no more memory is currently available.

SCPRC - Routine TERTN (bad return from control block processing) will restart the TCB request when we cannot get the memory resource to process it. Before releasing the control block the process is marked by setting bit PSMWT, so that it will not run before MKEY changes.

INTERPROCESS COMMUNICATIONS AND CONNECTIONS

IPC

The system modules which effectively perform the sending/ receiving of IPC messages are the modules IREC, ISEND and ISEN2. ISEN2 is chained to from ISEND in the event that spooling has to occur.

Additionally, some supporting IPC system calls are documented here: ?ILKUP, ?TPORT, and ?RSEND are coded in the module ISEN2.

IPC Initialization

IPC initialization for every new process (except CLIBT at system startup) is done during process creating in PROC2. The IPC spool file ":PROC:IPS.PID" is created if it does not already exist, and its CCB address is saved in the new process' process table. The spool file is then opened (unless it is in the page/swap/ipc pool) and its CCB locked. The bit map in block 0 is set up and the PTBL's spool file directory pointer (PSFDF/PSFDB) are initialized to -1. The CCB is then unlocked. The initial IPC messages (user and AGENT), if specified are sent.

Spool file directory chain

Each process has a spoolfile directory chain, which consists of spool file directory entries double linked off of PSFDF.W (forward) and PSFDB.W (backward). In addition, double offset PSFRC.W in the PTBL contains four single byte counters indicating how many messages are spooled for each of the 4 user rings (4-7). Each user can have up to 48. messages spooled to each ring. The chain is time ordered.

A spool file directory entry has the following format :

MEFD.W	chain forward link
MEBK.W	chain backward link
MESFL = ?ISFL	system flags
MEUFL = ?IUFL	user flags
MEOPH = ?IOPH	origin port number (hi)
MEOPL = ?IOPL	origin port number (low)
MEDPN = ?IDPN	destination port number
MELTH = ?ILTH	message length (in words)
MEPTR.W = ?IPTR	word pointer to message text in buffer (user buffer before the call, spool file buffer after spooling)
MHNAM	hostname

The length of a spool file directory entry is 13. words long, and are allocated from GSMEM on demand.

The IPC Spool File

The IPC spool file is organized as follows :

BLOCK	WORDS	USE
0	0 - 255.	Bit map
1	0 - 255.	Bit map
2	0 - 255.	Buffer
.		.
.		.
.		.
127..	0 - 255.	Buffer

Spool file bit map

1 bit = 16. word nugget
1 word = 256. word = full disk block

Allocation is from left to right in a word. A set bit means the corresponding nugget is free. ISEN2 will zero the appropriate bits when it allocates space to hold a message in the spool file.

The bit map for the spool file is initialized as follows by PROC2 at process creation time :

BITMAP WORD	CONTENTS	BLOCK ALLOCATION
0	0	bit map
1	0	
.	-1	free
.	-1	.
.	-1	.
127.	-1	free

Spooling an IPC message

When a message is spooled to a user ring (4-7) the appropriate counter is checked to insure that the per ring maximum is not exceeded. Then:

If there is a message associated with the ?ISEND:

a. The counter SMLLIPC.W is incremented (global number of headers with messages)
b. The counter GLOBIPC.W is incremented (global number of headers)
c. The spool file bit map (block 0) is read in
d. Space is found for the message
e. The message is written into the file
f. The bit map is updated.
g. An IPC spool file directory entry is allocated, filled in, and enqueued to the end of the PTBL's queue.
h. The per ring counter is incremented by one

If there is no message associated with the ?ISEND:

a. The counter ZEROIPC.W is incremented (global number of headers without messages)
b. The counter GLOBIPC.W is incremented (global number of headers)
c. An IPC spool file directory entry is allocated, filled in, and enqueued to the end of the PTBL's queue.
d. The per ring counter is incremented by one

Outstanding receive entries

When a process issues ?IREC and the corresponding ?ISEND cannot be found (but the origin PID does exist), then an ORR (Outstanding Receive Request) entry must be created. The ORR is allocated out of GSMEM, and filled in with the data found in the ?IREC packet. Then the ORR is linked to the end of the receiver's ORR chain. The chain originates at offset PIORR.W in the receiver's process table. The chain is terminated by a 0. To prevent race conditions, when the chain is being accessed, the IPC lock bit (found in the IPC spool file CCB) is set.

Format of an outstanding receive entry:

OLNK.W	link word
OBLNK.W	backward link word
OOPH	origin port number high (HID/PID)
OOPL	origin port number low (16 bit port #)
ODPN	destination port number
OLTH	buffer length (in words)
OADDR.W	pointer to user buffer
OBUFH.W	pointer to user header
OTCB.W	user TCB address
OSFL	system flag word
OHNAM	hostname

ORR entries are 16. words long and are allocated out of GSMEM when needed.

?ISEND

Entry points in ISEND and ISEN2 :

ISEND.P entry point for user system call

IS.R ISEND entry for ?IS.R and connection support. The privilege checks are skipped.

SIPC.P entry point for a system send

IIPC.P send an initial IPC message; this is used by PROC2 to send a process its initial IPC message.

SENDR.P implements ?RSEND call

ISEN2.P logic to spool an IPC message.

ISEND logic

First, the caller's IPC privilege is checked. The caller can send the message if:

1. The call is from the AGENT ring ... or ...
2. The caller has IPC privileges
3. The caller is a server of the target PID/ring

The destination local port and the receiver's PID are checked for validity and the full destination and origin ports are set up. At this point the receiver's spool file CCB is locked; if it was already locked we pend waiting for its release.

The receiver's chain of ORRs is then scanned to see if an exact match of ports can be found. If not, then the receive from all case is examined. If a match is found, the ORR is unlinked from the chain, and its memory released to GSMEM; the IPC message is then copied to the receiver using the MBTU code. If no match is found, either an error is returned(if so requested), or the message is spooled to the target process.

In the case where the sender or the receiver is the system, the ISEND logic is basically the same, however validity checks are simplified.

ISEN2 logic

Same as above

?IREC and ?IS.R

Module entry points :

IREC entry point for user system call

IS.REC receive part of is.r

IREC logic

First, the destination port is validated and the message buffer in the receiver's address space is examined to make sure the first and last pages are valid (we check the whole buffer area later).

If there is an entry on the receiver's spool file directory chain, the chain is scanned looking for a port match. If found, any message is moved into the user's address space, and the bit map updated. Then the per ring message count is decremented if the ring field is 4 - 7. Finally, the spoolfile directory entry is dequeued from the PTBL chain.

If no match is found but the sender's PID does exist, a ORR is built out of GSMEM space and linked onto the receiver's chain of ORRs hanging off its process table.

IS.R logic

This system call performs an IPC send which, if successful, is followed by a receive with the same ports. The packet is a regular ?ISEND packet followed by two words specifying the length and location of the receive buffer. The call is non-direct and executes entirely in the ISEND and IREC overlays.

First a ORR is allocated, so that if the call is restarted, the IPC message does not get sent more than once. The packet is then moved to the stack, the caller's IPC privilege is checked and the send issued via entry JSEND, followed right away by the receive. Any error condition will go through a path which releases the memory acquired for the ORR.

?ILKUP, ?TPORT, ?RSEND, ?GCPN

Module entry points (in ISEN2)

ILKUP.P : user entry for ?ILKUP

SLKUP.P : system entry for ?ILKUP, name in user space

VLKUP.P : " " " " " " system "

TPORT.P : user entry for ?TPORT

RPORT.P : networking ?TPORT

RSEND.P : " ?ISEND

GNCP.P : get console port number (in SSOV5)

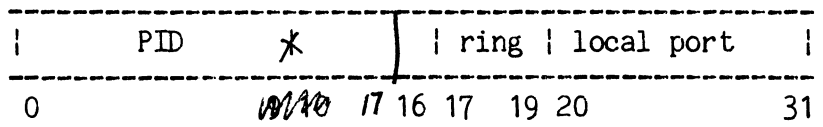
ILKUP (Port look up logic)

ILKUP takes a pathname, resolves it and checks for an IPC type entry with that name. If the file is of the correct type, the port information is obtained from the file's FIB and passed back to the caller TCB. If the file is not of type IPC (or does not exist), an error is returned.

RPORT / TPORT (Port translation logic)

For RPORT processing, the caller specifies a PID. For TPORT, the PID is that of the callers.

RPORT and TPORT take as input a local port number, ring number and PID and return the global port number assigned to it. The internal global port format is:



?GCPN logic

The entry point for this call, GNCP.P, is defined in module SSOV5. This call takes a PID or a process name and returns the port number of that process' console, or an error if the process has no console. After the checks are performed (PID exists), the process table is retrieved and the process console port number in offset PCONH.W is returned.

Databases offsets definitions

Often, the same information is present in several IPC databases, referenced by different offset names. To assist one looking at the system modules, the following charts illustrate which offsets correspond.

For example, a user who issues ?ISEND, has a packet with an offset of ?IOPN. In ISEND this is placed on the stack at offset 'OPN'. After port conversion, this word on the stack is referenced as 'DPN'. If this entry is spooled, this same word is stored at offset 'MEDPN' of the spool file entry.

A user who issues ?IREC, has a packet with an offset of ?ISFL. This is placed on the stack by IREC at offset 'SFLAG', and if necessary, put in an outstanding receive entry at offset 'OSFL'. If this entry is accessed by ISEND, it will be placed on that stack as 'RSFL'.

ISEND definitions

Header	Stack	Converted	Spool entry
?ISFL	SFLAG		MESFL
?IUFL	UFLAG		MEUFL
?IDPH	DPH		
		DPN	MEDPN
?IDPL	DPL		
		OPL	MEOPL
?IOPN	OPN		
		OPH	MEOPH
?ILTH	LTH		MELTH
?IPTR	MESA.W		MEPTR.W

IREC definitions

Header/user	Stack	ORR entry	ISEND stack
?ISFL	SFLAG	OSFL	RSFL
?IUFL	UFLAG		
?IOPH	OPH		
?IOPL	OPL	OOPN	
?IDPN	DPN	ODPN	

?ILTH	LTH	OLTH	
?IPTR	BUFA.W	OADDR.W	MESA.W
AC2	UAC2.W	OBUFH.W	UBUFH.W

The IPC ports

The IPC port numbers format is defined as follows :

Sender's packet:

DPH	destination port number high
DPL	destination port number low
OPN	origin port number

Receiver's packet:

OPH	origin port number high
OPL	origin port number low
DPN	destination port

Port 0 is reserved. It is often used by ISEND/SIPC. Ports 1-7777 are available for user ports. Ports 10000-17777 are AGENT ports. The PMGR uses ports 1-17777.

A user can receive on port 0 and match any user port. AGENT cannot receive on port 0. The PMGR can receive on port 0 and match any port.

The ports passed by the user are converted into system wide global ports. These converted port numbers are moved to the spool file or outstanding receive chain as necessary. Comparisons are made between two sets of converted port numbers to see if ports match.

Initial IPC and termination IPC

If ?PIPC <> -1 in the caller's ?PROC packet, then there is an initial message to be sent (via ISEND/IIPC) so that a "Receive from me" will receive it. Its IPC header address is in ?PIPC and the father establishes the format of this message. When CLI is the father, the message contains an edited version of the original CLI command, the command tree, and CLI file information, and the user flag is set to 1B0.

System calls ?RETURN and ?TERM effect process terminations. In either case, a message is sent to the father process. The default IPC header layout is :

?ISFL	0
?IUFL	term code/PID
?IDPH	0/dad's PID
?IDPL	0
?IOPN	?SPTM
?ILTH	0

```
?IPTR   0
```

Optionally, this default header can be overridden by ?RETURN or ?TERM. See the "AOS/VS Programmer's Reference Manual" for details on these system calls. Details on the format of the termination message can be found there too.

While AOS/VS is processing a termination, the system module PRCNG invokes ISEND/SIPC to send the termination message to the father process. However, this is not sent if the father is the root process or is also terminating.

AGENT IPC

When a user's AGENT is initialized, a port is assigned for each of his tasks. Tasks 1,2,3,etc... are assigned ports 10000,10001, 10002, etc... One use of these ports is for communication with the console while the debugger is running. (The debugger runs in the ghost.)

When the AGENT processes a ?PROC, it has the option of sending an AGENT-to-AGENT message on port 10000. If AC0<>0, then AC1 points to a message which is sent during process creation, invoking SIPC in ISEND. The message consists of a list of the generic file names, so that the new ghost will be able to open generic files.

Format of the AGENT-to-AGENT header :

```
?ISFL   0
?IUFL   0
?IDPH   0/PID
?IDPL   200
?IOPN   0
?ILTH   AC0
?IPTR   AC1
```

IPC locking

Whenever a code path accesses the IPC spool file directory chain, the spool file or the ORR chain of a process, that processes spool file CCB is locked. This prevents some race conditions.

The Connection Manager

Connection management system calls are implemented in the the following modules:

In MISC1:

SERVE.P user entry for becoming a server

In NET1:

RESIGN.P user entry for resigning as a server

TBC.P system entry for breaking a connection on a TERM or a CHAIN

DRCONX.P user entry for breaking a connection

VCNCT.P entry to verify a connection (used exclusively by and defined in IREC

In CONX2:

CONX.P user entry for becoming a customer of a specified server

ICNCT.P system entry for establishing a connection

PRCONX.P user entry for passing a connection from one server to another

In CONX:

MBTC.P user entry for moving bytes to customer

MBFC.P user entry for moving bytes from customer

MBTU.P user entry for moving bytes to a user

MBFU.P user entry for moving bytes from a user

The Connection Chain

The Connection Manager maintains a connection chain consisting of one 9. word database per connection. These databases are found in dedicated pages of GSMEM which are in linked via their CMEs. Each page can hold 113. connections. The first CME is pointed to by an 11. word header which is pointed to by CNXTB.W in page 0. The first connection page, and the chain header is allocated at the time of the first connection, and subsequent pages are allocated when needed, and linked to the front of the chain.

Below is the format of the per connection database:

CXFWL.W	forward link
CXBKL.W	backward link
CXCPID	customer PID
CXSPRNG	server PID/RING
CXCPRNG	customer PID/RING
CXSPID	server PID
CXSTS	status information

Five status bits are presently defined :

CXBMC = 1B0	connection broken by customer
CXBOB = 1B1	do not send an obituary message to customer
CXBPF = 1B2	not currently used
CXBOB = 1B3	send SIGNL instead of obit IPC on break
CXBMS = 1B15	connection broken by server

SERVE / RESIGN logic

SERVE simply sets the appropriate bit in the server ring bit map word (PSRNG) in the PTBL.

RESIGN checks if the caller is a server, turns off the bit in PSRNG in the caller's process table and sets bit CXBMS in the status word of every entry with this server's PID found in the connection table. If the obituary flag is not set for this particular connection, an IPC header is built using ?SPTM as origin port and 0 as local destination port, and an IPC termination message is sent via SIPC.P (remember, the SIPC.P entry skips privilege checks).

CONX / ICNCT logic

CONX establishes a connection as requested by a user, and ICNCT establishes a connection requested by an internal code path. The logic for both calls is similar, with the CONX logic doing a bit more error detection.

CONX will first format the PID/RING entries specified in the system call packet and verify that the target process is indeed a server. CONX will then search the connection chain (or allocate the chain if this is the very first system wide connection). If the connection is found, CONX will do one of the following:

If the connection was broken by the customer, return the 'connection broken' error condition
If the connection was broken by the server, reestablish the connection.
If the connection was not broken, update the user defined flags based on the caller's ACs.

If the connection is not found, a new entry will be added to the connection chain (a new page will be allocated if necessary).

DRCONX logic

If the connection table is not defined, return an error.Otherwise, DCONX builds a couplet using caller's PID and target PID, and then scans the connection table looking for a match on the couplet. If the entry does not exist the couplet is reversed and the table is scanned again. When the entry is found, we set the appropriate "Broken Connection" status bit, and if the result of the call is that both bits CXBPC and CXBPS are set, the

entry is cleared. If the result is that only one of the bits is set, a break message is sent to the other party; this independently of the state of flag CXBOB.The IPC header for the break message is as described above

PRCONX logic

PRCONX fetches the target process table and checks the server bit. The caller's PID is compared to the target PID as they must be different. The connection table is then scanned looking for a match on the couplet of the former customer's PID/caller's PID. If The connection is found and neither of the "Broken Connection" status bits are set, the new server's PID is stored into the right byte of the entry.

VCNCT logic

This entry simply verifies the existence of a connection by scanning the connection table looking for the correct couplet, and if found checking the status bits to verify that the connection is not broken.

TBC logic

TBC is an internal entry point called from overlay CLNUP on a process termination or chain.

If the connection table does not exist, we are done. If it does, scan down it looking for a couplet containing the terminating process' PID. For every couplet where this PID is found as a server, the "Connection broken by server" bit is set and an obituary message sent to the customer according to the state of CXBOB. For every couplet where the PID is found as a customer, the "Connection broken by customer" bit is set and an obituary message is always sent to the server.

CHAPTER 4 -- AGENT OVERVIEW
(AOS/VS Revision 5.00)

The Agent is the interface between the user and the rest of the Operating System. Every system call made by the user is processed through the Agent. The Agent may actually process the call, or preprocess the call and pass it onto the Kernel for further processing. On returning from the Kernel, the Agent may or may not post-process the call. Since the Agent is the outer most layer of the operating system, and the layer the users interface with, users see the Agent as being "the" operating system.

The Agent performs eight major functions:

* Dispatching of all system calls.
* Provides an environment for application oriented system calls within a top-down structured operating system.
* Validation of user supplied parameters
* Extended operating system support
* System call deflection for Resource Manager Agent
* System call deflection for EXEC
* System call deflection for GSMGR
* System call deflection for PMGR

Agent Data Bases

Before discussing how the Agent works, it is necessary to understand the structure of the Agent data bases. At this point the data bases will be listed, and their structure shown. The use of each data base will be described later in the manual when we deal with the part of the of the Agent that uses the data base. This section's main purpose is to group the data bases in one area, so they will be easy to find for reference. As you read further you will can refer to this section to remind you of the structure of each data base.

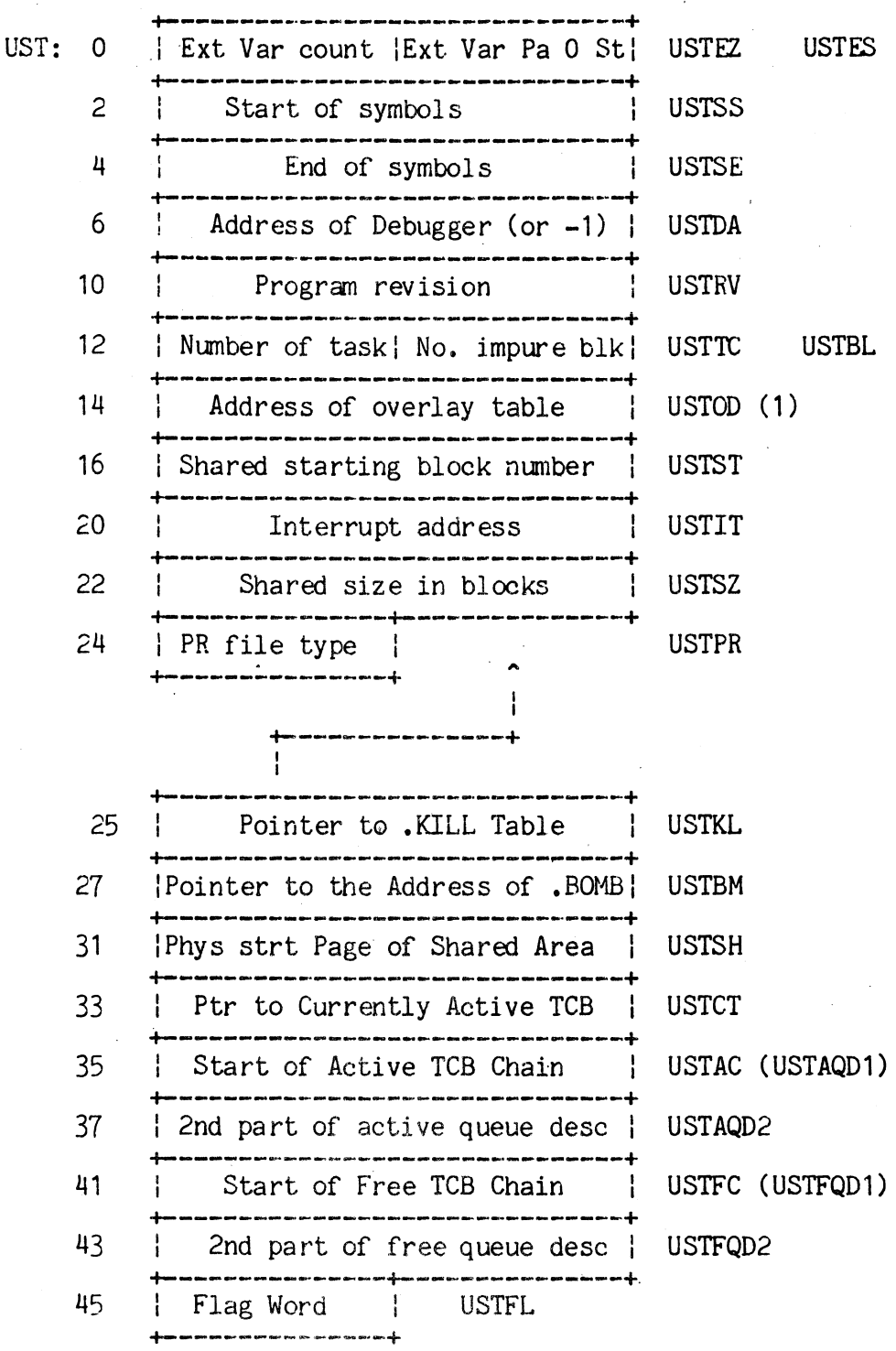
The following is a list of the Agent data bases. Each one will be described in more detail later.

* User Status Table (UST)
* Task Control Blocks (TCBs)
* Task Control Block Extenders (TCBXes)
* Channel Table and Channel Descriptor Tables (CDTs)
* Channel Maximization Table (CMT)
* System Call Ring Buffer
* Memory Data Base (Rings 3 - 7)
* Generic File Message Buffer
* Overlay Descriptor Table
* Agent Heap Memory (Unshared and Shared)
* Agent Stacks

User Status Table - UST

The UST is a per-process data base kept in the Agent. Currently, the UST starts at location 400 in the Agent's ring and is USTEN words long. The UST is created by LINK and contains information pertaining to the whole process.

The following table shows the layout of the UST.



The following bits have been defined in the UST flag word USTFL:

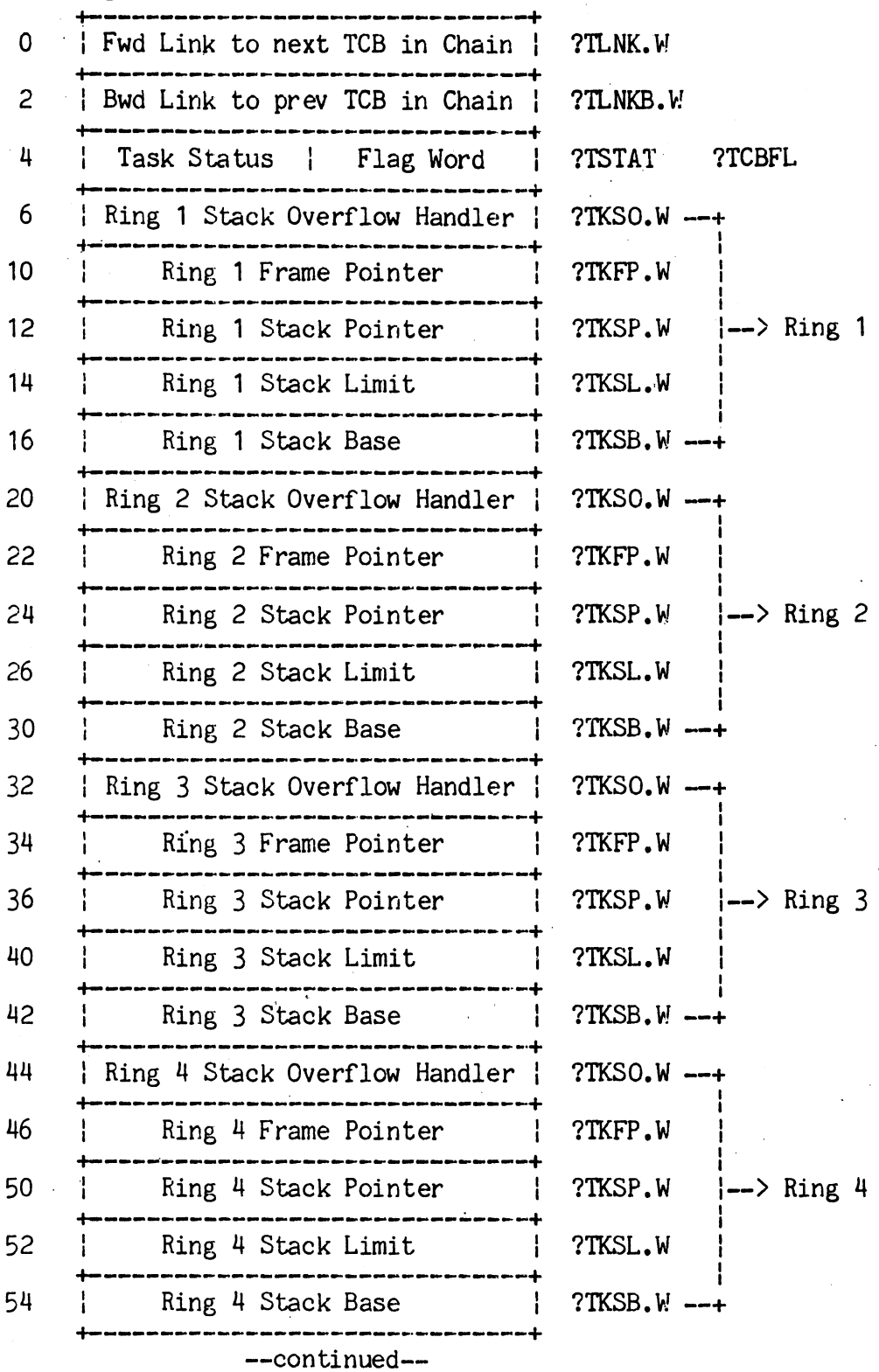
?UFDR - bit 2 - inhibit scheduling
?UFDB - bit 3 - Process is being debugged
?UFPH - bit 5 - Scheduling explicitly inhibited by Agent

?BUDB is the bit offset from the start of the UST to ?UFDB.
?BUPH is the bit offset from the start of the UST to ?UFPH.

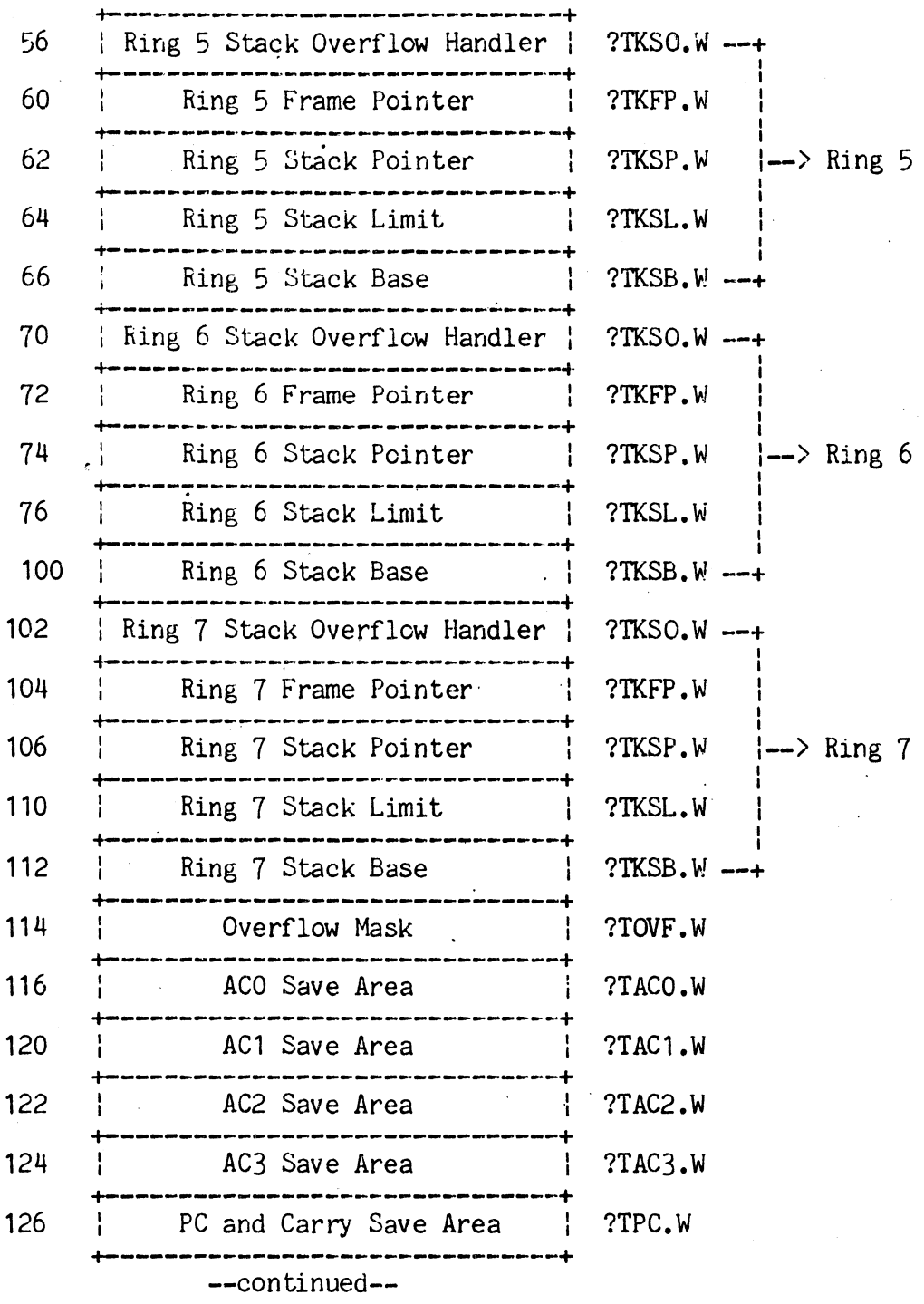
Task Control Blocks - TCBs

Unlike the UST, the TCBs are a per-task data base. There is a TCB for each task within a process. TCBs contain information, needed by the Agent and Kernel, about a particular task. This information varies for each task, unlike information in the UST, and thus must be kept in a per-task data base.

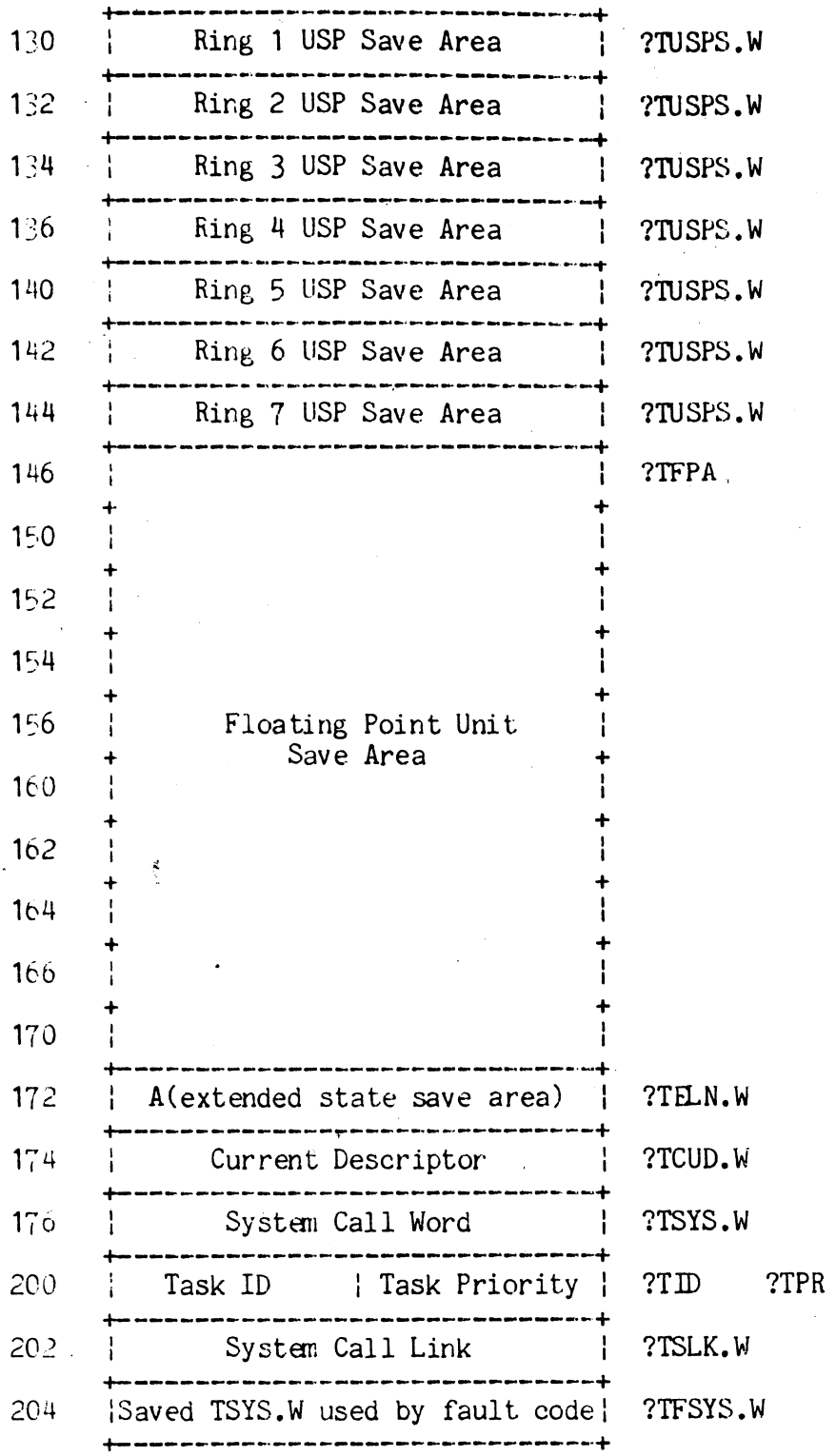
The following table shows the structure of the TCBs.



--continued--


56
Ring 5 Stack Overflow Handler
?TKSO.W
60
Ring 5 Frame Pointer
?TKFP.W
62
Ring 5 Stack Pointer
?TKSP.W
--> Ring 5
64
Ring 5 Stack Limit
?TKSL.W
66
Ring 5 Stack Base
?TKSB.W
70
Ring 6 Stack Overflow Handler
?TKSO.W
72
Ring 6 Frame Pointer
?TKFP.W
74
Ring 6 Stack Pointer
?TKSP.W
--> Ring 6
76
Ring 6 Stack Limit
?TKSL.W
100
Ring 6 Stack Base
?TKSB.W
102
Ring 7 Stack Overflow Handler
?TKSO.W
104
Ring 7 Frame Pointer
?TKFP.W
106
Ring 7 Stack Pointer
?TKSP.W
--> Ring 7
110
Ring 7 Stack Limit
?TKSL.W
112
Ring 7 Stack Base
?TKSB.W
114
Overflow Mask
?TOVF.W
116
AC0 Save Area
?TAC0.W
120
AC1 Save Area
?TAC1.W
122
AC2 Save Area
?TAC2.W
124
AC3 Save Area
?TAC3.W
126
PC and Carry Save Area
?TPC.W

--continued--


130
Ring 1 USP Save Area
?TUSPS.W
132
Ring 2 USP Save Area
?TUSPS.W
134
Ring 3 USP Save Area
?TUSPS.W
136
Ring 4 USP Save Area
?TUSPS.W
140
Ring 5 USP Save Area
?TUSPS.W
142
Ring 6 USP Save Area
?TUSPS.W
144
Ring 7 USP Save Area
?TUSPS.W
146
?TFPA
150
152
154
156
Floating Point Unit
Save Area
160
162
164
166
170
172
A(extended state save area)
?TELN.W
174
Current Descriptor
?TCUD.W
176
System Call Word
?TSYS.W
200
Task ID
Task Priority
?TID
?TPR
202
System Call Link
?TSLK.W
204
Saved TSYS.W used by fault code
?TFSYS.W

The following bits have been defined in the TCB status word ?TSTAT:

?TSPN - bit 0 - Task pended (general pend bit)
?TSSG - bit 1 - Task pended on ?XMTW/?REC
?TSSP - bit 2 - Task suspended
?TSRC - bit 3 - Task waiting for ?TRCON message
?TSOV - bit 4 - Task waiting for overlay
?TSWP - bit 5 - Task is faulting
?TSGS - bit 6 - Task pended on an Agent lock
?TSAB - bit 7 - Task pended awaiting ?GABORT
?TSTL - bit 8 - Task pended awaiting ?TUNLOCK from another task
?TSYG - bit 9 - Task has been ?SIGNALLED
?TSDR - bit 10 - Task pended by ?DRSCH (user disabled rescheduling)
?TSXR - bit 12 - Task pended on ?XMT or ?REC
?TWSG - bit 13 - Task pended on a ?WTSIG/?SIGWT
?TSUT - bit 14 - Task is executing ?UTSK code
?TSUK - bit 15 - Task is executing ?UKIL code

?BTPN is the bit offset from the start of the TCB to ?TSPN
?BTSG is the bit offset from the start of the TCB to ?TSSG
?BTSP is the bit offset from the start of the TCB to ?TSSP
?BTRC is the bit offset from the start of the TCB to ?TSRC
?BTOV is the bit offset from the start of the TCB to ?TSOV
?BTWP is the bit offset from the start of the TCB to ?TSWP
?BTGS is the bit offset from the start of the TCB to ?TSGS
?BTAB is the bit offset from the start of the TCB to ?TSAB
?BSTL is the bit offset from the start of the TCB to ?TSTL
?BSYG is the bit offset from the start of the TCB to ?TSTG
?BTDR is the bit offset from the start of the TCB to ?TSDR
?BSXR is the bit offset from the start of the TCB to ?TSXR
?BWSG is the bit offset from the start of the TCB to ?TWSG
?BSUT is the bit offset from the start of the TCB to ?TSUT
?BSUK is the bit offset from the start of the TCB to ?TSUK

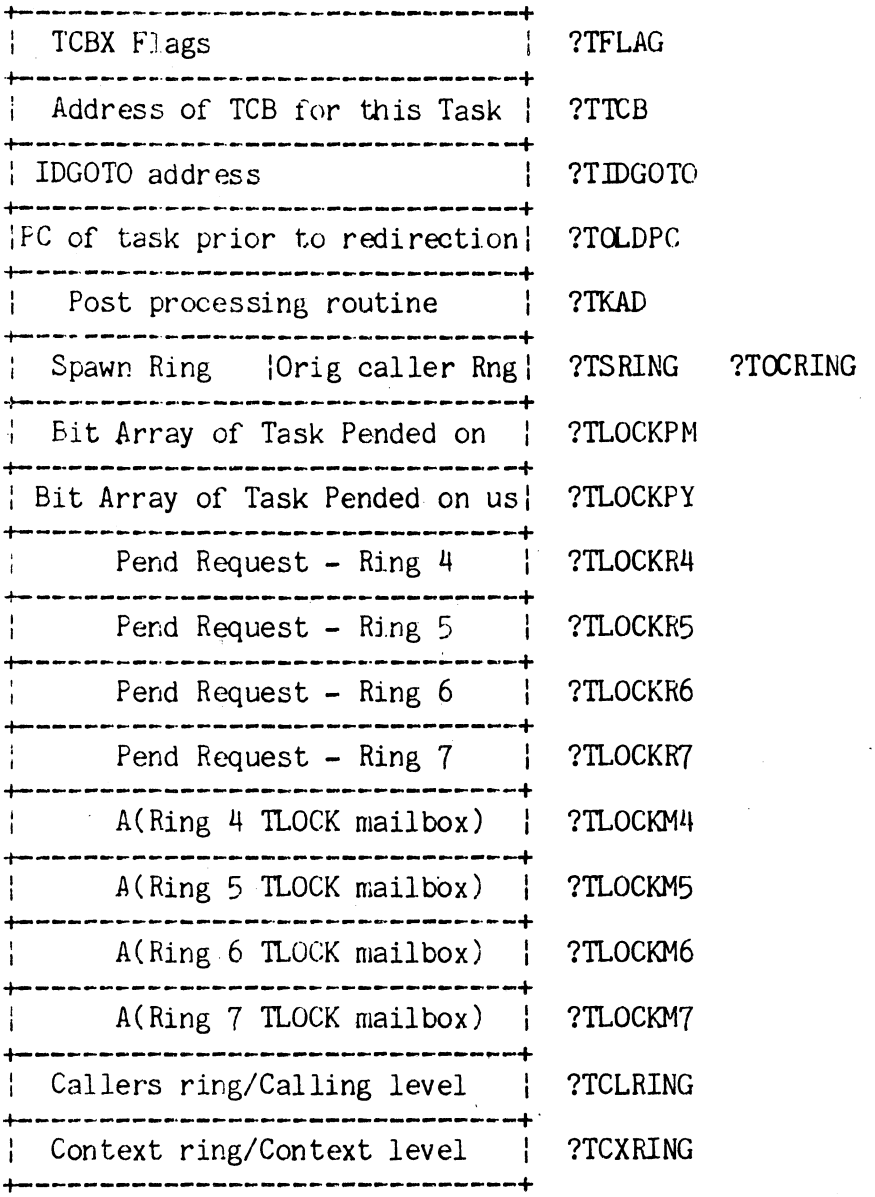
The following bits have been defined in the TCB flag word ?TCBFL:

?NREMSK - bits 0 - 2 - Mask for next ring of execution on a hard page fault
?TCBFLAL - bit 3 - Agent can skip ?ALLOCATE on ?SPAGE
?TCBFLDF - bit 4 - Task taking a depth fault
?UTIDMSK - bits 8 - 15 - Mask for the unique task ID of the task

TCBX Database: the Task Control Block Extender

The ?TLOCK implementation requires a fairly large area for variables. If the TCB were expanded to include these new variables, it would have been necessary to rebuild TCBs at Agent initialization time. Further, the scheduler requires that the TCBs be in a resident memory; the extended size of TCBs would cause a performance hit, especially in small memory configurations. Rather than expand the TCBs, it was decided to create a new database to give a home for these and other per-task variables. This database is called the Task Control Block Extender, or TCBX. The space for the TCBX is allocated during Agent initialization. A pointer to the base of the structure is kept in page zero of the Agent. Currently, the information in this database is used by the Agent only. The TCBX has one entry for each TCB for which the program was linked. TCBX elements are indexed by decrementing the UID (since UID is one-relative) and multiplying it by the length of the TCBX element; this forms the offset from the base of the table to the TCBX element for the specified task. Macros in PARSA are recommen- ded for conversions between any of the address of the TCB or TCBX, and the UID.

The TCBX table contains one element for each TCB. Each element has the following layout:



The following bits have been defined in the TCBX flag word ?TFLAG:

?TFPMGRB - bit 0 - Task is processing a PMGR request
?TABTB - bit 1 - Task is being aborted
?TUIDCB - bit 2 - ?UIDCALL in progress
?TMYR4B - bit 28 - ?TMYRING for ring 4 ?TLOCK
?TMYR5B - bit 29 - ?TMYRING for ring 5 ?TLOCK
?TMYR6B - bit 30 - ?TMYRING for ring 6 ?TLOCK
?TMYR7B - bit 31 - ?TMYRING for ring 7 ?TLOCK

?TFPMGRP is the bit offset from the start of the TCBX to ?TFPMGRB
?TABTP is the bit offset from the start of the TCBX to ?TABTB
?TUIDCP is the bit offset from the start of the TCBX to ?TUIDCB
?TMYR4P is the bit offset from the start of the TCBX to ?TMYR4B
?TMYR5P is the bit offset from the start of the TCBX to ?TMYR5B
?TMYR6P is the bit offset from the start of the TCBX to ?TMYR6B
?TMYR7P is the bit offset from the start of the TCBX to ?TMYR7B

?TFPMGRM is the bit mask for ?TFPMGRB
?TABTM is the bit mask for ?TABTB
?TUIDCM is the bit mask for ?TUIDCB
?TMYR4M is the bit mask for ?TMYR4B
?TMYR5M is the bit mask for ?TMYR5B
?TMYR6M is the bit mask for ?TMYR6B
?TMYR7M is the bit mask for ?TMYR7B

The bits ?TMYR<4,5,6,7> are cleared at task initialization time (and during Agent initialization, for the initial task's TCBX only), and are set when a ?TLOCK call is issued from a given ring with the flag ?TMYRING specified in AC0. They indicate that protec- tion is in effect from redirections from the current ring in addition to the higher rings.

?TSRING and ?TOCRING

The ?TSRING contains the ring of the initial PC which was specified when the task was created, the "starting ring".

The ?TOCRING word contains the ring of the original caller of the current system call. This ring is placed in ?TOCRING during Agent dispatching of calls which came from OUTER rings (ie from outside of the Agent).

Both of these ring numbers are integers, and will typically be in the range of 4 to 7.

?TLOCKM<4,5,6,7>: Mailbox Pointers

?TLOCK allows the user to specify the address of a doubleword mailbox which can be polled to determine if a task has attempted to redirect it. It is possible for each task to specify its own mailbox in each user ring; therefore, the TCBX has four doubleword pointers, one each for rings 4 thru 7. These pointers are named ?TLOCKM<4,5,6,7>.

At any given time, each mailbox pointer has one of the following values:

<zero> The ring corresponding to this mailbox pointer is not currently protected.

<(-1)> The ring corresponding to this mailbox pointer is protected; however, there is no mailbox.

<addr> The ring corresponding to this mailbox pointer is protected, <addr> gives address of mailbox.

?TLOCKPM: Bitmap of Tasks We Are Pended On

The ?TLOCKPM ("Pend Me") doubleword of the TCBX is used as a bitmap to indicate which tasks we are waiting for; these tasks must all issue a ?TUNLOCK before the current task is unpended. If any of these bits are on, the TLOCK suspend flag (?TSTL bit of word ?TSTAT of the TCB) is also set. At task initialization time, the "Pend Me" bitmap is zeroed. When a task issues one of the redirection calls, a bit is set for each task that it tried to redirect but was protected, and it sleeps if this word becomes nonzero in the process of attempting redirection. As each task issues its ?TUNLOCK, it clears the bit in our doubleword corresponding to that task, and wakes up the redirector if the doubleword becomes zero.

?TLOCKPY: Bitmap of Tasks that are Pended on Us

The ?TLOCKPY ("Pend You") doubleword is used as a bitmap of tasks which have attempted to redirect the current task, but are pended because the current task is protected from them.

TLOCKPRR sets the flag bit when redirection of a protected task is attempted. TLOCKCPR resets flag bits when a task ?TUNLOCKs.

?TLOCKR<ring>: Pended Request indicators

If a task attempts to redirect a protected task, the protected task needs some means of knowing what action it should take when it unlocks. The ?TLOCKR<ring> ("pended Request") doublewords are used to indicate the most recent redirection request of the highest priority type which applies to the ring. They only have meaning if at least one bit in ?TLOCKPY is set.
The actions are stored in ?TLOCKR<ring> as follows:

* ZERO: no pended requests for this ring.

* ?TLR.SUS someone wants to SUSPEND us

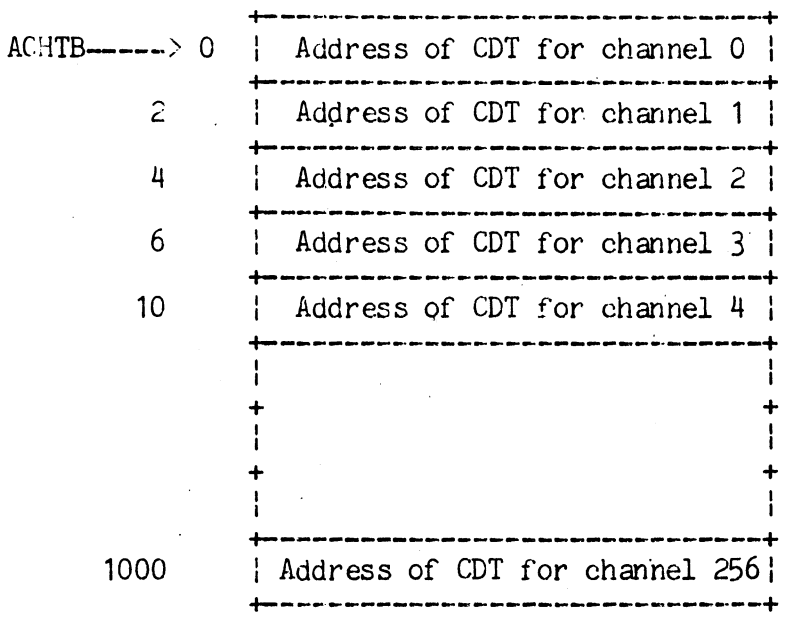
* anything other than (0,?TLR.SUS,?TLR.KILL) is the address which someone wants us to IDGOTO.

* ?TLR.KILL: someone wants to KILL us

Channel Table and Channel Descriptor Tables - CDTs

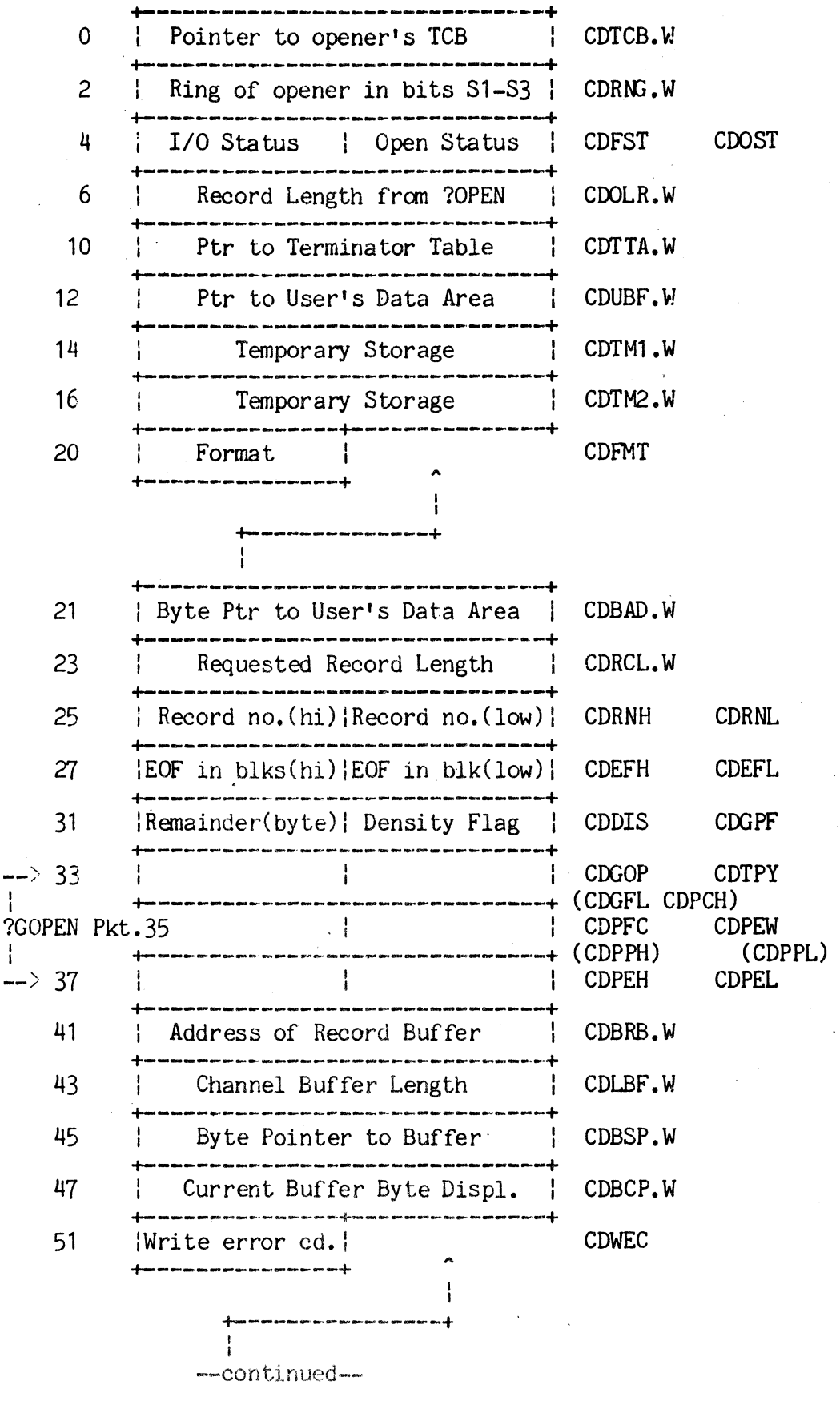
CDTs are used to describe the characteristics of a channel. A CDT is added to the channel table each time a channel is opened via the ?OPEN call. The information in the CDT is used to read from and write to a channel via the ?READ and ?WRITE calls. The CDT for a particular channel is deleted from the channel table when the channel is closed via the ?CLOSE call. CDTs are not used for channel activity through system calls other than these.

The following table describes the structure of the channel table.

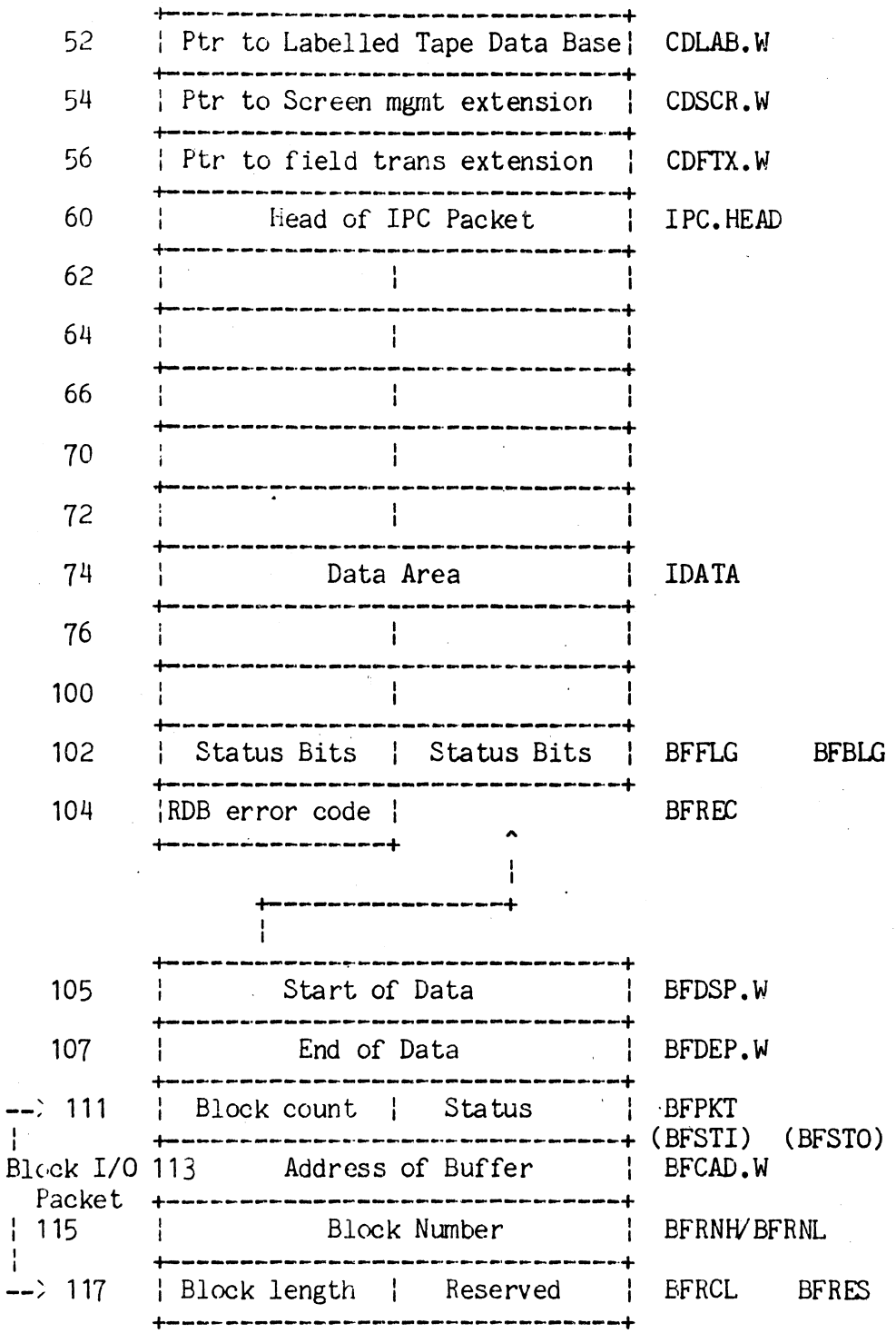


The entry for a channel in the channel table is zero until the channel is opened. When it is opened, a CDT is built and the address of the CDT is placed in the appropriate entry of the channel table.

The following table describes the CDTs.



--continued--


52
Ptr to Labelled Tape Data Base
CDLAB.W
54
Ptr to Screen mgmt extension
CDSCR.W
56
Ptr to field trans extension
CDFTX.W
60
Head of IPC Packet
IPC.HEAD
62
64
66
70
72
74
Data Area
IDATA
76
100
102
Status Bits
Status Bits
BFFLG
BFBLG
104
RDB error code
BFREC
105
Start of Data
BFDSP.W
107
End of Data
BFDEP.W
111
Block count
Status
BFPKT
(BFSTI) (BFSTO)
Block I/O Packet
113
Address of Buffer
BFCAD.W
115
Block Number
BFRNH/BFRNL
117
Block length
Reserved
BFRCL
BFRES

The following symbols are defined as bit offsets from the start of the CDT to bit in the I/O status flag work CDFST:

BCDFG - offset to bit 0 - Create flag; Used only for opening the file
BCDNG - offset to bit 1 - Negative flag
BCDEC - offset to bit 2 - Echo bit
BCDES - offset to bit 3 - End of address space encountered
BCDVH - offset to bit 4 - Variable header
BCDET - offset to bit 5 - End of labelled tape
BCDMB - offset to bit 6 - Multiple buffers
BCDNA - offset to bit 7 - New address bit
BCDSH - offset to bit 8 - Shared file bit
BCDAP - offset to bit 9 - Append flag; Used only for opening generic files
BCDWA - offset to bit 10 - Write access allowed to file
BCDPI - offset to bit 11 - Priority request allowed
BCDFS - offset to bit 12 - ?FSTAT has been done on this read
BCDSC - offset to bit 13 - Caller has previously issued screen management reads
BCDER - offset to bit 15 - Error occured

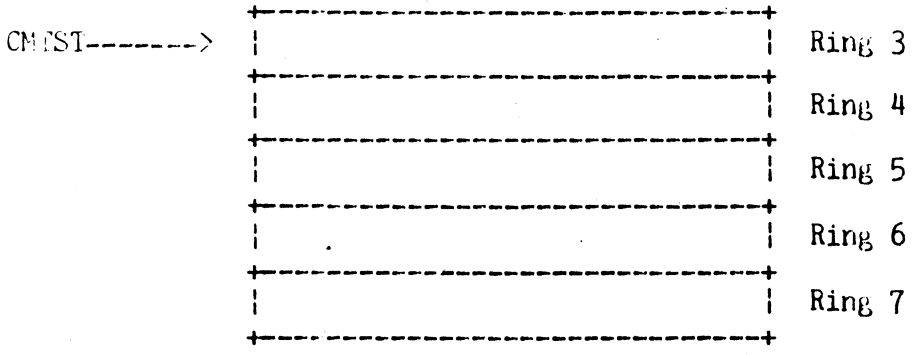
The following symbols are defined as bit offsets from the start of the CDT to bit in the Open status flag work CDOST:

BCDPD - offset to bit 0 - File is a peripheral device (ie. controlled by PMGR)
BCDSI - offset to bit 1 - Serial I/O device
BCDMT - offset to bit 2 - Magnetic tape file
BCDSP - offset to bit 3 - Spool file
BCDMF - offset to bit 4 - Magic file indicator
BCDSR - offset to bit 5 - Records are spanned
BCDVR - offset to bit 6 - Variable record processing
BCDNX - offset to bit 7 - Current block is next block
BCDEQ - offset to bit 8 - EXEC spool queue
BCDIP - offset to bit 9 - IPC file
BCDLT - offset to bit 10 - Labelled magnetic tape
BCDIN - offset to bit 11 - Input file
BCDOT - offset to bit 12 - Output file
BCDCR - offset to bit 9 - Create file
BCDCE - offset to bit 10 - Correct error

Channel Maximization Table - CMT

The CMT is a bit table used by all system calls that use channels, not just ?OPEN, ?READ, ?WRITE. and ?CLOSE. It is used to check if a task has access to a given channel.

The CMT is shown in the following table.

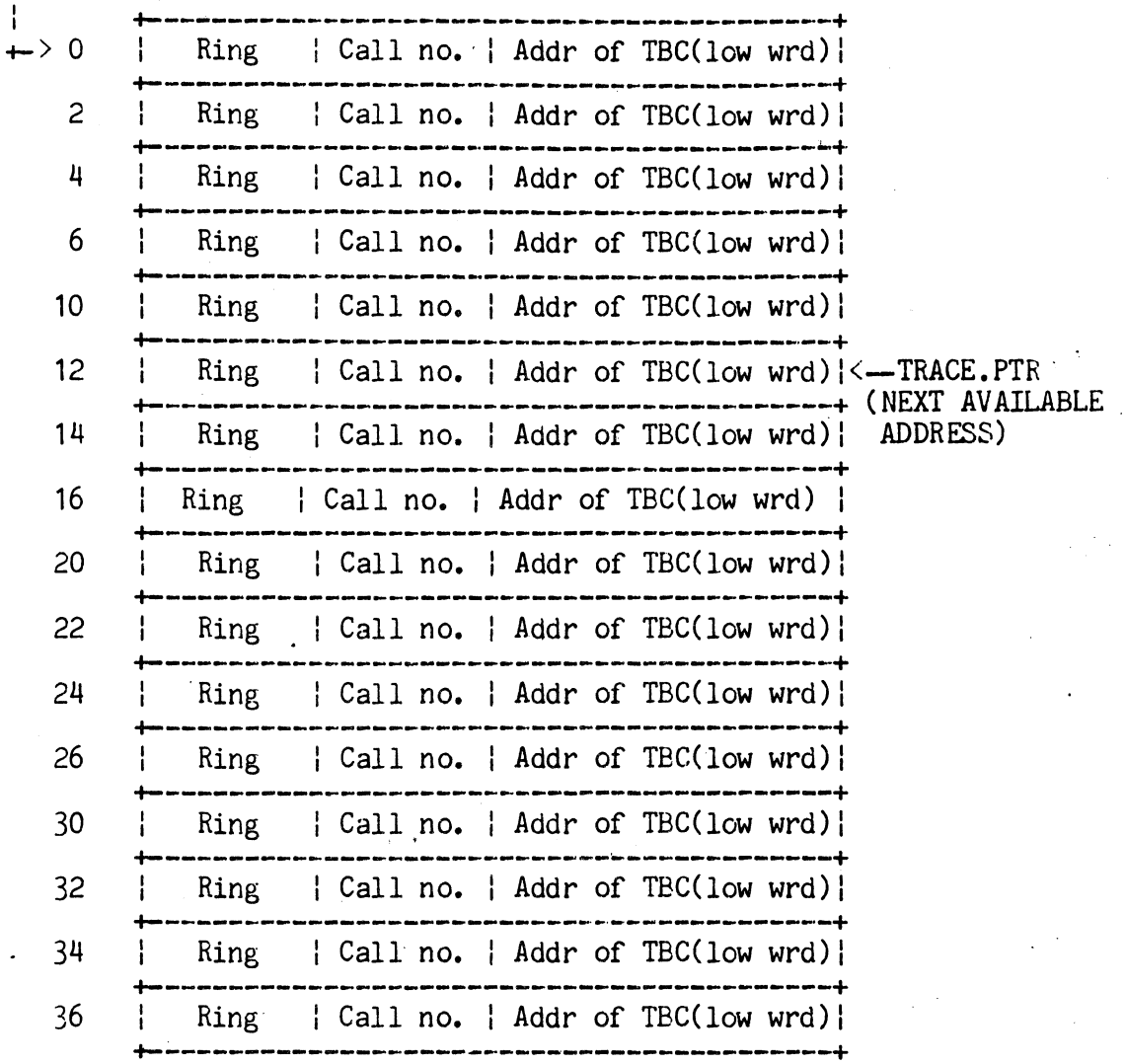


There is a bit for every channel within each of the Rings (3-7).

System Call Ring Buffer

The System Call Ring Buffer logs the most recent 16 system calls made. The buffer wraps around as a ring; the first double word logically follows the last double word.

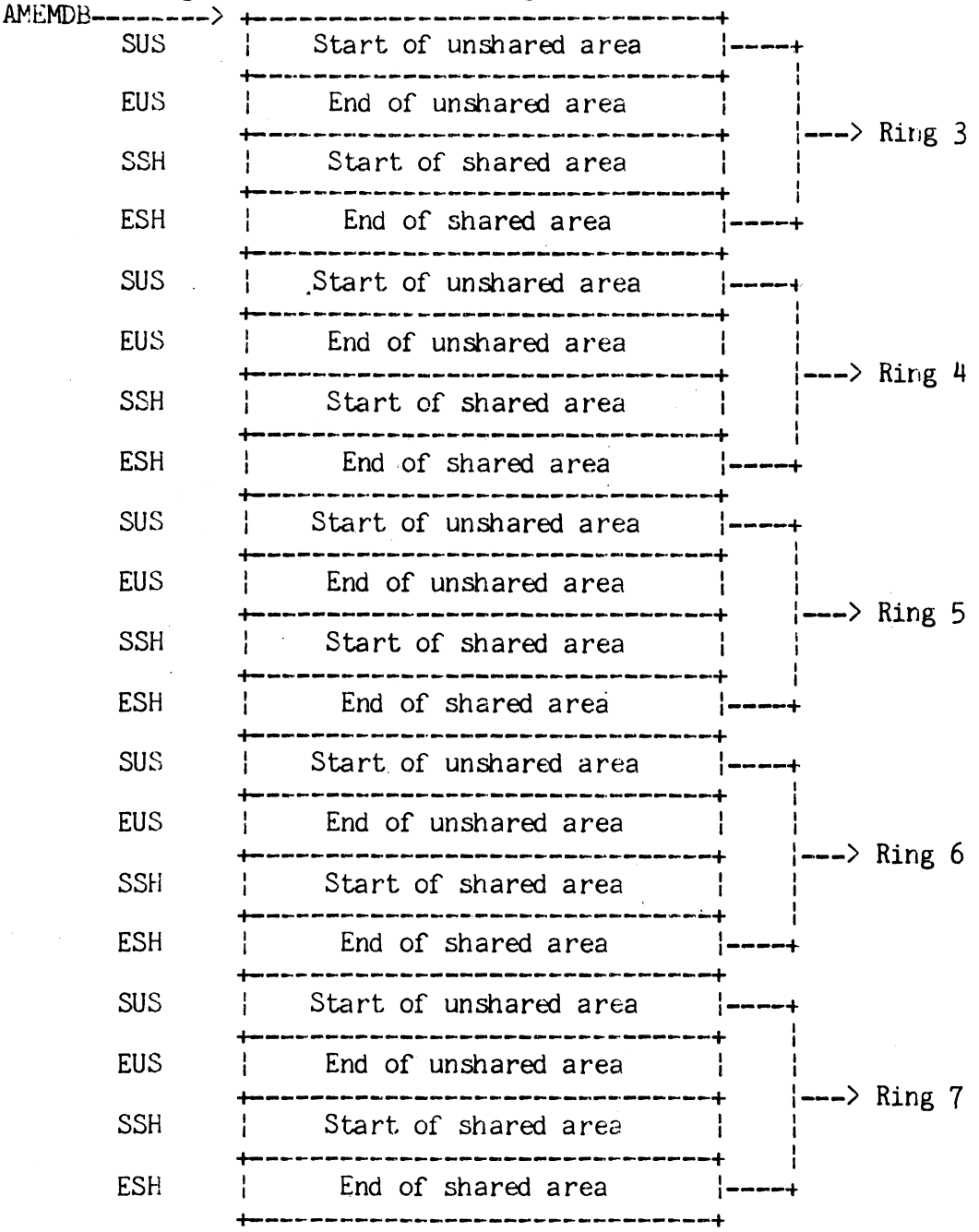
The following table describes the Agent Ring Buffer.



Agent Memory Data Base (Rings 3 - 7)

The Agent Memory Data Base maintains the starting and ending addresses of the unshared and shared areas of rings 3 through 7.

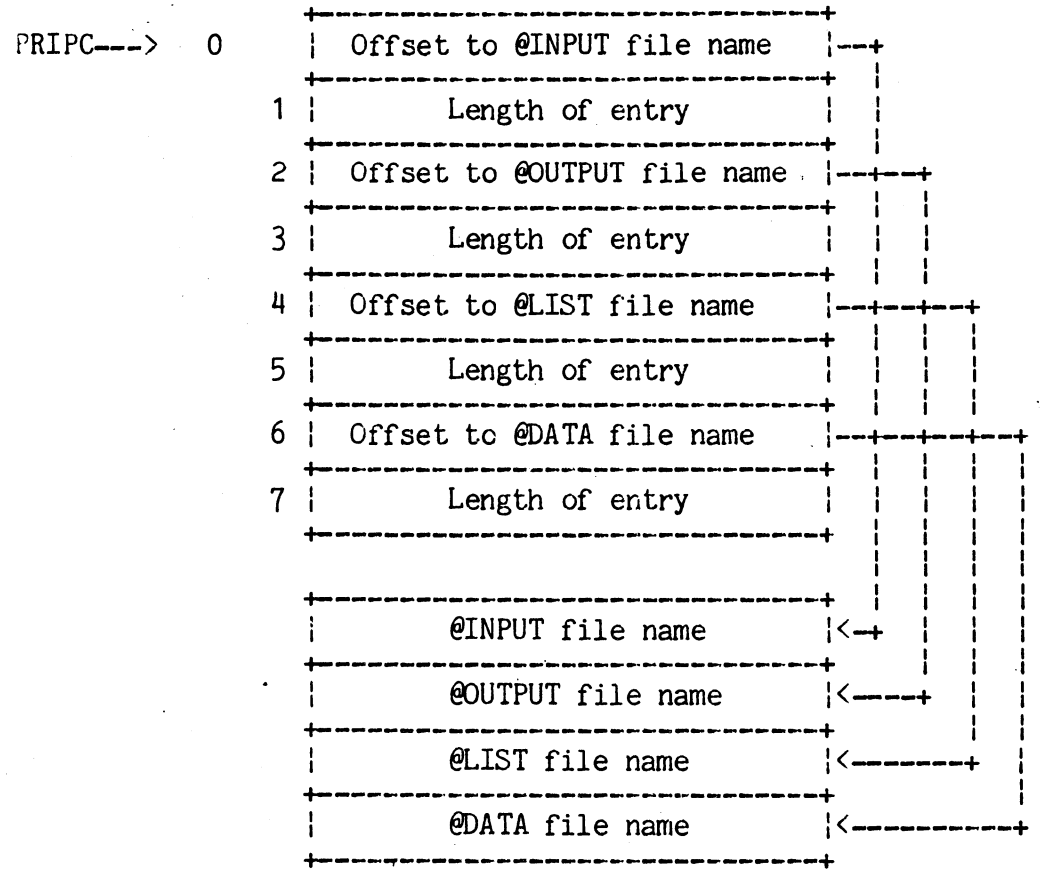
The following table describes the Agent Memory Data Base



Generic File Message Buffer

The Generic File Message Buffer contains information on resolving the generic files @INPUT, @OUTPUT, @LIST, and @DATA. When these files are opened, the files assigned to them are resolved and opened.

The Generic File Message Buffer has the following format.



All offsets in the Generic Message File Buffer are self relative.

Overlay Descriptor Table

The Overlay Descriptor Table is used for 16 bit processes only. It describes the nodes of the overlays and the areas of each node.

Notes on overlay descriptors:

* There may be zero or more node descriptors, one for each overlay node present.

* For each node descriptor, there is at least one area, and there is one area descriptor per area.

* To compute the address for a given area descriptor, take the sum of the following:
  A. The starting address of the node descriptor.
  B. The number of words in a node descriptor.
  C. The product of the area number (zero relative) and the length of an area descriptor.

* The total word size needed for a node descriptor is the sum of the following:
  A. One word (16 bit table) or two words (32 bit table) for the pointer to the descriptor, within the array node descriptor pointers at the beginning of the table.
  B. The size of the node descriptor (N.NDFAR or W.NDFAR).
  C. The number of areas times the area descriptor size (N.NDARS or W.NDARS).

* The total word size needed for an overlay descriptor table is the sum of the sizes for node descriptors plus one word (16 bit table) or two words (32 bit table) for the number of nodes.

The Overlay Descriptor Table has the following format.

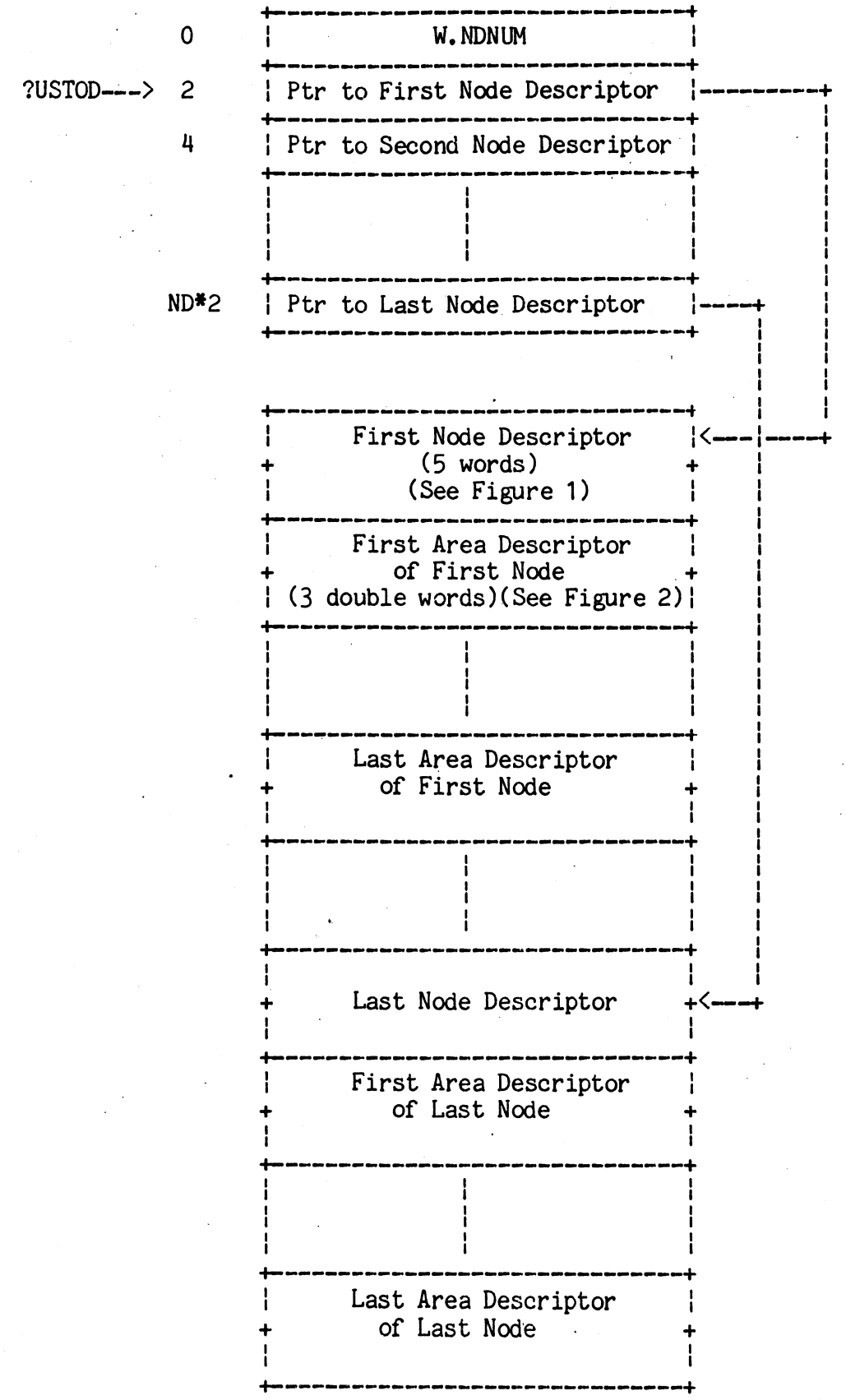


Figure 1
Node Descriptor

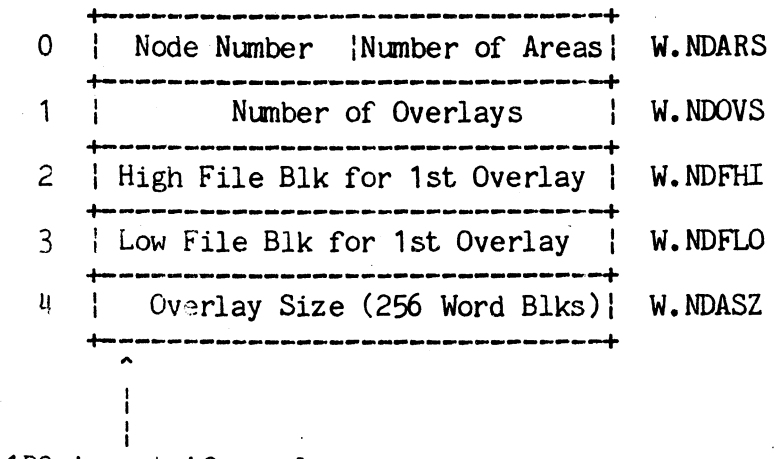
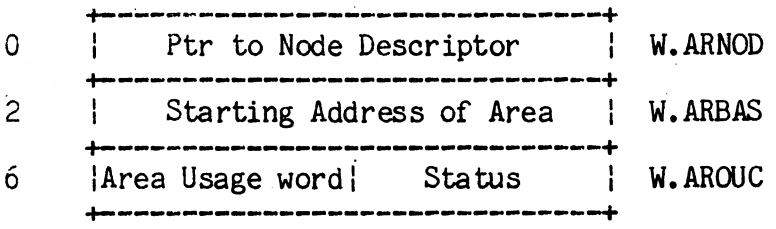


Figure 2
Area Descriptor



The following bits have been defined in the Area Descriptor status word, W.AROUC:

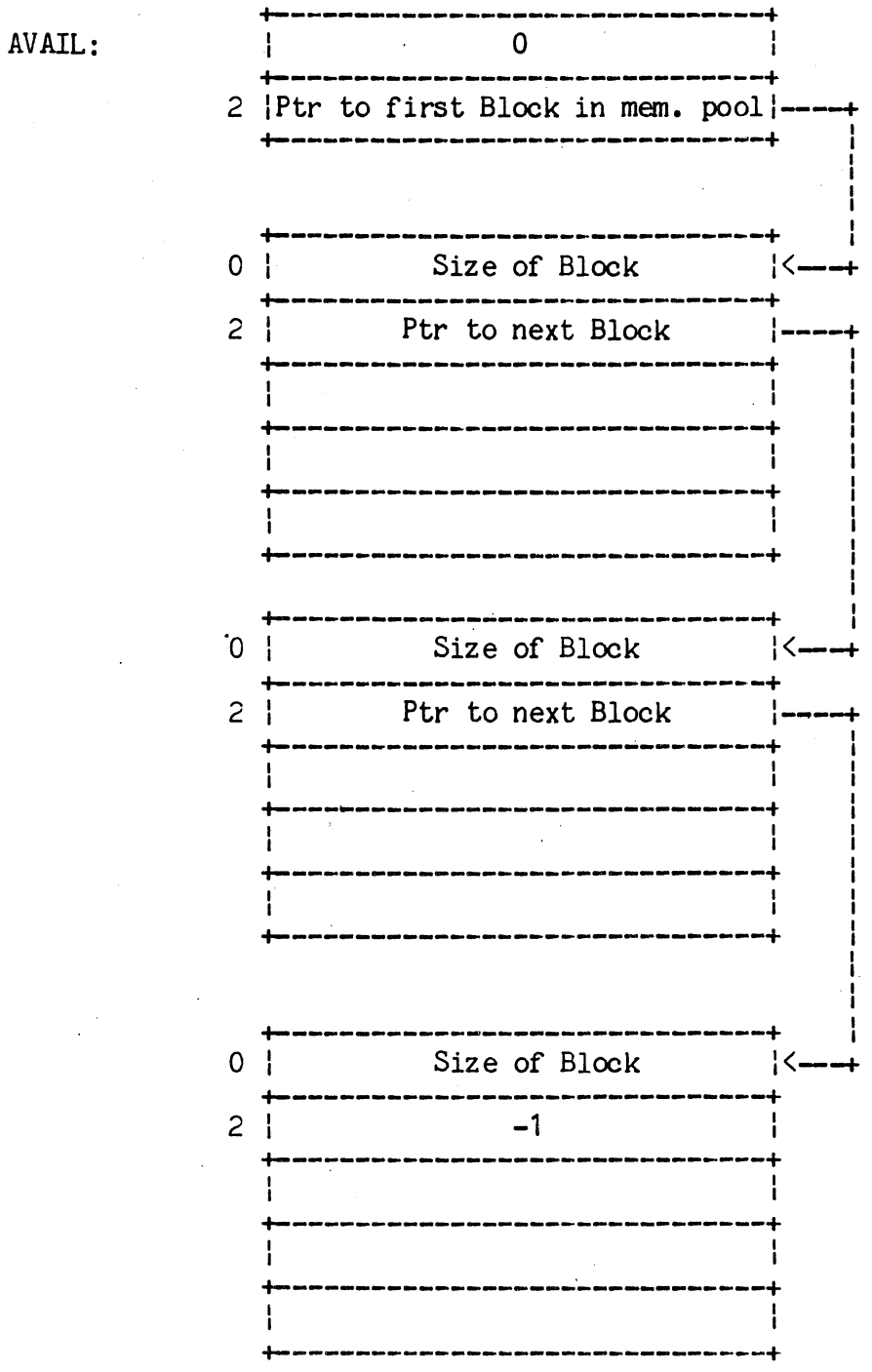
Bit 0 - loading flag
Bits 1 - 9 - Overlay number
Bits 10 - 15 - Usage count

Agent Heap Memory

Agent memory is divided into two parts, unshared and shared. Each of these has its own data base. We will first take a look at the unshared data base.

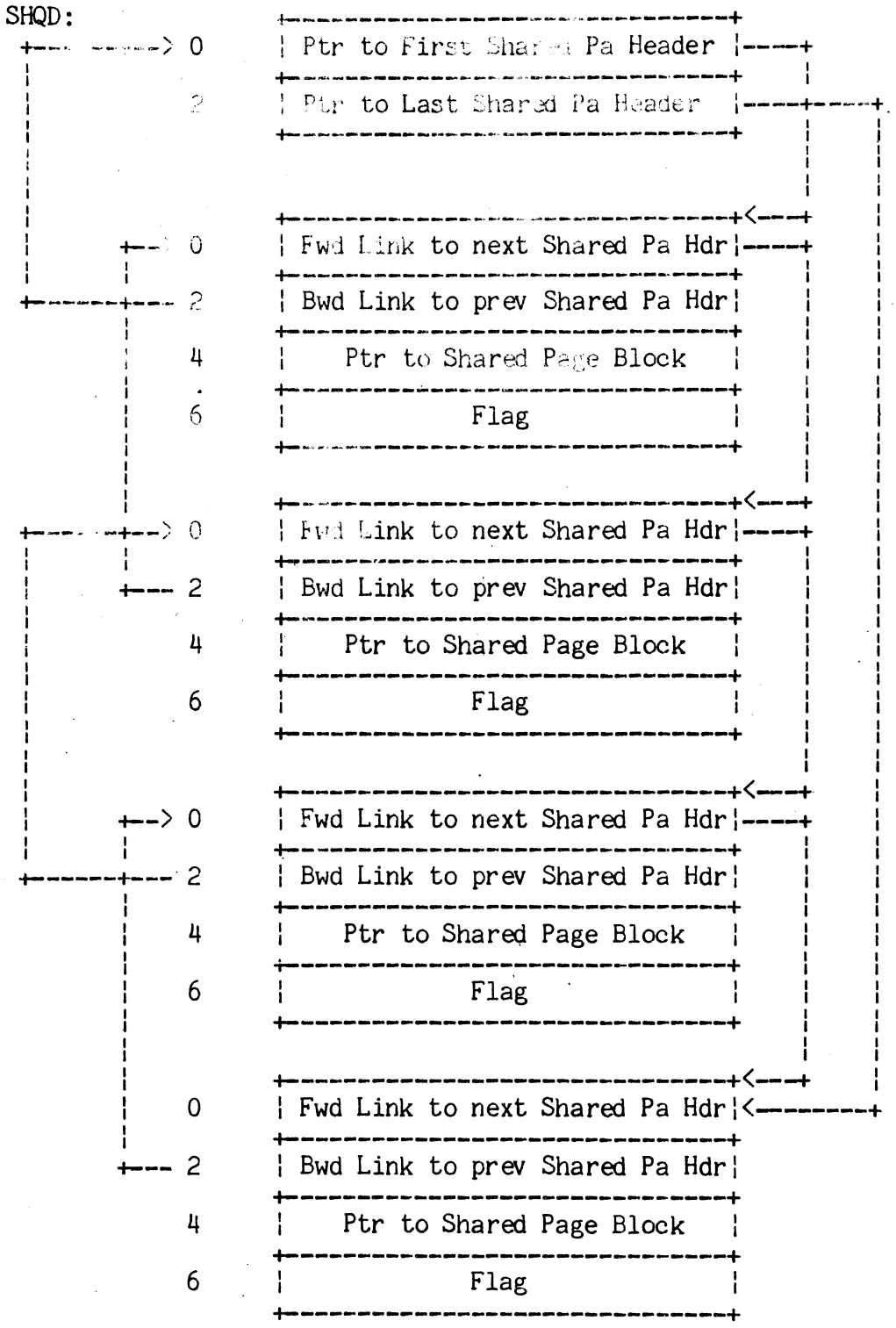
Agent Unshared Memory Data Base

The Agent maintains the following data base for its unshared memory pool.



Agent Shared Memory Data Base

The Agent maintains the following data base for the shared memory pool.

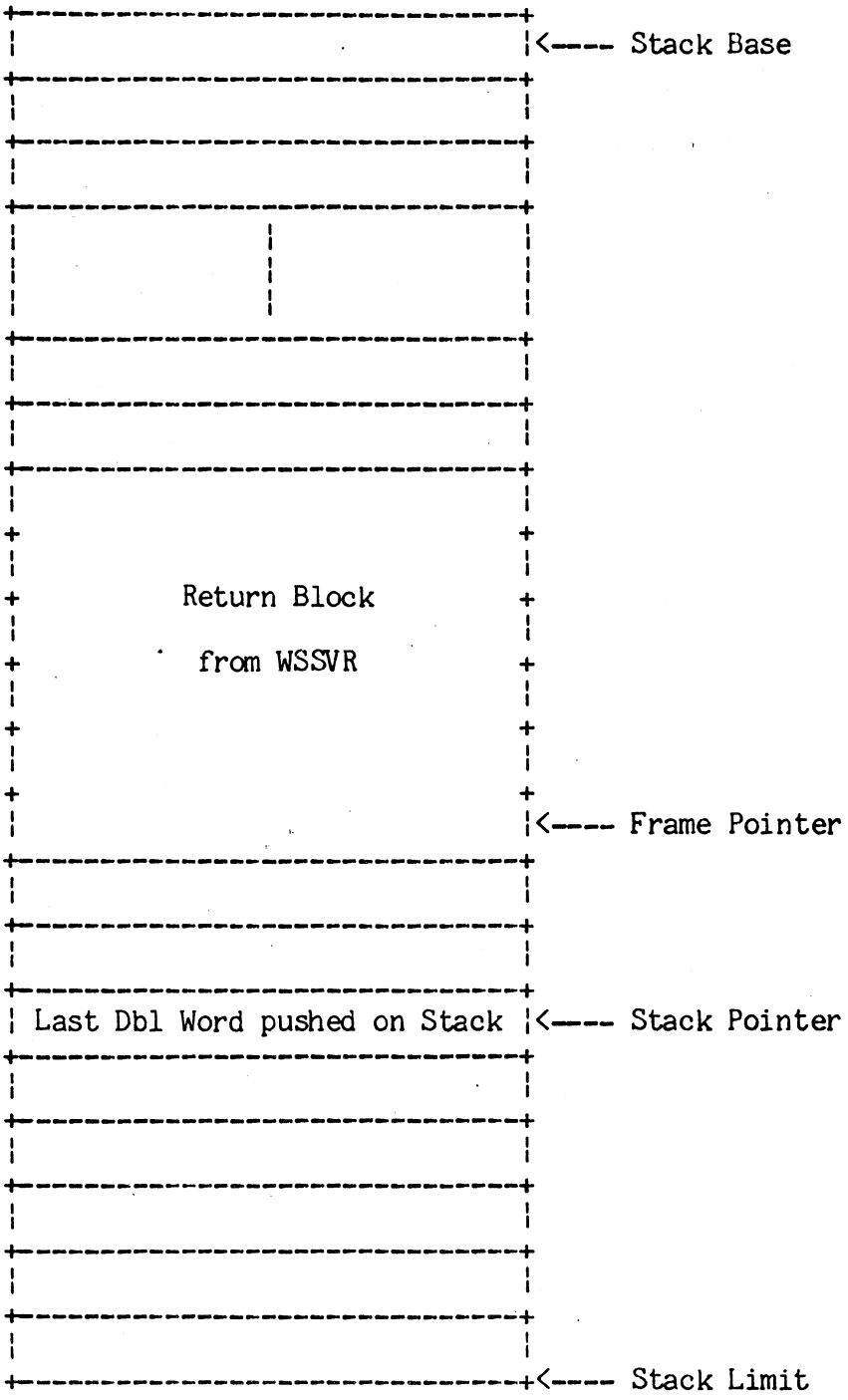


The flag in the shared page header indicates whether of not the shared page block is in use.

Agent Stacks

The Agent stack is used to store return blocks, and for temporary storage. There is a stack for each possible task. The maximum number of possible task is specified in the UST at USTTC.

The Agent Stack has the following format. There is a similar stack for each task.



Agent Initialization

At ?PROC time the Kernel passes control to the entry point AINIT of the Agent if the process is not coming up in the debugger. If the process is coming up in the debugger, control is passed to ADEBUG. At ADEBUG the Agent sets the "debug" flag in Agent flag word, then passes control to AINIT. AINIT does the following initialization.

* If there is a symbol table (ie. the contents of location 40 are non-zero), wire it so that PALANGUL does not crater, which it does when it hits a page fault on the symbol table. Then we touch each page of the symbol table to insure that it is faulted in.

* AMEMINIT is called to initialize the memory manager. AMEMINIT will be discussed in further detail later.

* Set up the memory data base for rings 3 and 7, the only loaded rings. Zero the data base for rings 4, 5, and 6.

* The channel maximization table is set up. The Agent gets memory for the table, zeros it, and stores the address of the table in page zero, (CMTST)

* The TCB extenders are set up. ?TCBXLN words are needed for each TCB extender (TCBX). The TCBX for the initial task is initialized with the ring it originated in (ring 7), the address of the TCB for this TCBX, and the ring of the original caller (ring 3).

* PMGR.INIT is called to initialize the local PMGR. Since the PMGR is out of the realm of the Agent, even though it occupies the same ring, it will not be delt with extensively in this manual.

* Build a fake return block on the stack. The starting address of the user's program is placed in the "return PC" double word in the return block. This is done so we can start up the user by doing a WRTN, which will take us to the start of the user's program.

* Initialize the TCB free chain. The first TCB following the UST is used for the initial task. The second TCB, if there is one, is the first free TCB. The number of TCBs is specified in the UST at USTTC. Since the first TCB is active, we will link USTTC - 1 TCBs in the free chain.

* The Agent then gets 192 words of memory for each task to be used as its stack. The stack parameters (SP, FP, SB, and SL) are set in each TCB for its ring 3 stack.

* The Agent sets up the .KILL table. The table contains four double word addresses. These are the addresses of the .KILL routines for rings 4, 5, 6, and 7. The addresses for rings 4, 5, and 6 are initialized to -1. The address of the ring 7 .KILL routine is taken from the UST (USTKL) and placed in the table. The address of the table is placed in the UST (USTKL).

* Part of the UST, up to and including USTPR, is copied into the user's ring (ring 7).

* The overlay table is set up for 16 bit users.

* The Agent does a ?IREC to receive the initial IPC message.

* If the user was PROC'ed up resident the Agent is wired.

* If the debug flag is set we go to the debugger to initialize it. The debugger will start the user up. If the debug flag was not set we do a WRTN to start up the user.

Agent Gates

There are seven entry points, which are accessible from outer rings, in the Agent. These entry points are appropriately called gates. An outer ring task can enter the Agent only by making an LCALL through one of these gates. This mechanism provides limited access to the system and increases the integrity of system protection. The following are a list of the seven Agent gates.

Gate number	Entry point	Function
0	INCOMING.CALLS	Main Agent gate for system calls
1	TASKA	Unsuccessful return from ?UTSK
2	TASKB	Successful return from ?UTSK
3	UKILA	Return from ?UKIL routine
4	UBKPT	User Debugger gate
5	IXMT	?IXMT gate
6	IXIT	16 bit caller ?IXIT gate

System calls come through gate zero of the Agent. In the next section we will look at how this is done, and what is done at INCOMING.CALLS to dispatch a task to the proper system call handler.

System Call Dispatching

System call dispatching is similar, but not identical, for 32 bit and 16 bit programs. We will take a look at how each one gets into the Agent.

32 bit programs

User's invoke system calls via ?XXXX, where XXXX is the name of the system call. ?XXXX is a macro which expands into the following for 32 bit programs:

```
        XJSR @6
        <system call number corresponding to XXX>
```

The "@" symbol stands for indirect addressing. For the 32 bit program this means take the address at location 6 and jump to that address. This is the same as

```
        XWLDA 2,6
        XJSR 0,2
```

At the address specified by location 6 is the system call wart (SYST), which looks like the following.

```
SYST:
        WPSH 3,3
        LCALL 6000000000,0
        WBR SYST1
        ISZTS
SYST1:
        ISZTS
        LDAPF 3
        WPOPJ
```

At this point, an explanation of how this mechanism works is called for. The XJSR @6 causes the address of the next instruction, in this case the address of the system call number, to be loaded into C3, and jumps to the specified address, which in this case is the system call wart. The system call wart pushes the address of the system call number on the stack (WPSH 3,3) and does an LCALL into the Agent through gate zero. Gate zero is the entry point for all system calls. Before we discuss how the system call is handled once in the Agent, let's take a look at how 16 bit programs get into the Agent.

16 Bit Programs

16 bit programs invoke system calls in the same manner as 32 bit programs, via ?XXXX. ?XXXX expands into the following for 16 bit programs.

```
      JSR @17
<system call number corresponding to XXXX>
```

The address specified by location 17 is the system call wart for 16 bit programs (SYST). It looks like the following.

```
SYST:
          PSH 3,3
          LCALL 6000000000,0
                JMP SYST1
                ISZ @40
SYST1:
                ISZ @40
                LDA 3,41
                POPJ
```

The 16 bit system call wart does the same thing as the 32 bit system call wart. Namely, it pushes the address of the system call number on the stack and does an LCALL into the Agent through gate zero.

Gates and gate arrays will be discussed further later in this manual. For now, it is sufficient to know that a task entering the Agent through gate zero enters at INCOMING.CALL. The Agent goes through the following series of steps to dispatch the task off to the proper system call handler.

* the PSW portion of the return block has already been pushed on the stack by the LCALL. The remainder of the return block is pushed on the stack by a WSAVR instruction. The LCALL loads the address of the instruction following the LCALL into AC3. Therefore, when AC3 is saved on the stack, by the WSAVR, it is the return PC which is pushed onto the stack.

* The Agent gets the caller's stack pointer from page zero of his ring. It loads the return PC from the top of the caller's stack (remember the WPSH 3,3 [or PSH 3,3 in the case of 16 bit programs] from the system call wart). The Agent then loads the single word pointed to by the return PC (remember the return PC pointed to the system call word).

* Next location 2 of Agent page zero is checked. The value in this location is interpreted as follows:

  0: process the system call in the Agent as usual. This is the standard mode of operation.

-1: make the kernel system call directly, bypassing all Agent processing. This mode is only used by the Kernel and File Systems group for testing system calls that do not have pre-processors in the Agent yet.

XXX: bypass Agent processing of the system call which has the system call number "XXX". However, allow the Agent to process all other system calls. This mode is also used by the Kernel and File Systems group for testing.

Since we are interested in the system call processing done by the Agent, and since this is the usual mode of operation, we will ignore the last two cases and concentrate on the first. It is sufficient to say that in the latter two cases, if Agent processing is not to be done, the Agent LCALLs into the Kernel at this point.

* The Agent checks to make sure that the system call number supplied by the user is valid (ie. that it is between the lowest and highest system call numbers allowed). If it is not valid, we return to the user with error code ERICM (Illegal System Command). If the system call number is valid, we increment CALL.TOT, which is a count of the number of system calls this process has made.

* The Agent checks if system call logging is turned on. If it is it jumps to a routine to log the system call. System call logging is turned on via ?LOGCALLS.

* The Agent then logs the system call in its system call ring buffer. If the ring buffer does not exist (ie. this is the first system call this process has made), it is created. Then the ring of the caller, the system call number, and the lower word of the address of the TCB making the call is logged into the buffer.

* If the caller is a 16 bit program, the dispatch table used contains entry points for the 16 bit code. If the caller is 32 bit, another dispatch table is used which contains the entry points for the 32 bit code. The Agent calls CALLER.TYPE to determine if the caller is 16 bit or 32 bit, then gets the base address of the proper dispatch table. The system call number is doubled to produce a double word offset into the dispatch table for this system call handler. The address of the system call handler for this system call is load in an AC from the dispatch table. If the address is minus one, there is no system call with this number, and the Agent takes an error return with the error code ERICM. If the address is not minus one, it is pushed on the stack, and the Agent does a WPOPJ to go to the system call handler. The WPOPJ pops a double word off the stack, uses it as an address, and jumps to that location.

Now that we have looked at the data bases in the Agent, what the Agent does during initialization, and how system calls are dispatched to the appropriate system call handler, let's take a look at the common routines in the Agent. These routines are commonly used throughout the Agent, so it is advantageous to know what these routines are and what they do. That way you will be familiar with these routines when we look at the various system calls.

Common Agent Routines

The common Agent routines fall into 10 categories.

* Agent Initialization
* Call Dispatching
* Caller Identification - 16 or 32 bit
* Word and Byte Pointer Validation
* Agent Heap Memory Management
* Resource Locking
* Abort Handling
* Deterministic Scheduling
* Channel Management
* Common Return Points

We have already discussed Agent Initialization and system call dispatching, so we will not go into futher discussion of these at this point. However, we will look at the others in more detail in the following sections.

Caller Identification

The routine used to identify a caller as either a 32 bit caller or a 16 bit caller is CALLER.TYPE. It is called in the following manner.

```
LPSHJ CALLER.TYPE
<return for 32 bit caller>
<return for 16 bit caller>
```

Input:
No input required
Output:
AC0 - Unchanged
AC1 - Unchanged
AC2 - Unchanged
AC3 - Unchanged.

CALLER.TYPE determines the type of caller in the following manner.

* It checks USTPR in the UST. USTPR is set up by LINK and is zero for a 32 bit program and non-zero for a 16 bit program. If USTPR is zero, the caller is 32 bit, and that return is taken.

* If USTPR is non-zero, CALLER.TYPE checks the ring that the call came from. If the call was from ring seven, then he is truly a 16 bit caller, otherwise he is a 32 bit caller.

Word and Byte Pointer Validation

The word and byte pointer routine names are governed by their function. The format of the routine names is that the first letter is an "A", for Agent routine. The second letter is an "R" or "W", which mean validate for read access or validate for write access respectively. The third letter is a "V" for validate. The next two letters are either "WP" or "BP", which stand for word pointer or byte pointer. The last letter indicates the delimiters used for validation. An "L" stands for absolute length. The entire length of the buffer must be validated. An "X" means validate a string of unknown length, but known maximum length. The string is validated until a delimiter is encountered. A "D" means the same as an "X", but the string is terminated by two consecutive delimiters. Lastly, an "ACL" means that the string is an access control list, which has the format

USERNAME<0><access type><0>[USERNAME<0><access type><0>...]<0>

An access control list is terminated by two consecutive delimiters.

Combinations of these gives us the following routines.

* ARVWPL - Validate a word pointer to a string of known length for read access
* AWVWPL - Validate a word pointer to a string of known length for write access
* ARVBPL - Validate a byte pointer to a string of known length for read access
* AWVBPL - Validate a byte pointer to a string of known length for write access
* ARVBPX - Validate a byte pointer to a string of unknown length, but known maximum length for read access. The string terminates with a delimiter.
* ARVBPD - Validate a byte pointer to a string of unknown length, but known maximum length for read access. The string terminates with two delimiters.
* ARVBPACL - Validate a byte pointer to an access control list.

Word and Byte Pointer Validation of a String of Known Length

These are the routines that end with an "L". These routines are used when a pointer to a string or buffer of known length must be validated. An example of this is the validation of pointers to parameter packets. These routines are called in the following manner.

```
LPSHJ A<R,W>V<W,B>PL
<exception return>
<normal return>
```

Input:
AC0 - Word or byte pointer to the string
AC1 - Ring of caller in bits S1 - S3
AC2 - Length of string in words or bytes

Output:
AC0 - Unchanged for normal return, error code for exception return
AC1 - Unchanged
AC2 - Unchanged
AC3 - Unchanged

The logistics of these routines are as follows.

* Check that the ring of the caller's word/byte pointer is greater than or equal to the callers ring (AC1 input). If not take an error return.

* Check that the target address falls within either the shared or unshared area of caller's word/byte pointer ring. If not take an error return

* Check that the caller has the appropriate access to the specified area (the shared area is read only). If not take an error return.

* Check that the ending address falls within the same area, shared or unshared, as the target address. The ending address is calculated by adding the length to the target address. If not take an error return.

Byte Pointer Validation of a String of Unknown Length

This is the routine that ends with an "X". It are used to validate a string of unknown length, but known maximum length, Where the string is terminated by a delimiter. An example of its use is the validation of data sensitive text strings. Obviously, only read validation can be done on a string of unknown length, terminated by a delimiter. This routine is called in the following manner.

```
LPSHJ ARVBPX
<exception return>
<normal return>
```

Input:

AC0 - Byte pointer to the string
AC1 - Ring of the caller in bits S1 - S3
AC2 - Maximum length of the string in bytes
AC3 - Address of delimiter table

Output:

AC0 - Unchanged for normal return, error code for exception return.
AC1 - Unchanged
AC2 - Unchanged
AC3 - Unchanged

The logistics of this routine are very similar to the "L" routines. However, if the target address is valid, and his maximum ending address is invalid, the following additional check is made.

* Starting at the target address, search forward through the caller's string looking for a match with any of the delimiters in the delimiter table. The search is ended when a delimiter is found or at the end of the area, shared or unshared. If a delimiter is found in this range, the byte pointer is valid.

Byte Pointer Validation of a String of Unknown Length - two delimiters

This routine ends with a "D". It is used to validate a string of unknown length, but known maximum length, where the string is terminated by two consecutive delimiters. It may be used to validate list, where each element in the list is separated by a delimiter. The list must be terminated with two consecutive delimiters. This routine is called as follows.

```
LPSHJ ARVBPD
<exception return>
<normal return>
```

Input:

AC0 - Byte pointer to the string
AC1 - Ring of the caller in bits S1 - S3
AC2 - Maximum length of the string in bytes
AC3 - Address of the delimiter table

Output:

AC0 - Unchanged for normal return, error code for exception return
AC1 - Unchanged
AC2 - Unchanged
AC3 - Unchanged

The logistics of this routine are similar to ARVBPX, except if the final search is conducted, the search is for a pair of delimiters rather than a single delimiter.

Byte Pointer Validation of an Access Control List

This routine is used to validate a byte pointer to an access control list. The list must be terminated by two consecutive delimiters. The format of an access control list is as follows.

USERNAME<0><access type><0>[USERNAME<0><access type><0>...]<0>

This routine is called in the following manner.

```
LPSHJ ARVBPACL
<exception return>
<normal return>
```

Input:
AC0 - Byte pointer to the string
AC1 - Ring of the caller in bits S1 - S3
AC2 - Maximum length of the string in bytes
AC3 - Address of the delimiter table

Output:
AC0 - Unchanged for normal return, error code for exception return
AC1 - Unchanged
AC2 - Unchanged
AC3 - Unchanged

The logistics of this routine are also very similar to ARVBPX, once again, except for the final search. In the final search, start at the target address and search forward for the next <0>. When it is located, space past the next byte, which is the access type byte, and check again for a <0>. If the byte after this is another <0>, this is the end of the list. At all times during the search, we make sure we stay in the same area, shared or unshared, specified by the target address.

All the validation routines have macros to do the LPSHJ. The macro names are the same as the routine names, but without the initial letter "A". For example, "LPSHJ ARVBPX" can be replaced by the macro "RVBPX".

Agent Heap Memory Management

There are two memory heaps in the Agent, unshared and shared. This memory is used by the Agent for the processing of system calls and for Agent data bases. We will look at the unshared memory management routines first, since they are the ones most commonly used.

Unshared Memory Management

The Agent unshared memory pool is initialized during Agent initialization using the following routine.

```
LJSR AMEMINIT
<normal return>
```

Input:
No required input

Output:
AC0 - Unchanged
AC1 - Unchanged
AC2 - Unchanged
AC3 - Frame pointer

AMEMINIT initializes the unshared memory pool by allocating any Agent unshared memory that follows the initial Agent stacks within that same page. Memory is allocated for use from the unshared memory pool using the following routine.

```
LJSR AGMEM
<exception return>
<normal return>
```

Input:
AC0 - Number of words of unshared memory requested

Output:
AC0 - Unchanged for normal return, error code for exception return
AC1 - Unchanged
AC2 - Address of the unshared memory block
AC3 - Frame pointer

AGMEM performs a "first fit" search of the unshared memory pool to satisfy the caller's request. It starts at the pool descriptor, location AVAIL in Agent page zero, and searches forward. If a block of sufficient size is not located, then a call is made directly to the Kernel (?MEMI) to acquire more memory. Acquired memory is placed at the end of the list. When a block of sufficient size is located, the requested size is carved

out from the beginning of the block, and the remainder, if any, is linked back into the list. AGMEM always returns at least eight words, and request have an upper limit of 262144 words (1000000 octal). AGMEM calls LOK with the double word MEMLOCK to ensure single tasking through AGMEM.

When the Agent is finished with allocated unshared memory, it calls the following routine to return the memory to the unshared memory pool.

```
LJSR AFMEM
<exception return>
<normal return>
```

Input:

AC2 - Address of the unshared memory block to be returned

Output:

AC0 - Unchanged for normal return, error code for exception return
AC1 - Unchanged
AC2 - Unchanged
AC3 - Frame pointer

AFMEM searches through the unshared memory pool until it locates either a higher block than the input block, or the end of the list. The specified block is then linked into the list at this point, and, if possible, it is merged with any neighboring blocks to make one larger contiguous block, thus decreasing fragmentation. AFMEM also calls LOK with the double word MEMLOCK to ensure single tasking with AFMEM (and AGMEM), thus maintaining the integrity of the unshared memory pool.

Shared Memory Management

There are two routines which handle the shared memory pool within the Agent. The routine to acquire memory from the pool is the following.

```
LJSR ASGMEM
<exception return>
<normal return>
```

Input:
No input is required

Output:
AC0 - Unchanged for normal return, error code for exception return
AC1 - Unchanged.
AC2 - Address of the shared block
AC3 - Frame pointer

ASGMEM acquires a shared page block from the shared memory pool. Each block is "SHMIM" shared pages long. SHMIN is defined to be long enough to satisfy all callers of ASGMEM. ASGMEM searches through the shared page header queue to find a free shared page block. The flag double word within the header indicates whether or not the corresponding block is free. If a header that indicates a free block is found on the queue, the address of the block is returned to the caller and the flag in the header is modified to indicate that the block is in use. If a free shared block is not found, a new shared page header is allocated from the unshared memory pool and a new shared page block is acquired by a ?SSHPT call. The header is linked into the queue with a flag indicating that the block is in use. The header is pointed to the shared block, and the address of the shared block is returned to the caller.

To free shared memory, the following routine is called.

```
LJSR ASFMEM
<exception return>
<normal return>
```

Input:
AC2 - Address of the shared block
Output:
AC0 - Unchanged for normal return, error code for exception return
AC1 - Unchanged
AC2 - Unchanged
AC3 - Frame pointer

ASFMEM searches through the shared page header queue for the header corresponding to the specified block. It then resets the header flag to

indicated that the shared page block is free. Both ASGMEM and ASFMEM call LOK with the double word lock SHLOCK. This ensures single tasking within these two routines.

Resource Locking

The Agent locks resources both globally and locally. Global locking is accomplished by two modes of disabling scheduling. Local locking is implemented by software locks of specific code paths and data bases. First, we will take a look at global resource locking.

Global Resource Locking

There are two methods used by the Agent to globally disable multitasking. Both disable rescheduling for other task within a process. When scheduling is disabled within a process, the only task which was the current task at the time of the disable (ie. the task that did the disable) will run.

The first method of disabling scheduling, and the most commonly used, is to set the single word "?TSMA" to a non zero value. ?TSMA is defined as location "one" of Agent page zero. ?TSMA is reset by the Kernel as the first part of a Kernel system call request. Therefore, if the Agent makes a call to the Kernel to reschedule a task, ?TSMA is set to zero upon entering the Kernel, and all task that are not pended for other reasons are considered for rescheduling. An example of the use of ?TSMA is when the Agent is playing with the UST or TCBs. ?TSMA is set so no other task will be scheduled, thus guaranteeing that the UST and TCBs will not be in a transient state when they are looked at.

The second method the Agent uses to disable scheduling is by setting the disable scheduling bit, ?UFPH, in the UST. This bit is set and reset by two macros.

The macro used to set the bit, thus disabling scheduling is ADRSCH. There is no input to the macro, and all ACs remain unchanged.

The macro used to reset the bit in the UST, thereby enabling task scheduling, is AERSCH. Again, there is no input to the macro, and all ACs remain unchanged.

Unlike ?TSMA, ?UFPH is not reset by the Kernel. Scheduling remains disabled until the Agent explicitly resets this flag bit. This is valuable if the Agent wants scheduling to remain disabled across a Kernel call.

Local Resource Locking

Local resource locking differs from global locking in that only task trying to access the locked resource will be pended. All other task will continue to be scheduled as usual, assuming scheduling is not disable by the Agent or the user. Code paths and data bases can be locked using the local resource locking routines.

Locking of a resource is done by calling the following routine.

```
LJSR LOK
<normal return>
```

Input:
AC2 - Address of the lock double word for this resource

Output:
AC0 - Unchanged
AC1 - Unchanged
AC2 - Unchanged
AC3 - Frame pointer

LOK associates a double word lock with a critical resource. LOK examines the lock. If it is zero, the resource is free, and the caller's TCB address is placed in the lock double word. Control is then returned to the caller. If the lock is non-zero, ?TSGS is set in the caller's TCB status word (?TSTAT). This marks the caller ineligible to run. The address of the lock double word is placed in the system call double word (?TSYS) of the caller's TCB. The variable WAITERCOUNT is incremented. Finally. LOK calls the Kernel to reschedule another task.

When a task is finished with a critical resource, it releases it by calling the following routine.

```
LJSR UNLOCK
<normal return>
```

Input:
AC2 - Address of the lock double word for this resource.
Output:
AC0 - Unchanged
AC1 - Unchanged
AC2 - Unchanged
AC3 - Frame pointer

UNLOK looks at WAITERCOUNT. If it is zero, UNLOK simply zeros the lock double word and returns to the caller. If WAITERCOUNT is not zero, UNLOK searches the active TCB chain for a task pended on a lock (?TSGS is set if the task is pended on a lock). When a task is found, UNLOK checks the address of the lock this task is pended on (it is stored in ?TSYS). If the task is not pended on this lock, UNLOK continues to search the active

TCB chain to find one that is. If no task is found pended on this lock, UNLOK zeros the lock double word and returns to the caller. If a task is found pended on this lock, UNLOK resets the located TCB's ?TSGS flag and zeros its ?TSYS double word. It places the address of the located TCB in the lock double word. If the relative priority of the located task is greater than that of the calling task, UNLOK calls the Kernel to reschedule a task. Otherwise, UNLOK returns directly to the caller.

Besides WAITERCOUNT, there is another counter used in the lock routines. It is called LOCKCOUNT and indicates the current number of task currently holding locks. Each time a task successfully locks a resource through LOK, LOCKCOUNT is incremented, and each time a lock is released through UNLOK, and there is no task waiting for this resource, LOCKCOUNT is decremented. If there is a task waiting for the lock, LOCKCOUNT is not touched, because directly after UNLOK releases the resource from one task, it gives it to another.

LOK and UNLOK disable scheduling by setting ?TSMA, to ensure that there are no race conditions with task trying to secure the same resource.

Abort Handling

When a task is being redirected, various cleaning up must be done before the task is allowed the proceed to its new destination. An example of this "cleaning up" is that any system call the task is pended on must be ripped down, and the task unpended. The system call that does this is called ?GABORT, and is called in the following manner.

```
?GABORT
<exception return>
<normal return>
```

Input:
- AC0 - Address of the TCB of the task to be aborted
- AC1 - PC to redirect the aborted task to

Output:
- AC0 - Unchanged for normal return, error code for exception return
- AC1 - Unchanged
- AC2 - Unchanged
- AC3 - Frame pointer

The ?GABORT system call is only valid if it is made from within the Agent. It rips down all system calls in progress, and may be called during ?IDGOTO, ?IDKIL, or ?PRKIL. ?GABORT clears certain task suspension bits, aborts ?READ/?WRITE request to the PMGR and IPC request to X25, and tears down certain system calls. It guarantees orderly restoration of Agent resources before the aborted task proceeds to its new instruction sequence. By convention, before calling ?GABORT, the task to be aborted is pended by setting the abort flag (?TSAB) in ?TSTAT of its TCB.

The logistics of ?GABORT are the following.

* Validate the callers input arguments.
* If the Abort PC table has not yet been allocated, allocate it now.
* Place the input PC in the Abort PC table entry corresponding to the ordinal position of the specified task's TCB, and zero the corresponding mailbox entry.
* Place the address of the special GABORT post-processing routine in the first return block on the Agent's stack. This allows the aborted task to finish up processing in the Agent before it is redirected. When this task does its last WRTN, which would bring it back into user space, it is sent to this routine to be redirected.

* For the task to be aborted, clear the ?TSTAT flag bits in its TCB corresponding to task suspension due to ?SUS, ?IDSUS, ?PRSUS, ?XMTW, ?TRCON, or 16 bit overlay waiting.

* If ?TSPN is set in ?TSTAT, the task to be aborted is pended on some other system call. If it is not set, continue to the last step.

* Check if the task is pended on a ?WTSIG or ?SIGWT to the PMGR. If so, see if the system call he made was a ?READ or ?WRITE. If it was prepare to send a special ?IS.R call to the PMGR to tell it to rip down the ?READ/?WRITE on behalf of the task to be aborted. We get the PMGR's control port by getting the CDT for the channel that this task has open to the PMGR (from the ?READ/?WRITE parameter packet). We can then get the PMGR's control port from the CDT directly.

* The Agent issues an ?IS.R call to the PMGR telling it to rip down this ?READ/?WRITE.

* If the task was not pended on a ?WTSIG or ?SIGWT to the PMGR, we check the system call double word (?TSYS) to see if it is pended on an ?IREC or ?IS.R.

* If the task is pended on and ?IREC or ?IS.R, we check to see if it is to X25 by checking if the port of the sender is X25's port.

* If the sender is X25, the Agent call NETABORT to send an ?IS.R to X25, telling it to rip down this IPC.

* If the task was pended on a system call (?TSPN is set in ?TSTAT), ?TABT is called to rip down the outstanding system call. ?TABT is a Kernel call.

* It is now time to wait for all task to finish their Agent processing on behalf of the call that is being ripped down. Clear the ?TSAB bit in ?TSTAT for the task to be aborted. Have the ?GABORT calling task go to sleep by issuing a ?REC on the mailbox in the specified Abort PC table entry.

When the task that is to be aborted issues its final Agent WRTN instruction, it jumps to the special GABORT post-processing code. This code does the following.

* The task gets its new PC from the corresponding Abort PC table entry.

* The task wakes up the ?GABORT calling task by issuing an ?XMT call on the mailbox in the specified abort PC table entry.

* The task jumps to its new PC.

Deterministic Scheduling

When the Agent wishes to explicitly cause a task rescheduling, as is the case when the Agent is going to pend the currently executing task, it makes a call to the Kernel through the Kernel's scheduling gate. There is a macro called SCHED which will do this. The following example illustrates the use of this macro.

```
SCHED
<normal return>
```

There is no input to this macro, and all ACs, except AC3, remain unchanged upon return. AC3 contains the frame pointer upon return from the Kernel. The task making a call to the scheduler will take the normal return when, if ever, it is rescheduled.

Channel Management

There are routines in the Agent to manage the two Agent channel data bases: the channel descriptor tables (CDTs) and the channel maximization table (CMT). There are two separate tables because the channel maximization table must be used for all channel activity, but the channel descriptor table only has entries for channels that are used through the Agent I/O system calls (?OPEN, ?READ, ?WRITE, and ?CLOSE).

Let us first address the routines used to maintain the channel maximization table (CMT).

Channel Maximization Table Management

As part of the multiple ring support offered by AOS/VS, channels opened by inner rings are protected from use by outer rings. To achieve this, a table of channels, and which ring they were opened from, must be maintained. This maintenance is done by three routines in the Agent.

The following routine is used to make an entry to the channel maximization table when a channel is opened.

```
LJSR ADDCMT
<exception return>
<normal return>
```

Input:
AC0 - Channel number

Output:
AC0 - Unchanged
AC1 - Unchanged
AC2 - Unchanged
AC3 - Frame pointer

ADDCMT finds the ring of the caller that made the original system call by taking his return address from the first return block on the stack. It then sets the bit corresponding to the specified channel and ring in the CMT. This bit is found using the following formula for the bit offset from the start of the CMT.

BIT OFFSET = ((RING OF CALLER - 3) * 256.) + CHANNEL NUMBER

Each time a channel is used, the ring of the caller must be checked against the CMT to determine if the calling task has access to the channel. The following routine does this.

```
LJSR CHKCMT
<exception return>
<normal return>
```

Input:
AC0 - Channel number

Output:
AC0 - Unchanged for normal return, error code for exception return
AC1 - Unchanged
AC2 - Unchanged
AC3 - Frame pointer

CHKCMT does the following.

* Validates the channel number passed as input by comparing it to the minimum and maximum possible channels. If the channel number is outside of this range, it takes the error return with the error code for "ILLEGAL CHANNEL".

* Loops through the CMT, checking the bit for the specified channel and each ring, 3, 4, 5, 6, and 7. If no bit is set for this channel in any ring, it takes the exception return with "CHANNEL NOT OPEN".

* If a bit is found set for this channel, it compares the ring of the caller to the ring specified in the CMT, If the callers ring is less than or equal to this ring CHKCMT takes a good return. Otherwise, it takes the exception return with the error "ILLEGAL CHANNEL".

When a channel is closed, the bits for this channel must be cleared from the CMT. This is done by the following routine.

```
LJSR DELCMT
<exception return>
<normal return>
```

Input:
AC0 - Channel number
Output:
AC0 - Unchanged for normal return, error code for exception return
AC1 - Unchanged
AC2 - Unchanged
AC3 - Frame pointer

The logistics of DELCMT is as follows.

* Call CHKCMT to make sure the calling task has access to the channel. If it doesn't take exception return with "ILLEGAL CHANNEL" or "CHANNEL NOT OPEN" if appropriate.

* Loop through the CMT resetting the bit for the specified channel in all rings, 4, 5, 6, and 7.

DELCMT is called in the Agent before the Agent calls the Kernel to close the channel. If the Kernel takes an error return on the close call, the Agent calls ADDCMT to put the channel back into the CMT. This is done in this manner to close a window in the channel maximization logic, and with the assumption that the Kernel rarely takes an error return when closing a channel.

Unlike the CMT, the CDT is not maintained for all system calls that open,

close, or access channels. It is only used for channels that are opened, closed, and accessed by the Agent I/O system calls ?OPEN, ?READ, ?WRITE, and ?CLOSE. Let's take a look at the routines that maintain the channel table and its CDTs.

Channel Descriptor Table Management

Each time a ?OPEN is done, a CDT is created, containing various information from the ?OPEN packet, and added to the channel table. The information from the CDT is used when a ?READ or ?WRITE is done on the channel, and the information in the ?READ/?WRITE packet says to use the parameter specified at ?OPEN time, or the parameter passed in the packet is only used at ?OPEN time.

The routine to insert a CDT into the channel table is the following.

```
LJSR INSCDT
<normal return>
```

Input:
AC2 - Address of the CDT to be inserted into the channel table
Output:
AC0 - Unchanged
AC1 - Unchanged
AC2 - Unchanged
AC3 - Frame pointer

The logistics of INSCDT is the following.

* Lock the channel table.

* Check if the channel table has been initialized yet. If not, get memory for it.

* Put the address of the CDT in the channel table. The channel table is set up such that there is one double word entry for each channel. Thus the formula for indexing, from the start of the channel table, into the table is the following.

INDEX = CHANNEL NUMBER * 2

* Unlock the channel table.

When a ?READ/?WRITE is issued on a channel, information stored in the CDT for that channel is needed to determine the ?READ/?WRITE characteristics, address of the Agent buffer, pointer position, etc. The routine to retrieve a CDT for a channel from the channel table, is the following.

```
LJSR GETCDT
<exception return>
<normal return>
```

Input:
AC0 - Host ID/Channel number

Output:
AC0 - Unchanged for normal return, error code for exception return
AC1 - Unchanged
AC2 - Address of the CDT for the specified channel
AC3 - Frame pointer

The logistics of the GETCDT routine are the following.

* Make sure that the channel is in the range of legal channels. If not, pass back the error "ILLEGAL CHANNEL".

* Check to see if the channel table is initialized yet. If it is not, no channels have been opened, so take exception return with "CHANNEL NOT OPEN".
* If the channel table is initialized, get the CDT entry for the specified channel. If the entry for the channel is zero, take the exception return with "CHANNEL NOT OPEN".

* If the CDT entry for the specified channel is not zero, pass the address of the CDT back to the user in AC2.

When a channel is closed by a ?CLOSE, the CDT for that channel must be deleted from the channel table. The following routine deletes a CDT from the channel table.

```
LJSR RELCDT
<normal return>
```

Input:
AC0 - Host id/Channel number

Output:
AC0 - Unchanged
AC1 - Unchanged
AC2 - Unchanged
AC3 - Frame pointer

RELCDT has the following logic.

* Lock the channel table.

* Call GETCDT to get the address of the CDT for the specified channel. This is done to make sure that the channel is open.

* Zero the channel table entry for this CDT.
* Unlock the channel table.

Common Return Points

There are two points that a task can leave the Agent and return to a user ring. One, AGENT.NORMAL, is for normal returns from the Agent. The other, AGENT.EXCEPTION, is for exception returns. All system call handlers do an LJMP to these routines. There are two macros defined which will do the LJMP. The first is called NORMAL.RETURN, which jumps to AGENT.NORMAL. The second is called EXCEPTION.RETURN, which jumps to AGENT.EXCEPTION.

At the common return points the Agent checks to see if the caller is returning back into the Agent, or into a user ring. If it is returning to a user ring, the Agent calls a routine DRCHK, to see if scheduling has been disabled by the user. and to pend the task if appropriate. There will be more on this routine in the section on multitasking. When the task returns from this routine, the Agent, for a normal return, adds two to the return PC and issues a WRTN. For an exception return, the Agent stores the error code in AC0, adds one to the return PC and does a WRTN.

Multitasking

Tasks are the fundamental elements of a process. Like a process, a task is an executing entity; it is a flow of control through source code, not the source code itself. A process may have up to thirty two asynchronously executing tasks. The concept of a process having more than one task executing is called multitasking.

As a program technique, multitasking offers several advantages, including

* Parallelism - Multitasking gives a program the flexibility to respond to external asynchronous events.

* Efficiency - While one task may be suspended on an I/O operation, another task can be executing.

While it is the Kernel that provides the scheduling that underlies multitasking, it is the Agent that provides management facilities for the user. All multitasking system calls are handled in the Agent.

There are three major data bases needed for the support of multitasking: the User Status Table, UST, the Task Control Blocks ,TCBs, and the Task Control Block Extenders, TCBXes. The format of these data bases was described in the section on Agent data bases. The LINK utilitity reserves space for the UST and TCBs. The TCBXes are set up during Agent initialization.

TCBs

The TCBs maintained in two queues. The queues are step up such that they may be searched with the hardware queue instructions. The queue descriptors for both queues are in the UST. One queue is a chain of active tasks, the other is a chain of inactive task. When a process is initialized there is one TCB on the active chain, with the remainder of the TCBs on the free chain. Each time a ?TASK system call is made, 'n' TCBs are dequeued from the free chain and placed on the active chain, where 'n' is the number of tasks to be initialized (ie. ?DNUM in the ?TASK parameter packet). When a task is terminated by one of the 'kill' system call, the TCB(s) for the terminated task(s) are dequeued from the active chain and enqueued on the free chain until they are activated again by another ?TASK system call.

Task Redirection Protection

Two new system calls, ?TLOCK and ?TUNLOCK, enable inner ring segment images to lock themselves from task redirection. The task redirection system calls are defined to be the ?IDGOTO, ?IDKIL, ?PRKIL, ?IDSUS, and ?PRSUS system calls. These calls either rip a task from its current line of execution or unconditionally pend the task within its current line of execution. Either event can play havoc within the system of locks that control a critical region.

An inner ring segment image can issue a ?TLOCK call to protect itself from task redirection from higher rings. Ring maximization is the protection model employed by these calls. The key value here is not the current ring of execution (current PC), but the lowest ring from which a ?TLOCK system call is currently in effect for the task to be redirected. This means that concurrent ?TLOCK calls can be issued from multiple inner rings and must be remembered separately by AOS/VS. As an example, if ?TLOCK calls have been made for one task from both rings 5 and 6, then task redirection calls will only be immediately satisfied from rings 4 and 5. An option will be provided to permit tasks to also disable task redirection calls from within their own ring.

The ?TUNLOCK system call from a given task in a given ring cancels the effect of earlier ?TLOCK calls on behalf of the same task in the same ring. Consecutive ?TLOCK calls from the same ring on behalf of a single task are legal but accomplish no further protection. ?TUNLOCK calls that do not follow ?TLOCK calls will return with an error indication. To amend our earlier example, if a ?TUNLOCK call is later issued from ring 5, then task redirection calls can now be successfully issued from rings 4, 5 or 6.

A task in a higher ring (higher than the current level of TLOCK'ing) that issues a task redirection call on behalf of a TLOCK'ed task will pend until the task issues ?TUNLOCK calls on behalf of all protected lower rings. This can become quite complicated if a task issues a ?PRKIL or ?PRSUS system call and passes an input priority that specifies more than one protected task. In that case, AOS/VS will make a note of all tasks that match the specified priority at the time of the redirection system call. If waiting is necessary, due to ?TLOCK protection, the pended redirecting task will wait for the noted tasks and those tasks only. If a redirector specifies more than one task, it is possible that the redirections will occur separately over indeterminate periods of time.

The ?TLOCK call will accept an input argument, in AC2, that specifies a double word mailbox. An input value of -1 in AC2 indicates that the caller does not wish to specify a mailbox. If the caller specifies a mailbox, it does not have to contain a zero on input to the ?TLOCK call. The mailbox address, if provided, will enable the operating system to inform a protected task that another task is trying to redirect it.

AOS/VS will place a non-zero flag in the mailbox at the time of a pended task redirection request by another task. A task executing within an inner ring segment image can poll the mailbox as desired. It might react to the redirection event by aborting its progress in the critical region in a controlled manner and then issuing a ?TUNLOCK call to help the redirection request to proceed.

It will also be possible to use the ?TLOCK system call to protect against task redirection calls issued from within the same ring. An input flag, ?TMYRING, is defined for AC0. The caller must specify the ?TMYRING flag on ?TLOCK input to cause the call to protect against task redirection calls issued from within the same ring.
On output from ?TLOCK, the operating system will set the ?TALOCK flag in AC0 if the ring was already locked for the calling task at the time of the ?TLOCK system call. The ?TMYRING flag in AC0 may also be set by AOS/VS on ?TLOCK output. AOS/VS will set that flag if the caller's ring was already locked against redirection calls from within the same ring ?TLOCK input time. (Note: ?TMYRING will only be set if ?TALOCK is also set.)

Successive ?TLOCK calls can be used to change the state of task redirection protection. At present, that state consists of the address of the double word mailbox and the ?TALOCK and ?TMYRING flag variables. When a ?TLOCK system call changes the address of the mailbox associated with the calling task in the ring, AOS/VS will place in the new mailbox the current contents of the old mailbox. Each ?TLOCK call will output the values of the state variables, as they were prior to the system call. The various state values should be saved by users until they leave their critical region.

It is the responsibility of ?TLOCK users to appropriately restore the state variables when exiting the critical region. Callers can employ a ?TLOCK call to retain task redirection protection while changing either the mailbox address or ?TMYRING variable. Callers can issue a ?TUNLOCK call if their initial ?TLOCK call indicated that task redirection protection was not already in effect on entrance to their critical region.

The ?TLOCK/?TUNLOCK functionality is necessary and reasonable for tasks that are spawned from outer rings. However, inner ring servers may also wish to initialize proprietary tasks that execute only in the inner ring. Such a task might, for example, solely issue ?IREC system calls to wait for global server termination indication. To prevent unwitting conflict with outer ring task redirection calls (principally ?PRKIL and ?PRSUS), a ring maximization rule is also applied to the task redirection calls. The rule states that a task created within an inner ring can only be redirected by a call from that ring or lower.

Agent/Exec Interface

Overview of ?EXEC processing

Certain CLI commands cause a 16-bit ?EXEC packet to be built by CLI, which then makes a 16-bit ?EXEC call. When the Agent gets such a call, it processes it by:

* The 16-bit ?EXEC handler (called NEXEC or the Narrow Packet Converter) builds a 32-bit ?EXEC packet and issues a 32-bit ?EXEC call.

* The 32-bit ?EXEC handler (called AEXEC) builds a message and sends it to Exec via the ?IS.R call.

* Exec receives the call, processes it, and returns the message to the Agent. Certain requests will cause Exec to return information in the message packet.

* The 32-bit ?EXEC handler updates the 32-bit ?EXEC packet as needed, and returns to the caller (in this case, the caller is the 16-bit ?EXEC code path).

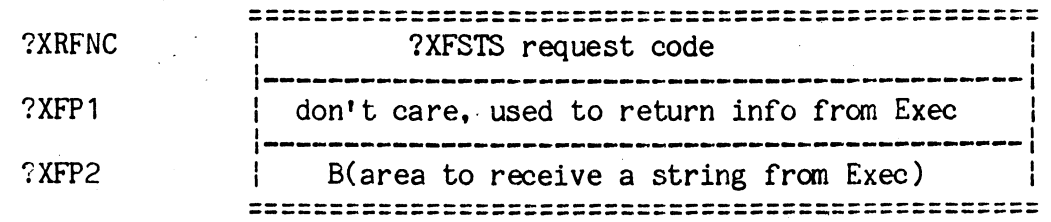
* The 16-bit ?EXEC handler updates the 16-bit ?EXEC packet as needed, and returns to the caller (CLI).

Details of the processing which occurs in the 16-bit and 32-bit ?EXEC call handlers can be found in later sections.

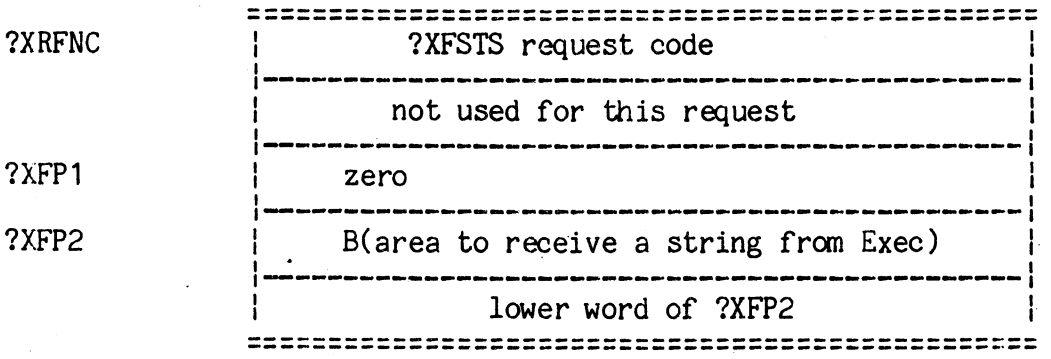
A Simple Example

Suppose that a user logs on to @CON4, and types the command "WRITE [!CONSOLE]". CLI issues a ?EXEC call for function ?XFSTS, which returns status flags and (optionally) the console or stream name.

The 16-bit ?EXEC packet built by CLI looks like this:



When the CLI issues a ?EXEC call, the Agent dispatches to the NEXEC Narrow Packet Converter routine to translate the packet into the corresponding 32-bit packet and reissue the ?EXEC call as a 32-bit user. The 16-bit byte pointer to CLI's string buffer is expanded to 32 bits. The packet which is constructed by NEXEC looks like this:

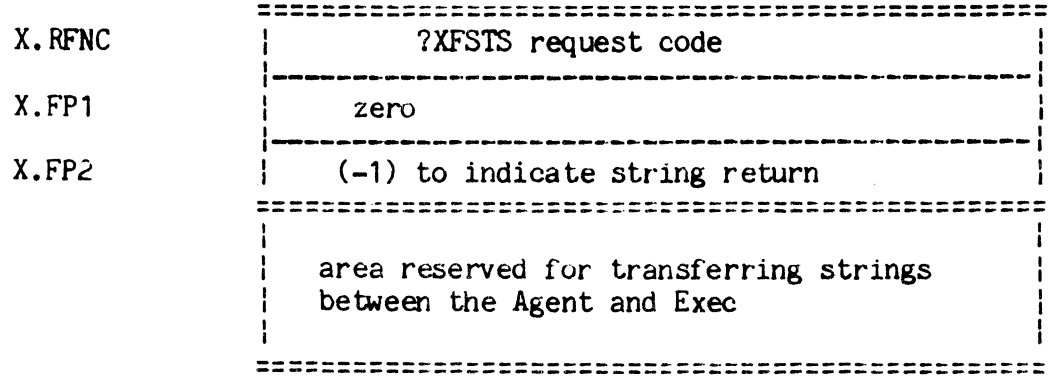


The 32-bit ?EXEC handler, called AEXEC, constructs a message to be sent to Exec. This message is called "EXC" in the rest of this document. For the ?XFSTS request, the EXC message looks very much like the ?EXEC packet, except:

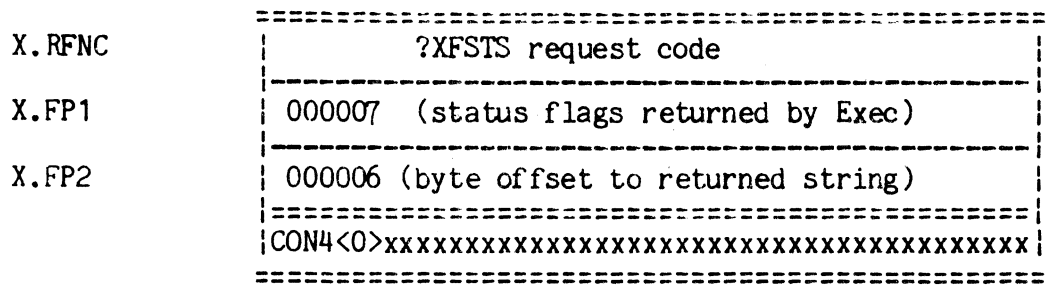
* The ?XFP2 bytepointer becomes a byte offset from the beginning of the EXC message into the string partition

* The packet includes a string area of sufficient size to contain all strings to be transferred between the Agent and Exec.

Therefore, the EXC message has the following format:



After this message is built, it is sent to Exec via the ?IS.R call. Exec processes the request, and the packet now looks like this:



Note that this request returns only one string, and uses no strings as input. Therefore, the byte offset to the string returned by Exec is 5. This offset is relative to the beginning of the message, by convention. If no string was requested to be returned (X.FP2 had 0 when the ?IS.R call was made), the string area (and X.FP2) is not changed by Exec.

The contents of the single word X.FP1 (flags word) are returned to the 32-bit ?EXEC packet into offset ?XFP1 of the packet.

If the user requested that the console or stream name be returned int the user's string area, the contents of the string area of the EXC message are moved into the user's string area, as indicated by the ?XFP2 bytepointer in the 32-bit ?EXEC packet. Assuming there were no errors detected by the Agent or by Exec, a normal return is made.

Since the 32-bit ?EXEC caller was the 16-bit ?EXEC handler, it receives control when the 32-bit processing has completed. The ?XFSTS request returns information to the user; therefore, the contents of the flags word ?XFP1 in the 32-bit ?EXEC packet are moved into the flags word ?XFP1 in the 16-bit ?EXEC packet.

Since the 32-bit handler has moved the string (if any) into the users string buffer, it does not necessary to copy any other action, so a return is made to the 16-bit caller (?IS.).

A Complex Example

The commands to CLI which cause the biggest amount of work to be performed by the Agent are those which submit requests to any queue using the ?XFSUB request. The reason for all the extra work is that the packet format depends on the queue type for the submit request; therefore, the Agent has to submit a ?XFQST request to Exec to find the queue type before it can translate the packet (note that the queue names are user-specified at system generation time, so the Agent cannot itself infer the queue type from the contents of the queue name string).
In this example, the CLI user gives the command:

```
QSUBMIT/QUEUE=PRINT/NOTIFY/DEST=ZIPPY :UDD:TRAPPER:YOW.CLI
```

CLI builds a ?EXEC packet which looks like this:

?XRFNC	?XFSUB request code
?XTYP	Bytepointer to queue name string ("LPT<0>")
?XDAT	zero (enqueued date will be returned here)
?XTIM	zero (enqueued time will be returned here)
?XLMT	zero (limit will be returned here)
?XPRI	-1 (default, since user didn't specify)
?XFGS	000010 (flags - /NOTIFY for this example)
?XSEQ	sequence number (returned)
?XRES	zero (reserved)
?XFBP	zero (default forms name)
?XPBP	Bytepointer to ":UDD:TRAPPER:YOW.CLI<0>"
?XAFD	zero (no /AFTER= switch was given)
?XAFT	zero (no /AFTER= switch was given)
?XXW0	zero (default, since user did not specify)
?XXW1	zero (default, since user did not specify)
?XXW2	zero (default, since user did not specify)
?XXW3	Bytepointer to "ZIPPY<0>"

CLI issues the ?EXEC call, and the Agent dispatches to NEXEC. The Agent recognizes the packet to be a ?XFSUB request, but needs to know the queue type in order to determine the packet format. It calls the WHICHQ routine to issue a 32-bit ?XFQST request to Exec to find the queue type, using a packet like this:

?XRFNC	?XFQST request code
	not used for this request
?XFP1	zero
?XFP2	Bytepointer to "PRINT<0>"
	lower word of ?XFP2

WHICHQ receives the queue type, and translates the queue type to the associated request type (using the QTYPE table), and so returns ?XFLPT as the packet type. The NEXEC routine constructs the 32-bit packet, which has the same format as the 16-bit packet except that bytepointers are 32 bits long instead of 16, and issues a 32-bit ?EXEC packet.

AEXEC (the 32-bit ?EXEC call handler) receives the packet - and the request type is still ?XFSUB, so it ALSO needs to call WHICHQ to find the packet type corresponding to the queue type name. The packet built by WHICHQ and sent to EXEC is the same ?XFQST packet illustrated above. When AEXEC determines that the packet type is ?XFLPT, it constructs an EXC message for that packet type (but preserves the function code as ?XFSUB):

X.RFNC	?XFSUB request code
X.TYP	000040 (byte offset to "PRINT<0>")
X.DAT	zero (will be returned by Exec)
X.TIM	zero (will be returned by Exec)
X.LMT	zero (will be returned by Exec)
X.PRI	-1 (default queue priority will be used)
X.FGS	000010 (/NOTIFY flag bit for this example)
X.SEQ	zero (will be returned by Exec)
X.FBP	zero (default forms name)
X.PBP	000046 (offset to ":UDD:TRAPPER:YOW.CLI<0>")
X.AFD	zero (no /AFTER was specified)
X.AFT	zero (no /AFTER was specified)
X.XW0	zero
X.XW1	zero
X.XW2	zero (default number of copies)
X.XW3	000073 (offset to "ZIPPY<0>" in string area

|PRINT<0>:UDD:TRAPPER:YOW.CLI<0>ZIPPY<0>xxxxxxxx|
|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|

Note: The diagram above does not attempt to show which word each character of the strings will be in, but rather illustrates the order and position of the strings. The "x" characters represent unused bytes.

Notice that the byte offset to the first string ("PRINT<0>") is 40, not zero. The reason is that the offset is the number of bytes from the beginning of the message packet. not from the beginning of the string area. This convention was adopted for ease of implementation.

The EXC message is sent to Exec via the ?IS.R call. Exec processes the request and returns the message. The returned message is unchanged, except for the following fields: X.DAT, X.TIM, X.LMT, X.PRI, and X.SEQ. These singleword values are returned to the 32-bit ?EXEC packet (offsets ?XDAT, ?XTIM, ?XLMT, ?XPRI, and ?XSEQ) and AEXEC returns to its caller (NEXEC).

NEXEC updates the 16-bit packet with the values returned by AEXEC and returns to its caller (CLI).

CLI finishes by giving the message "QUEUED, SEQ=xxx, QPRIO=xxx", using the values returned by the ?EXEC call.

Data Structures and Tables for Agent/Exec Interface

The Agent/Exec interface is largely table driven. The way in which a packet is translated from a 32-bit ?EXEC packet to the EXEC message is controlled by the following tables:

* The FUNCTION TABLE: Used by NEXEC and AEXEC routines, it gives general information about the packets, indexed by the ?EXEC packet function code.

* The VALIDATE PACKET TABLE: Used by AEXEC routine, it describes the source and destination offsets and the entry type for each field to be translated from ?EXEC packet format to EXC message. The entry type indicates how the entry will be converted.

* The UPDATE PACKET TABLE: Used by AEXEC routine, it describes the source and destination offsets and the entry type for each field of information to be returned to the user from the EXC message returned by EXEC.

* The QUEUE TYPES TABLE: used by WHICHQ to determine the type of queue from the queue name string.

* The STRING LENGTH TABLE: for string entry types (filename, VOLID, password, etc), gives the maximum length of the string.

* The VALIDATION DISPATCH TABLE: used by AEXEC to dispatch to the appropriate Validation/Move routine for one field in the 32-bit ?EXEC packet. Indexed by entry type. Located in A.32.14.SR.

* The UPDATE DISPATCH TABLE: used by AEXEC to dispatch to the appropriate Update routine to return information for for one field. Indexed by entry type. Located in A.32.14.SR.

* NPC.DISPATCH table: used by NEXEC to dispatch to the routine to convert a packet from 16-bit to 32-bit ?EXEC format. Indexed by ?EXEC function code. Located in A.16.14.SR.

* The "BIG TEMPORARY AREA" is used by AEXEC to store temporary variables in processing one 32-bit ?EXEC request. Defined and allocated in A.32.14.SR.

The Function Table

The function table is used to look up information for the specified request function from the ?EXEC packet offset ?XRFNC. Used by NEXEC and AEXEC routines, the Function Table gives the length of the packets, flags, and specifies the packet table entries to be used.

The function table contains one entry for each function code in the range ?XFMIN to ?XFMAX (Exec minimum/maximum function codes). It is indexed by ?EXEC function code, and is located in the module XTABLES.SR.

The following information can be found in the function table:

* Length of the USR packet (the number of words in the caller's 32-bit ?EXEC packet)

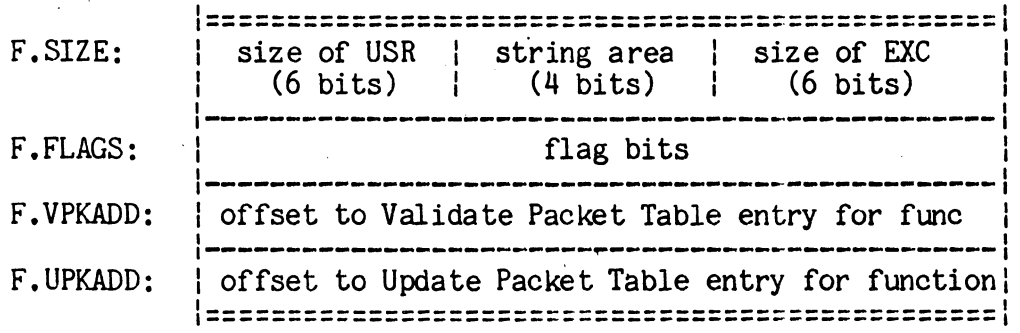
* Space to be allocated for the EXC message, including the space required for the string partition of this message.

* Flags for Internal-Only Function, Undefined Function, and a flag to indicate that information should be returned to the caller's packet after EXEC returns its message.

* Offset from the beginning of the Function Table to the validate packet table entry corresponding to this function code.

* Offset from the beginning of the Function Table to the update packet table entry corresponding to this function code.

Each entry has the following format:



The "size of USR" field gives the number of words required for the 32-bit ?EXEC packet as defined in PARU.32.SR.

The "string area" of F.SIZE gives the number of 64-word blocks which will be needed for the string area of the EXC packet for a given function. Note that the "string area" and "size of EXC" fields of F.SIZE, taken as a 10-bit integer, give the total number of words required for the EXC packet.

F.FLAGS gives information regarding a function:

* Bit F.BUND indicates that this function code is undefined.

* Bit F.BINT indicates that this is an "internal" function: such functions are restricted to calls made by the Agent/ Ghost. Users who request an internal function get ERXUF.

* Bit F.BUPD indicates that either the packet or the user's strings should be updated by the Agent/Ghost on receiving the return message from Exec.

The Packet Tables have varying numbers of entries depending on the function code. Also, some functions share the same Packet Table entries. F.VPKADD gives the word offset from the beginning of the Validation Packet Table to the beginning of the entries for validating/moving the packet for a function. F.UPKADD gives the word offset from the beginning of the Update Packet Table to the beginning of the entries for updating a packet for a function.

The F.T macro is used for generating the entries for the Function Tables. The macro is kind enough to check for certain obvious error conditions; if you notice an assembler diagnostic for a source line which begins "**ERROR -", it may be the macro's way of informing you that you have a damaged brain.

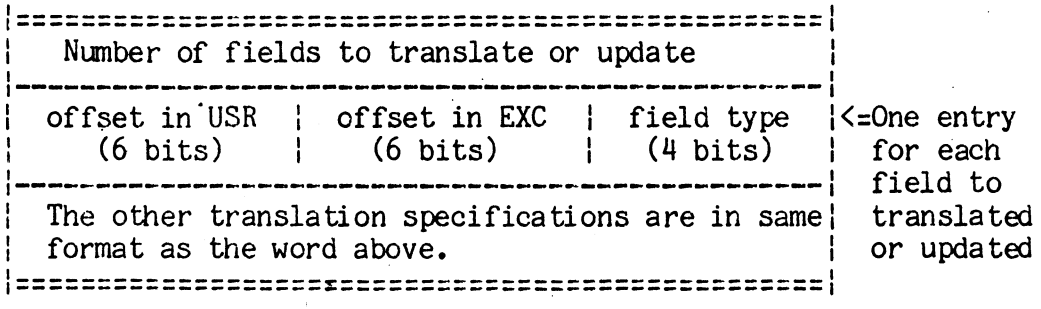
The Packet Tables

There are two packet tables for each function which give the locations and entry types of the fields for validating ("preprocessor") and for updating ("postprocessor") a packet.

The Validate/Move Packet Table, which is defined in XTABLES.SR, gives the information for each entry to be translated from the user's ?EXEC packet to the EXC message. The Validate/Move Packet Table has variable length entries and is not indexed; in order to find the starting address of an entry, the Function Table entry offset F.VPKADD is used to calculate the starting address of the packet table entry.

The Update Packet Table is defined in XTABLES.SR. It is used by AEXEC to find out what fields of information should be returned from the EXC message to the user. Like the Validate/Move Packet Table, the Update Packet Table is not indexed: the Function Table entry offset F.UPKADD is used to calculate the starting address of the packet table entry.

The entries of the Packet Table vary in length depending on the number of fields to be translated for a function. The first word of each Packet Table entry gives the number of translations to be performed. Following this is one word for each translation which indicates the location of the field in the USR and EXC packets, and a field type code which indicates what kind of translation is to be performed.



Note that several functions may share one Packet Table entry if they use the same fields in the same way. Thus you must be careful when changing this table to ensure that you are not affecting another function.

The V.P.T and U.P.T macros are used to define the structure of each type of packet. The first entry for each packet has "<" as the first argument. This indicates to the macro that it should generate a header entry specifying the number of fields to be translated, i.e. the number of entries which follow the header. The other entries specify the word offset from the beginning of the USR packet to the entry to be translated, the word offset from the beginning of the EXC packet to the location of the translated parameter, and the entry type. The entry type tells the Preprocessor how to perform the translation.

Since the x.P.T macros generate rather complex entries, it performs some

validation of its own within the macro; if you see an assembler diagnostic for a line which starts with "**ERROR - ", it is the x.P.T macros' own way of telling you you have brain damage.

Please note that the Validate and Update routine packet tables must have the same format since they are dispatched in the same way.

The Queue Types Table

The Queue Types Table associates a queue type (an integer value, as returned by Exec) with the Exec function code to access that queue. For example: queue type QTLPT is associated with the Exec function ?XFLPT.

The queue types tables is used by the WHICHQ subroutine. It is located in XTABLES.SR, and is indexed by queue type number.

Entries for this table is generated by the Q.T macro. The only argument to this macro is the queue type (like LPT, FTA, and so on). The end of the Queue Types Table is defined by using the Q.T macro with no argument.

The String Length Table

The String Length Table associates string entry types (like filename, pathname, VOLID, VOLID list, etc.) with the maximum length allowed for that string type. For example, a filename (entry type E.FNM) may not exceed FNM.LEN characters including the delimiter. It is used by the Validate/Move and Update routines, and is indexed by entry type.

The string length table is generated thru the use of the U.L macro. The only argument is the string type (like FNM for filename). To define the end of the string length table, use the U.L macro with NO ARGUMENTS.

Routines for the Agent/Exec Interface

The routines for the Agent/Exec Interface include:

* NEXEC: sometimes referred to as NPC (Narrow Packet Converter) routine, it translates 16-bit ?EXEC packets into 32-bit ?EXEC packets.

* AEXEC: the 32-bit ?EXEC Packet Translator, which translates 32-bit ?EXEC packets into the message format which EXEC expects, sends the message, and returns any information to the user's packet.

* VALIDATION/Move routines: these are used by AEXEC to translate one field of the 32-bit ?EXEC packet to the corresponding field of the EXC packet. For string fields, the bytepointer is translated to a byte offset into the string partition of the message, and moves the string into the string partition.

* UPDATE routines: these are used by AEXEC to return information, one field at a time, from the message returned by EXEC to the caller's packet.

* The WHICHQ routine: for ?XFSUB and ?XFOTH functions, this routine finds out what flavor of submit packet has been requested, according to the queue name string. The function code implied by the queue name string is returned.

NEXEC: the Narrow Packet Converter Routine

The NEXEC routine converts 16-bit format ?EXEC packets into 32-bit ?EXEC packets. It is the entry point to which transfer is controlled when a 16-bit process issues the ?EXEC call from ring 7.

The conversion involves changing 16-bit bytepointers to 32-bit bytepointers, and relocating the parameters according to the 32-bit packet format. Following this conversion, a 32-bit ?EXEC call is made to construct the EXC message and send it to EXEC. When the 32-bit ?EXEC call is completed, strings and parameters are updated as required by the function.

NEXEC Logic

The first word of the packet is validated so that the Agent can safely access the function request code. Using the function code, lookup the length of the 32-bit packet in the function table, and allocate some Agent memory in which to build the 32-bit packet. Finally, dispatch to a routine to convert the remainder of the packet, make a 32-bit ?EXEC call, and return information as necessary.

If the function code is ?XFSUB or ?XFOTH, the remainder of the packet format depends on the queue type, which is a user specified string. Since the string contents to queue type conversion is system generation dependent, the WHICHQ routine is called to ask EXEC to convert the queue name string to a queue type number. Then WHICHQ looks up the function code for this queue type number in the Queue Types Table.

In the case where the ?XFSUB request is made for the batch queue, the function code remains ?XFSUB, since there is no specific request code for the batch queue submit function.

The NEXEC Conversion Routines

These routines may be shared by functions which have identical packet formats. For example, the ?XFHOL, ?XFUNH, and ?XFCAN (queue hold, unhold, and cancel) are able to share the same routine.

For submit-style packets (FTA, SNA, HAM, LPT, PTP, PLT, SUB, and OTH) a check is performed to see if the user-specified function code was ?XFOTH, since this function can be dispatched to any of the submit packet handlers. If this is the case, the username and password are also converted. The CHECK.OTH routine, local to the module, performs this function.

The routines which perform packet conversion for the various request codes are macro-driven, so that the source code for this part of the module is essentially table-driven. Thus, a new function request code can be added easily by the use of these higher level macros.

Macros for Packet Conversion Routines

The R.SETUP and W.SETUP macros contain code which is common to all of the packet conversion routines. They perform packet area validation and set AC's for use by the other macros.

The macro R.SETUP checks for read access and sets up AC's for converting the fields of the packet. The macro W.SETUP is identical, except it checks for write access. Use W.SETUP when will need to return info to the caller.

The first argument for the R.SETUP and W.SETUP macros is the function code. For example, the EXEC function ?XFQST uses QST as the first argument to the W.SETUP macro. The second argument for the setup macros is the label of the error handler routine.

The other macros are used by NPC to convert each field of the 16-bit ?EXEC packet to a 32-bit ?EXEC packet. These macros assume that the AC's have been set up properly by the x.SETUP macro: AC0 contains zero, AC1 (used as a work accumulator) can have any value, AC2 has A(32-bit packet), and AC3 contains A(16-bit packet).

The only argument is the name of the field to be translated. For example, to translate the singleword at offset ?XFP1 of the 16-bit ?EXEC packet, code "N.MOVE XFP1".

N.MOVE "Narrow MOVE": Move a singleword from the narrow packet to the wide packet.

NW.CONVERT "Narrow to Wide CONVERT": Get a singleword from the narrow packet and sign extend it, then store it as a doubleword in the wide packet.

BP.CONVERT "Byte Pointer CONVERT": Get a singleword from the narrow packet.

If it is zero or -1, sign extend it to 32-bits; otherwise, use 170000 for the top word to make it into a 32-bit byte pointer. Store the doubleword into the 32-bit packet.

N.ZERO "Narrow ZERO": Store a 16-bit zero into the specified W.offset

The N.RETURN ("Narrow RETURN") macro is used to return information from the 32-bit packet received from EXEC to the 16-bit caller's packet. It expects that AC2 has the address of the 32-bit packet, and that AC3 has the address of the caller's 16-bit packet. It takes a word from the 32-bit packet and returns it to the caller via the caller's 16-bit packet.

Note that is not necessary to update strings, since they are updated by AEXEC in the area specified by the bytepointer.

AEXEC: The 32-bit ?EXEC Packet Translator

AEXEC is the dispatch point in the Agent for ?EXEC calls from a 32-bit process (or from inner ring servers, including the Agent, of a 16-bit process). The highest level part of AEXEC performs initial validation and allocates memory for the new packet, then calls the validate/move routines to build the EXC format packet from the 32-bit ?EXEC packet. Next, an ?IS.R packet is built; the address of the EXC message is used for both of the ?IPTR and ?IRPT pointers. When EXEC answers, information is returned to the user (as required for the function) via the update routines.

AEXEC Initialization and Prevalidation

The initialization for the Agent/Exec interface is simply getting the memory for the big temporary area and saving the pointer on a stack temporary doubleword.

Prevalidation checks for certain brain-damaged situations: making sure that EXEC is up, ensuring that we have access to the function code of the ?EXEC packet. making sure that the function code is within a legal range, checking that the specified function code is defined, and bouncing any internal-only requests issued from a ring other than the Agent ring. Any of these problems will save us a lot of work, because we won't have to borrow memory for the packets, construct them, or send the IPC to Exec.

* Get memory for big temp by calling AGMEM.

* Validate the ?EXEC packet for read access of one word (the function code) only.

* Issue the ?ILKUP to get EXEC's port. If this fails, return an error to the user that EXEC isn't available.

* Check that the function code is within the range of minimum to maximum defined functions. If not, return ERXUF (EXEC Unknown Function) to the user.

* Store the function code in the big temp area word FUNCTION, find the address of the function table entry corresponding to the function, and store that in FT.ENTRY.

* If the "Undefined" (B.BUND) bit of the flags word in the function table entry is set, return ERXUF to the user.

* If the "Internal-only" (B.BINT) bit of the flags word is set, check the original caller's ring to make sure that the call came from the Agent. Otherwise, return ERXUF to the caller.

AEXEC10: Special Test for ?XFOTH Function

If the requested function is ?XFOTH, examine the superuser status via ?SUSER. ?XFOTH is only legal from superusers; if the process is not a superuser. return ERPRV ("Caller Not Privileged for This Action) to the caller.

The ?XFOTH function is intended for use by STACKER and SFTA (networking file transfer process); these two programs are normally proc'ed as superusers.

We have to do this to prevent malicious users from using the function to steal passwords for other users. If they have superuser privilege anyway, there are easier ways to steal passwords.

AEXEC20: Get Memory for Packets

* Get the word from the function table entry which specifies the lengths of the user packet (USR), the message packet which will be sent to Exec (EXC), and the length of the string area portion of the EXC packet.

* Using masking and shifting instructions, isolate these values, and store them in the big temp area variables USR.LEN and EXC.LEN (note that EXC.LEN includes the length of the string area).

* Validate the caller's packet for the entire length USR.LEN, read access. When we validated the packet before, we only validated the first word so that we could look at the function code.

* Add USR.LEN and EXC.LEN to find the total memory needed for the packets. Call AGMEM to get the memory.

* Store the address of the borrowed memory (from AGMEM) as the beginning of the EXC packet, in variable EXC.PKT of the big temp area.

* Convert the EXC.PKT address to a bytepointer and save it in EXC.PKTB. This will be used to calculate the byte offset from the beginning of EXC to a given string in the EXC string area.

* Isolate the length of the string area only from the total length of the EXC packet. Calculate a bytepointer to the beginning of the EXC string area and save it in EXC.STR.

AEXEC30: Get Packet Table Entries

* For functions ?XFOTH and ?XFSUB, call WHICHQ to correct the function code (but do NOT change FUNCTION in the big temp area).

* Calculate the address of the function table entry for the function code (this will have changed if the function code was corrected as per the above item).

* Using the function table entry determined above, get the address of the validation/move packet table entry for this function and save it in VPT.ENTRY of the big temp area.

* Get the address of the update packet table entry for this packet. and store it in UPT.ENTRY in the big temp area.

AEXEC40: Build EXC Packet via Validate/Move Routines

Set up a loop which will have one iteration for each entry in the validation packet table entry. Use the top of the stack for the loop counter.

For each entry in the validation packet table,

* Calculate the source address for the parameter to be validated/moved by adding the USR offset in the VPT entry to the address of the USR packet.

* Calculate the destination address for the parameter to be validated/moved by adding the EXC offset in the VPT entry to the address of the EXC packet.

* Perform the action specified in the VPT entry by dispatching to the appropriate validation/move routine.

* Decrement the loop counter at the top of stack and reiterate if there is more work to be done.

AEXEC50: Finish Building EXC Packet for ?XFOTH Only

If the function code is ?XFOTH, call the V.MST routine once each for the username and password bytepointers. We do this manually to avoid having an additional set of batch-type entries for each possible operation implied by ?XFOTH.

AEXEC60: Talk to EXEC via ?IS.R

* Build the ?IS.R packet. Use the portion of the big temp area which is allocated for the ?IS.R packet. The pointer to the message (?IPTR) will simply have the address of the EXC packet. which is the message to Exec. The Exec port number was established earlier via ?ILKUP in prevalidation.

* Send the message off to EXEC via ?IS.R call.

* If ?IS.R takes the error return, send this error back to the ?EXEC caller.

* Check for errors returned by Exec in the ?IUFL word of the ?IS.R packet. and return it to the user if nonzero.

The label AEXEC65, which is the address of the ?IS.R call, is defined as an external so that it is easy to set a breakpoint for messages to be sent to EXEC.

AEXEC70: Return Info to User's Packet via Update Routines

Validate the caller's ?EXEC packet for write access. Earlier we validated the packet for read access only.

Note that the info will be returned to the USER'S ?EXEC PACKET, not the USR packet. The USR packet is just our copy of the user's ?EXEC packet.

Set up a loop to return information from the EXC packet to the user. If there is any information to be returned, use the update packet table entry for the (possibly corrected) function; we calculated the address for this entry earlier. Use the top of stack for the loop counter; the loop count is taken from the number of entries in the update packet table.

For each parameter to be returned to the user,

* Calculate the source address for the parameter to be updated by adding the EXC offset in the UPT entry to the address of the EXC packet.
* Calculate the destination address for the parameter to be updated by adding the USR offset in the UPT entry to the address of the USR packet.

* Perform the action specified in the UPT entry by dispatching to the appropriate update routine.

* Decrement the loop counter at the top of stack and reiterate if there is more work to be done.

We're done now. Branch to the normal return point.

AEXEC90: Normal and Error Return Points

AEXEC90 is an error return point which will return ERXUF ("EXEC Unknown Function") to the caller.

AEXEC91 returns ERXNA ("EXEC Not Available") when the ?ILKUP fails.

AEXEC98 is the common error return handler. It assumes that the error code is in AC0. Memory which was borrowed for the packets and for the big temp area is returned before taking the caller's exception return.

AEXEC99 is the normal return handler. It also returns memory which was borrowed for the packets and big temp area, and then takes the caller's normal return.

If the free memory routine (AFMEM) fails, the process will be terminated with "SYSTEM RING TRAP, EXECUTE PROTECTION", which is the way the Agent "panics".

The Validate/Move Routines

The Validate/Move routines have essentially the same input parameters, and depend heavily on the "big temporary area".

For the routines which access strings, the byte pointer to the string is validated; then the string is move into the string area, starting with the next available byte.

The Validate/Move routines always move information from the USR packet (the Agent copy of the 32-bit caller's ?EXEC packet) to the EXC packet (the message to be sent to Exec).

V.NUM Move a singleword. No validation is performed.

V.STR Validate an optional string (pointer can be zero, the default value) and check for illegal characters.
V.MST (Must be a STring). Validate a required string (the pointer must not be zero) and check for illegal characters.

V.VLS Validate a Volume ID list. Pointer must be nonzero.

V.SPC Special Case Validation

V.IRS Indicate Return String

The V.NUM Routine

This routine moves a singleword value (numeric or flags) from the USR packet to the EXC packet. No range validation is performed by the Agent.

The V.STR Routine

This routine moves an optional string from the user's space into the EXC string area, and places a byte offset to the EXC string into the EXC parameter area.

The string is optional for this routine: if the bytepointer to the string is zero, no string will be moved, and the parameter for this string will be zero instead of a byte offset into the string area.

This routine checks for a nonzero pointer, in which case it calls the V.MST string to do the real work of string moving.

The V.MST Routine

This routine validates and moves a required string.

Validate the bytepointer to the string (no default is allowed). Check for illegal characters for this entry type. Check that the string length does not exceed the maximum for that entry type.

If the string is valid, get the byte offset of the next available byte of the string area and store it into the string parameter. Move the string into the string partition of the EXC message, starting with the next available byte. Update the offset to the next available byte for the next use of the string area.

The V.VLS Routine

This routine validates and moves a Volume ID list.

Validate the pointer to the string, using ARVBPD (Read Validate Bytepointer for string with Double delimiter). Check the length of each Volume ID in the list to ensure that they are all six or less characters long (seven, including the delimiter). Ensure that the entire string does not exceed the maximum number of characters allowed for a Volume ID List.

The V.SPC Routine

This routine handles a "special case" for Exec requests ?XFHOL (hold a queue entry), ?XFUNH (unhold), and ?XFCAN (cancel).

The doubleword parameter at offset ?XFP2 of the packet is to be interpreted as a bytepointer only if the singleword parameter at ?XFP1 has the value (-1). In that case, the V.MST routine is called to move a jobname into the string area.

Otherwise, the FIRST WORD at offset ?XFP2 is interpretted as a job sequence number, and the word at ?XFP2+1 is ignored.

The V.IRS Routine

This routine is used when a user has specified an area to be used to return a string from the ?EXEC call. It validates the bytepointer for write access before a call to Exec is made, and indicates to Exec that a string is to be returned for this parameter.

The reason for validation before calling Exec is to avoid an IPC in the case where the pointer is invalid, or points to an area to which we do not have write access.

By convention with Exec, if the parameter is zero, no string will be returned. Any nonzero value will cause Exec to move a string into the

string portion of the EXC packet and save the byte offset to that string in the parameter. The nonzero value used to signal Exec to return the string is (-1).

The Update Routines

The Update routines are very similar to the Validate/Move routines, except that they are used to move information from the EXC packet (which was returned by Exec) back to the user's packet or string buffers.

For the routines which access strings, the byte pointer to the string buffer is validated for write access. If valid, the string is moved from the EXC string area to the user's string buffer.

The Update routines always move information from the EXC packet (the message received from Exec) to the user's 32-bit packet or the user-specified string buffer.

U.NUM Return a singleword.

U.MST Return a string.

The U.NUM Routine

This routine returns a singleword value (numeric or flags) from the EXC packet to the USR packet.

The U.MST Routine

This routine returns a string from the EXC string area into the user-specified string area.

Validate the user's bytepointer to the string buffer for write access. If the pointer is valid, move it into the user's buffer and terminate the string with a NULL character.

WHICHQ: Routine to Determine Implied Function Codes

The WHICHQ routine is used to determine the request function codes which applies to ?XFSUB and ?XFOTH packets from the queue type specified by the caller. This implied function code is important because it specifies the format of the extender words of the submit-style packets (parameters ?XXW0 thru ?XXW3).

The WHICHQ routine is called by both the 16-bit and 32-bit ?EXEC handlers (NEXEC and AEXEC).

The WHICHQ routine follows these steps:

* Validate the bytepointer to the queue type name string, using the Agent routine ARVBPX. If invalid, return the error code to the caller.

* Create a 32-bit ?EXEC packet for a ?XFQST request. The bytepointer to the queue name string, as specified in the submit packet, is passed in ?XFP2.

* Issue a 32-bit ?EXEC call for queue status.

* Determine the implied function code, using queue type number which is returned by Exec to index into the queue types table.

* Return the implied function code to the caller.

Comparison to the Old Agent/Exec Interface

The Old Agent/Exec Interface (AOS/VS Revision 1.40 and earlier) is different from the current interface in that it merely validated the pointers in the packets, and sent the (32-bit) packet to Exec. It did not send the strings to Exec. This approach required Exec to issue the ?MBFU (Move Bytes From User) once for each string to be fetched from the user space, and the ?MBTU (Move Bytes To User) once for each string to be returned to the user.

The calls to transfer bytes between processes are expensive; further. they required the ability to transfer bytes between a ring 7 caller (EXEC) and a potentially inner ring of the user. Starting with AOS/VS Revision 1.50, the byte transfer calls were restricted by the rules of ring maximization, so that such transfers became illegal.

The old interface did all packet translation by specialized code in the Agent. The addition of a new function request type required the addition of new code in the Agent specific to the packet. In the new interface, the packet format is controlled by the tables in the global module XTABLES.SR (shared by EXEC, the Agent, and the Ghost), so that new functions could be implemented by only adding new entries to the table. Therefore, the main advantages to the new interface are:

* Better performance for Exec, which no longer has to issue as many expensive IPC-type calls

* Reduced IPC traffic on the system, resulting in improved system-wide performance

* Flexibility of new functions, and the ability to add new functions to both AOS and AOS/VS by changing one module

Another difference was the addition of the ?XFQST function so that the Agent could determine the packet format for SUBMIT type requests. Formerly. the parameters ?XXW0 thru ?XXW3 were treated identically between packet types, so that (for example) the page number was treated as a bytepointer. This worked in most but not all cases; the limitations could not be resolved. With the new function, the Agent is able to translate packets properly.

Resource Management Agent

As of revision 2.00 of AOS/VS there has been available a new piece of the Xodiac™system called RMA. The Agent performs the deflection of the RMA calls from the user to the remote system. The addition of this function added another layer of complexitiy to the Agent.

RMA Databses Used in the AGENT

As a result of the inclusion of the RMA interface in the AGENT several new databases were added to the AGENT. They are:

* IRMA databse - pointers to all other RMA databases

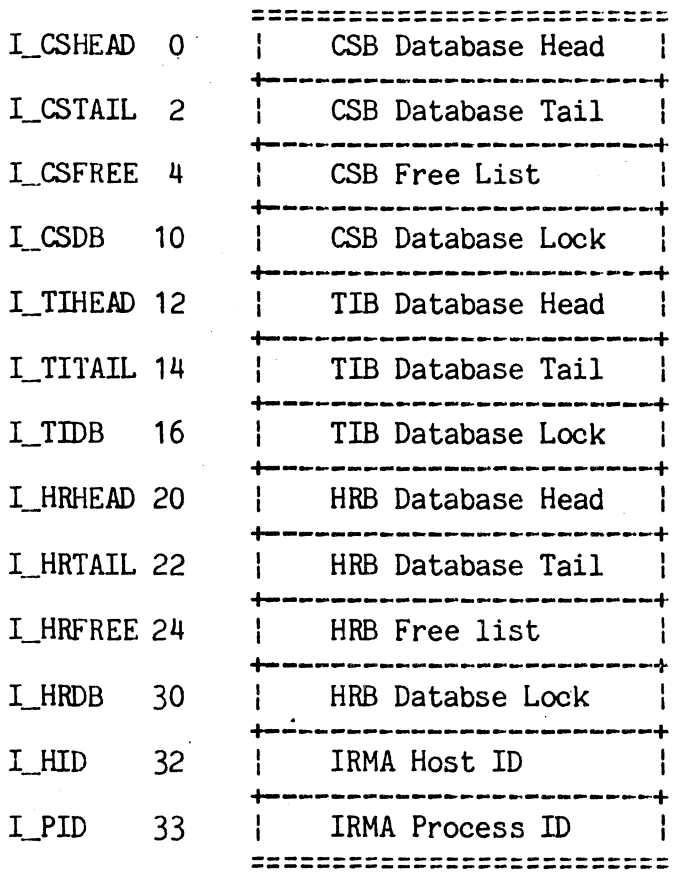
* CSB database - connection state block

* TIB database - task information block

* HRB database - host record block

All of the above databases are used by the Agent in implementing the RMA interface to Xodiac. Each database has a link in the IRMA datbase. These pointers are to the head and tail of each database. A lock word is also kept for each database. The CSB and HRB dabase also have a list of free elements which can be used by RMA.

The format of the IRMA database is as follows:



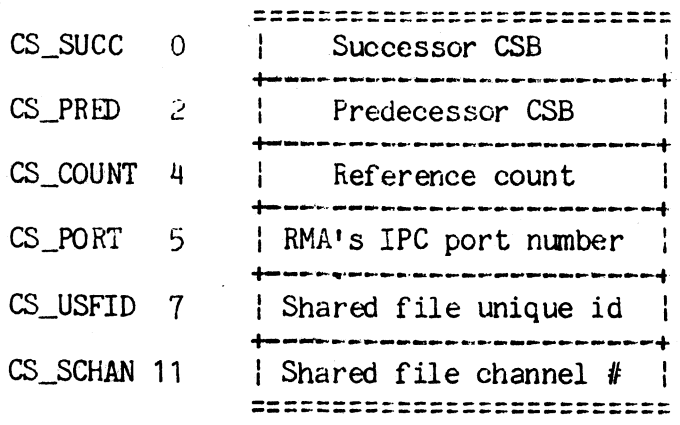
IRMA Database

The IRMA data structure is allocated and initialized at "AINIT" time by the IRMA (AGENT) initialization code. Free lists exist because more than one element is allocated at a time to reduce the fragmentation of AGENT unshared memory.

Connection State Block

The Connection State BLock is associated with a connection to the URMA process. It is built when a deflection to RMA occurs. This is the connection made when there is a system call to pocessed accross the network.

The format of the CSB database is as follows:



CSB Database

IRMA Task Information Block

The Task Information Block is the basic operating structure for tasks in IRMA. The TIB locates information about the Global RMA server. If TI_CSB is zero, there is no RMA process known to this TIB. If TI_CSB is non-zero then the CSB contains important information about the RMA server, including shared file information obtained by the first task that "accessed" the RMA process. This information is then shared with other tasks.

The format of the TIB database is as follows:

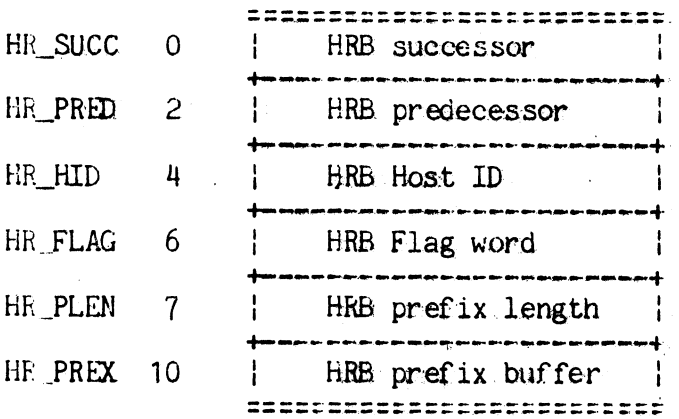
TI_SUCC	0	Successor TIB
TI_PRED	2	Predecessor TIB
TI_CSB	4	TIB's CSB
TI_PORT	6	RMA port number
TI_SFB	10	Shared file buffer ID
TI_SMEM	11	Shared memory address
TI_CTAB	13	Conversion table addr.
TI_CLTH	15	Conversion flags
TI_CFLG	16	Conversion flags
TI_ISY	17	ISYWKP procedure
TI_HRB	21	Current HRB address
TI_HID	23	Current HID
TI_TYPE	24	Host type
TI_PAGES	25	Request buffer pages

TIB Database

Host Record Block

The Host Record Block contains the Host ID, a flag word, and the string containing the Host prefix name. The Host prfix name is in the format of ":net:<hostname><0>". The string can be used to prefix pathnames, process names,or queuenames when returning results to the user.

The format of the HRB database is as follows:



HRB Database

The following are the Host Record Block flag word definitions. The most important bit in the flag is the remote system type flag. It seems that if there are incompatibilities withtin system calls between AOS and AOS/VS then IRMA must reconcile these differences before sending the packet accross the network.

The bits are:

bit 0 - remote system type flag
0 = AOS
1 = AOS/VS

bit 1 - => "incomplete HRB"

bit 15 - intialization flag

Access to URMA

Access to URMA is through the use of either an AU.REQUEST or AU.REPLY IPC message. These messages are of zero length as all the infomation needed is contained in the IPC header. The request may send a HID in the header. The reply may include a shared file buffer identifier.

To access remote resources there are both an ARR.REQUEST and ARR.REPLY. These are both zero length IPC messages. IRMA sends the shared file buffer identifier, maximum pages needed, and host identifier. The reply will contain an error code if some interface or other error occurred (such as a "host type mismatch). URMA reports normal system errors through the ASAP messages (discussed later) in the shared file buffer, and not through the IRMA/URMA interface. IRMA must set the host type bit in the ARR.REQUEST IPC header indicating what it thinks the remote host type is.

There are two types of messages that can be flagged in the IPC message headers. They are: access URMA or access remote resource. These are contained in the IPC header flag as well as the host type bit. IRMA sends the SFB(shared file buffer) identifier, maximum pages needed and HOST ID. The field definitions for the IRMA - URMA IPC message headers are as follows:

Bit	0	0 = AOS 1 = AOS/VS
	1	access URMA
	2	access remote resource
	3	assign SFB
	4	return HOST type
	5	return file capability
	6	some error accurred error code in ?IPTR/?IPTL

The following are the ASAP (RMA) protocol message op-codes: (they must be the same as AOS)

AOP_IDENT	1	IDENTIFICATION
AOP_SYS	2	SYSTEM CALL
AOP_TERM	3	TERMINATE
AOP_CHAIN	4	CHAIN

As sytem calls are executed through the RMA server they may have to be converted from/to 16 bit packet format. This conversion is done for input in the module IABAQ, and for output int the module ISYWKP. The conversion is accomplished by using th system call number and accessing a 16 word segment in a table contained in module VSCDB.

The IRMA interface consists of six modules. They are:

IABAQ - builds a system call request message

IRESO - obtain necessary resources to process a system call request resources are data structures, shared file resources, and URMA server.

IRMA - initialize RMA interface, IRMA patch space, ?IRMA call entry to obatain resources, INULTAB null delimiter table

IUSEND - IPC communication with URMA

ISYWKP - process system call reply messages

VSCDB - IRMA system call conversion table.

CHAPTER 5 -- THE USER PROGRAM
(AOS/VS revision 5.00)

The purpose of this chapter is to describe the user context and its relation to AOS/VS. It will describe virtual memory, ring structure, Inner-ring management terms, and system calls. It will discuss memory, processes, tasks and user devices.

Introduction

AOS/VS is a 32 bit, demand-paged, virtual-memory operating system that runs on MV class machines.

AOS/VS combines the flexibility and convenience of minicomputer architecture with the processing power of a large mainframe computer. AOS/VS is uniquely suited to both commercial and scientific applications. Specifically, AOS/VS provides the following:

- A logical address space of up to 2048 megabytes per process
- Virtual memory management
- Sophisticated process-protection scheme
- Support for concurrent 16 and 32 bit programs
- Compatibility with AOS
- A wide range of system and applications utilities
- High-level language suppoert
- Full functional support for inner rings

Full functional support for inner rings allows you to write multitasked programs that will execute in more than one user ring (user rings are Rings 4 through 7). specifically, full functional support for the inner rings provides the user with the following advantages:

- Improved software performance

  By creating local servers in the inner rings better use of the large logical address space can be made. INFOS_LS.PR is a good example of a local server as it is used to make all the INFOS calls and is loaded into Ring 4 of the user when an open of an INFOS file is needed.

- Larger logical address space

  By using inner rings, the logical address space can be expanded from 512 megabytes (the capacity of one user ring) to 2048 megabytes (the capacity of four user rings).

Virtual Memory

Virtual memory allows the user to run programs that are larger than the physical memory of the configuration. With virtual memory, AOS/VS can move active portions of a program from disk to memory while the program is executing. When the system needs more memory, AOS/VS returns the inactive portions of the program to disk. This process of moving portions of the program in and out of memory is called demand paging.

The portion of an executing program (called a process) that is in physical memory at any given time is its working set. The size of each process's working set changes as demands of the process change. AOS/VS determines the working set size by examining the number of pages the process currently needs as well as its history of page faults.

Page faults are references to memory locations that are not currently in physical memory. When a page fault occurs, the AOS/VS demand paging mechanism moves the page that is needed from disk into physical memory.

AOS/VS allocates a large working set to a process that has a history of many page faults. Therefore, to run a system as efficiently as possible, the number of page faults must be reduced. To do this, the code of a program should be in modules that cluster the instructions and data together as closely as possible. The fewer page faults that processes cause the smaller and more stable is its working set. However, some page faults are unavoidable.

Ring Structure

The entire range of memory locations that a process can address is called its logical address space. The logical address space is divided into eight 512 megabyte units called segments. Although these segments are connected by strict protocols, they are independent of one another. Therefore AOS/VS can use each segment for a different function. This makes virtual memory systems very efficient and reliable.

Each segment is protected by a ring that is permanently bound to that segment. Thus, Ring 0 (the innermost ring) protects Segment 0, ring 1 protects Segment 1, and so forth through Ring 7 (the outermost ring) and Segment 7. These rings prevent segments from interfering with one another, even though each segment may be performing a different function. If a program that is in one segment needs to change or access the contents of another segment, it must follow strict protocols established by the rings. (The system follows these protocols without your knowledge.)

The eight segments (and their rings) are arranged hierarchically. Segment 0 has the greatest ability to change or access the contents of other segments, and Segment 7 has the least. Similarly, Ring 0 gives segment 0 the greatest protection from interference by other segments, and Ring 7 gives Segment 7 the least protection.

Segments 0 through 3 contain the AOS/VS operating system. Segments 4 through 7 contain user programs. Because the user programs and the AOS/VS operating system share the single large logical address space, context switching is unnecessary. In fact, system calls and calls to routines that are in another segment become subroutine calls. This means that when users issue a system call, there is no need for AOS/VS to change contexts. AOS/VS does take part, however, in the execution of most system calls.

Ordinarily. a segment can only change or access contents of segments whose segment and ring numbers are higher than or equal to its own segment and ring number. For example, the rings will not allow a program that is executing in Segment 4 to access the contents of Segments 0 through 3, but they would allow that same process to access Segments 4 through 7. With a subroutine call, however. a segment number is higher than or equal to the target segment can access the segment in which the subroutine actually resides. In this case, the ring that protects the target segment allows the subroutine call to pass through a gate. This gate points to the starting location of the subroutine. Although the user cannot make a cross-ring subroutine call directly to the starting location of the subroutine, the user can return directly from the subroutine. Subroutine returns do not have to pass through gates. The only restriction on subroutine returns is that they must originate from a segment whose number is lower than or equal to the target segment.

Inner-ring Management Terms

- Segment image
  A .PR file that AOS/VS has made part of a process's logical address space.

- Process image
  A union of user segment images and of system segment images.

- Program file
  A segment image linked for any one ring

- Process
  An executing set of segment images, plus all of the system resources that the process image needs to execute.

- Task
  An asynchronous flow of control within a process.

- Global server
  A separate process that performs functions on behalf of a customer process.

- Local server
  A server that shares the same logical address space as its customer.

System Calls

AOS/VS supports a wide variety of system calls. System calls are command macros that call predefined system routines. There are various categories of system calls, which allow the user to do the following:

- Create and manage processes
- Manage the logical address space
- Establish interprocess communications
- Create and maintain disk files and directories
- Perform file input and output
- Create and maintain a multitasking environment
- Define and access user devices
- Establish binary synchronous communications
- Establish customer/server connections between processes
- Perform input and output in blocks, rather than records or lines

Ring Structure

The logical address space of the system is divided into eight 512-megabyte units called segments. Although strict protocols connect these segments, they are independent of one another. As a result, AOS/VS can use each segment for a different function.

Each segment is protected by a hardware ring which is permanently bound to the segment. Ring 0 (the innermost ring) protects Segment 0, Ring 1 protects Segment 1, and so forth through Ring 7 (the outermost ring) and Segment 7. See Figure 5-1.

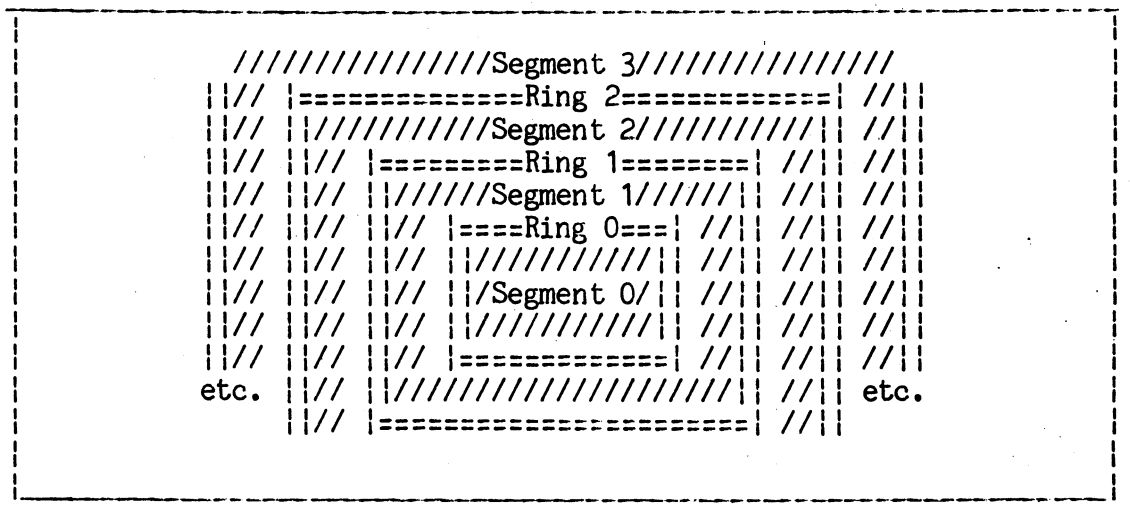


Figure 5-1. Segments and Their Protection Rings

These rings prevent segments from interfering with one another, even though each segment may be performing a different function. If a program which is executing in one segment needs to change or access the contents of another segment, the program must follow strict protocols established by the rings.
(The system follows these protocols without your knowledge.)

The eight segments (and their rings) are arranged hierarchically. Segment 0 has the greatest ability to change or access the contents of other segments, and Segment 7 has the least. Similarly, Ring 0 gives Segment 0 the greatest protection from interference by other segments, and Ring 7 gives Segment 7 the least protection.

Segments 0 through 3 contain the AOS/VS operating system, with the kernel residing in Segment 0. Segments 4 through 7 are for user programs. (In the future, however, rings 4 and 5 may contain software supplied by Data General.)

Subroutine Calls

Because the user programs and the AOS/VS operating system share a single logical address space, context switching is unnecessary. In fact, system calls and calls to routines which are in another segment become subroutine calls. This means that when you issue a system call, AOS/VS does not need to change contexts. AOS/VS does take part, however, in the execution of most system calls.

Ordinarily, a segment can only change or access the contents of segments whose segment and ring numbers are higher than or equal to its own segment and ring number. For example, the rings will not allow a program which is executing in Segment 4 to access the contents of Segments 0 through 3, but they would allow that same process to access the contents of Segments 4 through 7.

With a subroutine call, however, a segment whose segment number is higher than or equal to the target segment can access the segment in which the subroutine actually resides. In this case, the ring which protects the target segment allows the subroutine call to pass through a gate. This gate points to the starting location of the subroutine.

Although you cannot make a cross-ring subroutine call directly to the starting location of the subroutine, you can return directly from the subroutine. Subroutine returns do not have to pass through gates. In fact, the only restriction on subroutine returns is that they must originate from a segment whose number is lower than or equal to the target segment.

User Rings

As you may recall, the user rings are rings 4 through 7, with ring 7 being the default user ring. You may, however, load a program file into one of the other user rings (4 through 6) by issuing the ?RINGLD system call. Also, AOS/VS allows you to write programs which execute in more than one user ring.

This functional support for the inner user rings results in:

- Improved software performance

  You can take better advantage of the large logical address space of the MV-series hardware by using the inner user rings to create local servers. (These are servers which share the same logical address space as their customers and which can be loaded into the inner rings of a process.)

  Local servers are faster than global servers because they do not need to use the interprocess communications (IPC) facility system calls or the ?MBFC and ?MBTC system calls to move data between customer and server. (See Chapters 7 and 8 for details.) Instead, because a local server resides in the same logical address space as its customer, local servers can use MV-series hardware instructions to perform identical synchronization and data movement.

- Improved accounting

  When you use the inner user rings to implement local servers, the server becomes part of the logical address space of the process which uses it. As a result, the server is no longer a separate process. A local server's use of resources is accounted for by AOS/VS as part of the resources used by the customer's process.

- Larger logical address space

  By using the inner user rings, you can expand your logical address space from 512 megabytes (the capacity of one user ring) to 2048 megabytes (the capacity of the four user rings).

Demand Paging

AOS/VS is a demand-paged, virtual-memory operating system. Virtual memory means that memory is a composite of main memory and disk memory. Demand paging is the AOS/VS method of adding logical pages to the working set of a process as the process "demands" (refers to) those pages. The working set is that subset of a process's logical address space which is currently in memory. The working set changes in size and content as the process references pages.

The pages outside the working set make up the process' virtual address space.

Pre-Paging at Fault Time Option

By default, when a page fault occurs (that is, a process demands a page), the system adds one page to the working set. You have the option, however..of requesting that the system add the faulting page, plus a cluster of logically contiguous virtual pages, to the working set at fault time. This option is known as "pre-paging at fault time". Pre-paging is the process of adding unreferenced virtual pages to a working set.

The pre-paging option is useful when

- a program includes large array-like structures (large meaning that the virtual addresses of the structure exceed the main memory available)
- the algorithm which processes the structures tends to reference the entire structure or parts of it sequentially
- the area in which you want pre-paging to occur is in unshared or unused memory

If your program has such characteristics and you understand its page referencing patterns, pre-paging can speed up execution considerably. The system is far more efficient when it moves a cluster of contiguous pages into main memory than when it moves them in one by one.

Before you can use the pre-paging option, your system manager must set the pre-paging parameter during the VSGEN dialogue (if the pre-paging parameter is either 0 or 1, pre-paging is off system-wide; otherwise, this parameter indicates the maximum number of pages you can add to the working set during one fault).

Assuming pre-paging is enabled, you must then use the SPRED utility to edit the preamble of your program file. When you do so, you indicate:

- the starting and ending addresses for the cluster area (remember this must be an unshared or unused portion of memory)
- the cluster size in pages

Program Load Option

Every process starts with a working set large enough to accommodate Page 0 (the first 2K bytes of the logical address space) and the program counter (PC) page. The PC points to the instruction which is currently executing in a program.

You have the option, however, of loading part or all of the unshared address space in your initial program file into physical memory. This program load option is useful when the program which you're executing

- is short
- runs briefly
- frequently references a large unshared area.

By loading in pages initially, you save the time incurred by multiple, sequential page faults.

Before you can use the program load option, your system manager must enable the initial program load option during the VSGEN dialogue by indicating the number of pages a process can have at initial load time. You must then use the SPRED utility to edit the preamble of your program file to indicate the address range of the area you want loaded. For details, refer to "How to Generate and Run AOS/VS" (093-000243).

Variable Swapfiles

Memory contention occurs on a system when currently active processes all desire total working sets larger than the memory available. When contention is light, AOS/VS removes inactive pages from processes and keeps them in a "page file" dedicated to that process. If the process later demands the page(s), the system restores them to the working set.

When heavy memory contention occurs, the system picks a process to swap out to disk via a "swap file". Each process has its own swap file in the SWAP directory. By default, these files have a fixed size.

The fixed size, however, can be a disadvantage for certain processes. For example, if the swapfile is 124 pages and the system decides to swap out a process whose working set size is 250 pages, the system has to break up the working set to fit it in the swapfile. When this same process is later swapped back into memory, the process must incur a series of page faults to restore its working set back to 250. For processes with large working sets, this paging can be costly.

To incur less cost, you can set up a system to allow swapfiles which vary in size from process to process.

To allow the use of variable swapfiles,

- During the VSGEN dialogue, the system manager indicates that he wants variable swapfiles and specifies a default and a maximum swapfile size.

- The system manager gives certain users (those who run programs with large working sets) the privilege of changing their working set size.

- The privileged users edit the preamble of their program files with the SPRED utility. In doing so, they specify a size for the swapfile equal to the typical size of the working set of the program.

Shared and Unshared Memory Pages

Memory pages can be either unshared or shared.

Unshared Pages

Unshared pages are pages in your logical address space which only one process can access. You cannot write-protect unshared pages.

Shared pages

Shared pages are pages of physical memory which may be shared among multiple users. Unlike shared pages, where each user of a page has his own private copy, several users can access (and possibly modify) one physical shared page. AOS/VS keeps track of the use of a shared page. When this use count is 0, the page may stay in memory if the demand for memory is low; if the demand for memory exceeds what is available, the page is released.

If a shared page is not currently in use, AOS/VS places it on an LRU chain. An LRU chain is a list of released shared pages, which is arranged in least recently used (LRU) order. The shared pages on the LRU chain are candidates for re-use by any process.
When you issue the ?RPAGE system call, AOS/VS does not immediately release the shared page from memory. If you modified the page and, therefore, want to release and update it immediately, you must issue either a ?FLUSH system call, a modified version of the ?RPAGE system call, or the ?ESFF system call. All three provide ways of writing the contents of a shared page to disk.
Before you can use the ?SPAGE, ?RPAGE, or ?FLUSH system calls, you must use the?SOPEN system call to open the target file for shared access. A file opened this way is called a shared file. The ?SOPEN system call gives you the option of opening your shared file for Read-only access. To close a shared file, you must issue the ?SCLOSE system call.

There are three ways to use shared memory pages:

- Explicitly, by using the shared-page system calls, such as ?SSHPT, ?SOPEN, ?SPAGE, and so forth

- Implicitly, by defining a shared area with assembly language pseudo-ops

- By opening a file for shared access with a special form of the ?OPEN system call

The .NREL and .PART pseudo-ops allow you to define shared areas in an assembly language program. The .NREL pseudo-op directs the macroassembler (MASM) to place the code or data that comes after it into one of the predefined NREL (normal relocatable) memory partitions. To specify which partition you want, use the appropriate nonzero argument with the pseudo-op.

For example, the statement .NREL 5 tells MASM to place all subsequent source statements in the predefined shared-data partition. The statement .NREL 1 and .NREL 7 tell MASM to place all subsequent source statements in the predefined shared-code partition.

To define your own partitions in NREL memory, use .PART pseudo-ops. This pseudo-op allows you to define a variety of attributes (characteristics) for the partition, including whether it is part of the shared or unshared memory.

When you link your source code, the Link utility uses your .NREL and .PART specifications to create shared (and unshared) partitions in the final program file. The shared areas become part of the logical address space of any proces that uses the program file.

Protected Shared Files

A set of common logical servers can use shared memory files to coordinate access to a common resource. Each local server that wants to share the memory must first open and then read from or write to the same shared file.

Inner-ring servers may need to limit access to their shared files. They may not want any segments other than themselves to have access to their shared memory. However, the access control list (ACL) protection mechanism cannot protect a local server, because all segments within a process share the same username. The ?SOPPF and the ?PMTPF system calls permit a more private form or protecting shared files.

You can use the ?SOPPF system call to open a shared file in a protected manner. Once a shared file has been opened in a protected manner, the opener can issue the usual shared-page system calls, just as if the channel were opened by a ?SOPEN system call. To close a shared file, whether or not it was opened in a protected manner, you can use the ?SCLOSE system call.

The first ?SOPPF system call behaves differently than subsequent ?SOPPF system call opens of the same shared file that you want to open in a shared manner.

After a segment image uses the ?SOPPF system call to open a protected shared file for the first time, that segment image is called the "first opener" of the file. The first opener of a protected shared file can use the ?PMTPF system call to permit other segment images to access the file. The ?PMTPF caller also informs AOS/VS of the type of file access privileges that the caller wants to pass to another segment image.

Only the first opener of a protected shared file can issue a ?PMTPF system call against that file. Also, there must be a valid connection between PID/ring tandem from which the ?PMTPF system call is issued (the server) and the PID/ring tandem of the target (the customer).

A first opener that issues the ?PMTPF system call cannot pass access privileges it does not have itself. In addition, access privileges are not cumulative.

An access grant remains active until one of the following events occurs:

- The connection between the first opener of the protected shared file and the target segment image is broken.
- The first opener closes the file.
- The first opener revokes the access grant by issuing anther ?PMTPF with fewer or no access privileges.

This means that a segment image possesses only access privileges specified by the most recent ?PMTPF system call that addressed that segment image. Thus, a ?PMTPF system call that specifies no privileges can revoke a segment image's access privileges.

Dedicated and Undedicated Memory Pages

Just as AOS/VS distinguishes between shared and unshared pages, it also distinguishes between dedicated and undedicated memory pages.

- Dedicated pages are memory pages that AOS/VS reserves for specific purposes. They include physical pages occupied by the resident portion of AOS/VS and pages wired to a resident process by the ?WIRE system call.

- Undedicated pages are pages that AOS/VS can assign to any process as the process needs them. Undedicated pages are not necessarily "unused" pages; they are simply available for reassignment. The ?GMEM system call returns the current number of undedicated pages available to the calling process.

User Context

The user's unshared area starts at the first word of the logical address space in the current ring, and extends toward numerically higher addresses. The shared page area occupies the numerically highest portion of the address space and expands upward and downward.

Between the shared and unshared portions of the logical context, there can be an "unused" area. You can allocate this area with the system calls ?MEMI and ?SSHPT. The ?MEMI system call modifies the unshared area's upper boundary, while the ?SSHPT system call modifies the number of shared pages in the logical address space and the position of the shared area in your user address space.

Figure 5-2 shows the relationship among the unshared, unused, and shared areas in a typical user context.

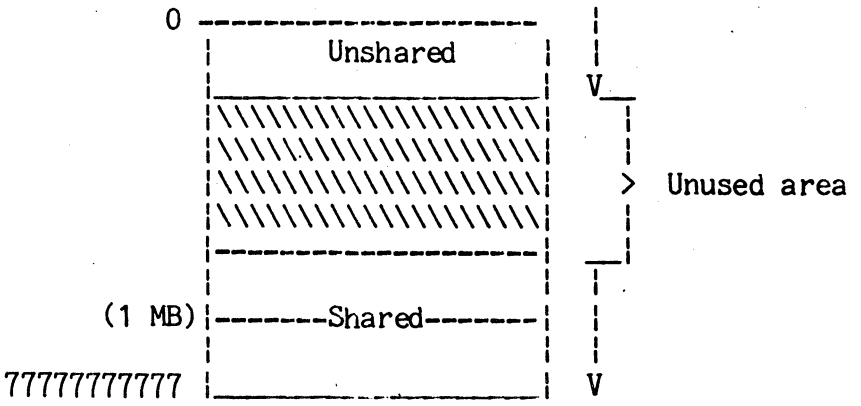


Figure 5-2.

Processes

This section defines processes and how AOS/VS uses them. You must be familiar with the following terms and what they mean to AOS/VS:

- Program File

  A file containing executable code. To the system, this file is a static image, known as a segment image, for a particular segment of memory.

- Segment Image

  A program file that AOS/VS has made part of a process's logical address space. Segment images are either created by users or supplied by the operating system (as resources). Together, user segment images and system segment images make up a process image.

- Process

  An executing set of user segment images and system segment images.

- Task

  A task is a path through a process. Tasks are asynchronously controllable entities to which the system allocates CPU resources for a specific time. A task is the basic element of a process and may only execute code within the address space allocated to its process.

Each process is made up of one or more tasks, which execute asynchronously. You can design your code so that several tasks execute a single re-entrant sequence of instructions, or you can create a different instruction path for each task. Control always goes to the highest priority ready process, and within that process, to the highest priority ready task.

When you create a process, it exists until one of the following events occurs:

- The process traps.
- The process terminates voluntarily.
- Another process terminates the process.
- The process's father terminates.

Memory Scheme

AOS/VS allocates memory and CPU time to each process based on its priority and scheduling characteristic. (A process is scheduled either on a round-robin or a heuristic basis, as described later in this section.

The entire range of locations addressed by aprocess is its logical address space. Under AOS/VS, a process's user-visible address space can consist of up to 512 megabytes for each of the four user rings (rings 4 through 7).

At any given time, only a subset of each process's logical address space is in memory. This subset, which is called the working set, changes in size and content as the process references pages and then returns them to disk.

Every process starts with a working set large enough to accommodate Page 0 (the first 2K bytes of the logical address space) and the program counter (PC) page. The program counter points to the current control point in a program.

Users have the option, however. of starting off with a much larger working set by initially loading some or all of the unshared address space in their program files.

The rest of the logical address space -- the pages outside the working set -- is virtual address space.

The size of a process's working set directly relates to the number of memory pages the process currently needs or is likely to need. When a process refers to a page or set of pages outside its working set, the hardware signals a page-fault condition. AOS/VS responds by adjusting the size of the working set. The ?WIRE and ?UNWIRE system calls give a process with sufficient privleges control over its working set. The ?WIRE system call wires (that is, permanently binds) pages to the working set. The ?UNWIRE system call releases previously wired pages.

You can also control the size of the working set through the ?AWIRE, ?WIRE, and ?UNWIRE system calls, all of which determine how many pages are wired (bound) to the working set. In addition, the ?PROC system call lets you set the minimum and maximum working set size when you create a process. Furthermore, a change in the type of the process (with ?CTYPE or ?PROC) affects the size of the working set, as resident processes (those always residing in memory) automatically have more pages wired to their working sets.

When a page fault occurs, the operating system normally adds one page to the working set. You have the option, however, of requesting that the system add a cluster of pages to the working set when a page fault occurs.

Process Types

To manage the multiprocess environment, AOS/VS allocates main memory to processes based on their priorities and scheduling characteristics. Processes fall into two main categories:

- Those which always reside in memory (these are called resident). In general, only the most critical processes in your system environment should be resident.
- Those which the memory manager moves back and forth between disk and memory (these are called preemptible and swappable).

NOTE: Under AOS/VS preemptible and swappable processes are almost identical; for differences, see the section on "Priority Changes" in this chapter. Under AOS, however, a preemptible process ALWAYS has a higher priority than a swappable process.

When you create a process with the ?PROC system call, by default the process is the same type as its father. You may, however, give it another type, if you wish.

Any process can issue the ?WIRE system call to bind pages to its working set. Remember, however, that if you start wiring a lot of pages to a resident process, you'll degrade the performance of the system because of the increased number of pages the system will be unable to swap out when contention occurs.

In addition to any pages you may wire with ?WIRE, AOS/VS automatically wires the Agent of a resident process to its working set (the Agent is that part of AOS/VS which pre-processes system calls and serves as an interface to the operating system.) You may, however, issue an ?AWIRE system call to unwire all the Agent pages from a resident process, except for those needed to support user devices. As a result, you free up some pages of memory and improve the efficiency of the system as a whole. Your resident process, however, may seem less efficient.

As a general rule, AOS/VS keeps interactive swappable processes in memory longer than non-interactive swappable processes. You may change this, however, by setting the bias factors.

AOS/VS treats a pre-emptible process as a high-priority swappable process. However, when a resident process or a highe priority pre-emptible process requires memory, AOS/VS swaps the pre-emptible process out to disk. Also, when another process explicitly blocks a pre-emptible process (with the ?BLKPR system call), AOS/VS can swap the pre-emptible process out to disk if it needs more memory.

Priority Numbers

Eligible processes compete with each other for CPU time, based on their individual priority numbers. AOS/VS uses priority numbers to determine each process's priority. When you create a process, you may assign it a priority number.

Priority numbers range from 1 (the highest priority) through 511 (the lowest). These numbers span three scheduling groups (with no overlap and no gaps), whose boundaries are determined during VSGEN. For details, see the section on "Process Scheduling" in this chapter.

Priority Changes

If a process wants to change its own priority, it may issue the ?PRIPR system call. To change the priority of another process, however, the calling process must be in Superprocess mode.

Changing Type

The priority of a process may also change when you change its type with either ?CTYPE or ?PROC. Given that the boundaries of the 3 scheduling groups are

Group 1 = 1 - G1
Group 2 = G1+1 - G2
Group 3 = G2+1 - 511

then Tables 5-1 and 5-2 summarize the changes in priority which occur when a process changes type. Notice that a swappable process can never assume a priority of 1, 2, or 3, but it may APPEAR to do so because of the way priority numbers get mapped (see the discussion of Mapping" below.)

Table 5-1 Priority Changes Going from a Resident or Preemptible to Swappable Type

Priority Before Change	Priority After Change
1 - 3	1 - 3 * **
4 - G1	2 * **
G1+1 - G1+3	1 - 3 **
G1+4 - G2	G1+4 - G2
G2+1 - 511	G2+1 - 511

* This parallels what happens under AOS.

** Although you would see these numbers if you displayed the priority of a process with the CLI PRIORITY command, the actual priorities would be G1+1 - G1+3. See "Mapping" below.

Table 5-2 Priority Changes Going from a Swappable to a Resident or Preemptible Type

Priority Before Change	Priority After Change
1 - 3 **	1 - 3 *
4 - G1	4 - G1
G1+4 - 511	G1+4 - 511

* This parallels what happens under AOS.

** Although you would see these numbers if you displayed the priority of a process with the CLI PRIORITY command, the actual priorities would be G1+1 - G1+3. See "Mapping" below.

Mapping

A resident or preemptible process can assume any of the priority numbers 1 through 511. The system uses this number in gauging the importance of the process during scheduling and displays this same number if you request the process's priority.

To maintain compatibility with AOS, however, AOS/VS has to map priority numbers for swappable processes. As a result, the actual number the system uses in its scheduling calculations and the number it displays when you request the process's priority may differ.

The discrepancy between actual and displayed priority numbers occurs in three cases:

1) If you assign a swappable process a priority of 1, 2, or 3.

2) If you assign a swappable process a priority of G1+1 - G1+3.

3) If a resident/preemptible process with a priority of 1, 2, or 3 changes its type to swappable.

In all three cases, AOS/VS uses a priority number of G1+1 - G1+3 when scheduling the process because a swappable process cannot have a priority of 1, 2, or 3. The system cannot, however, display the numbers G1+1 - G1+3 for a swappable process, and so displays 1 - 3.

In all other cases (4 - G1 and G1+4 - 511), the actual number is the same as the displayed number.

Remember, however, that if you do assign a swappable process a priority of 1 and then it changes type to resident (or preemptible), the resident process WILL have an actual priority of 1, even though the swappable process could not.

Examples of Mapping

1) If a resident process with a priority of 2 changes its type to swappable, the system displays a priority of 2, but it actually uses G1+2 when scheduling the swappable process.

2) If a resident process with a priority of 3 changes its type to preemptible, the system displays and uses a priority of 3 for the preemptible process.

3) If a preemptible process with a priority of G1+3 changes its type to swappable, the system displays a priority of 3, but uses G1+3 in scheduling the swappable process.

4) If a preemptible process with a priority of G2+44 changes its type to swappable, the system displays and uses a priority of G2+44 for the swappable process.

5) If a swappable process with a displayable priority of 3 (meaning its real priority is G1+3) changes its type to resident, the system displays and uses a priority of 3 for the resident process.

6) If a swappable process with a priority of 5 changes its type to preemptible, the system displays and uses a priority of 5 for the preemptible process.

Process Scheduling

AOS/VS schedules eligible processes based on their priority numbers and scheduling characteristic. As you may recall, the range of process priority numbers (1 through 511) spans three scheduling groups.

Group 1 ranges from 1 to a number, "G1", which is set during VSGEN. AOS/VS schedules any process whose priority number places it in Group 1 on a round-robin basis. Under this scheme, each process is allocated a uniform slice of time during which it may execute. Once a process of a specified priority temporarily stops executing (having used up its time slice), it is not chosen to execute again until all other processes of that priority have been chosen to execute.

Group 2 ranges from G1+1 to a number, "G2", which is also set during VSGEN. AOS/VS schedules any Group 2 process heuristically, which means that the system takes the process's past behavior into account when alloting it an interval of time during which it may execute.

Group 3 ranges from G2+1 to 511. AOS/VS handles processes in this group on a round-robin basis.

For details of setting G1 and G2, see "How to Generate and Run AOS/VS" (093-000243.)

NOTE: If you need to maintain compatibility with AOS, G1 and G2 must be set to 255 and 258, respectively.

Group 1 processes are always more important (that is, more likely to be chosen for execution) than those in Group 2 or 3, and Group 2 processes are always more important than those in Group 3. Within each group, the lower the priority number, the greater the importance of the process. The importance of a process may, however, alter as a result of a change in type, as Tables 5-1 and 5-2 show.

If an executing process cannot proceed, you can issue the ?RESCHED system call, which allows the calling process to give up control of the CPU and forces AOS/VS to immediately schedule another process for execution.

Process Identification

A process identifier (PID) and a process name identify each process. When you create a process, AOS/VS assigns it a unique process identifier in the range from 1 to 255. At the same time, you must assign a process name to that process.

A full process name is a character string that consists of a username and a simple process name, with a colon(:) between two elements. Each element can contain up to 15 valid filename characters. The valid filename characters are:

- Letters A through Z.
- Numbers 0 through 9.
- Period (.), dollar sign ($), question mark (?), and underscore (_).

A username functions like a family surname. AOS/VS uses this part of the process name to determine the process's genealogy and its access rights to files. By default, each son process bears its father's username. A father process can assign its sons a different username only if the father was created with the privilege to do so.

You can use either the full process name or a simple process name as input to the system calls. When you supply a simple process name, AOS/VS expands it.

You cannot assign the same simple process name to processes that have the same username. If you do, AOS/VS returns error code ERPNU (process name already in use).

Process States

When a process has gained memory, it competes for CPU time. At this point, AOS/VS looks at both the priority and state of a process to determine its order of execution. A process is always in one of the following three states:

- Eligible

  A process is eligible for CPU time when it has acquired memory and is ready to run.

- Ineligible

  A process is ineligible when it has not acquired memory, even if it is otherwise ready to run. Every process is ineligible at its inception.

- Blocked

  A process is blocked if its execution is suspended to wait for a specific event that may or may not occur. A process can block voluntarily, another process can block it (generally via the ?BLKPR system call), or AOS/VS can block it.

Process Blocking

AOS/VS blocks a process under the following conditions:

- When another process explicitly blocks it, using the ?BLKPR system call.

- When the process creates a subordinate process, called a son, and voluntarily blocks itself until the son terminates.

- When the process issues a system call that suspends its only active task.

The last condition implies that the process has only one task or that all of its other tasks are suspended. ?IREC and ?WDELAY are two examples of system calls that can cause a process to block.

AOS/VS unblocks a process under the following conditions:

- When the process previously blocked with ?BLKPR is explicitly unblocked with ?UBLPR. (?BLKPR and ?UBLPR work as a pair; ?UBLPR unblocks only those processes that were previously blocked with ?BLKPR.)

- When a son created by the process terminates (provided the father voluntarily blocked to wait for the son to terminate)

- When a task within the process becomes ready to run (AOS/VS blocked the process because it had no ready task)

When memory contention occurs, AOS/VS is more likely to swap blocked processes or to remove pages from them. The processes that have been blocked the longest are the prime candidates for these actions.

Keep in mind that resident processes cannot be explicitly blocked.

Process Traps

A process trap is a hardware error. Each process exists until it terminates voluntarily, becomes terminated by another process, or encounters a process trap (that is, "traps"). Any one of the following conditions can cause a process to trap:

- The process tries to reference an address that is outside its logical address space or refers to an invalid address within Ring 7.

- The process tries to use more than 16 levels of indirection in a memory reference instruction.

- The process tries to read, write, or execute code that is protected against any of these actions (for example, it attempts to write to write-protected shared area of its logical address space). The ?VALIDATE system call decreases the likelyhood of this kind of trap by letting you check an area for access before attempting a read or write.

- The process uses I/O instructions while LEF is disabled and I/O protection is enabled.

- A process tries to execute a privileged instruction in a user ring.

When a process traps or terminates voluntarily, AOS/VS uses the IPC facility to send that process's father a termination message. If the process terminated on a trap, the IPC message describes the cause.

Break Files and Memory Dumps

When a process terminates, you can save the state of certain memory parameters and tables (for example, the process's UST and TCB's) in two ways:

- You can create a break file

  A break file is a status file in the terminated process's working directory that contains this information. You must be logged on to examine a break file.

- You can dump the contents of a particular ring to a dump file.

  A dump file contains all of the information that a break file contains, plus a copy of the memory image. Also, you do not have to be logged on to examine a dump file.

  To perform a dump, issue the ?MDUMP system call, which creates a dump file wherever you specify.

There are two ways to terminate a process and explicitly create a break file:

- Issue a ?BRKFL system call.

- Type a CTRL-C CTRL-E sequence from the process console.

To create a breakfile every time a process traps, set bit ?PBRK in offset ?PFLG of the process's ?PROC packet.

AOS/VS copies the following words to the break file:

Status Word	Contents
?BRAC0	Value of AC0
?BRAC1	Value of AC1
?BRAC2	Value of AC2
?BRAC3	Value of AC3
?BRPC	Value of PC
?BRTID	Task ID
?BRFP	Value of the stack frame pointer
?BRSP	Value of the stack pointer
?BRSL	Value of the stack limit
?BRSB	Value of the stack base

Unless you specify another pathname, AOS/VS assigns the break file the default pathname is:

?pid.time.BRK

where:

- pid is the 3-digit PID of the terminated process

- time is the time of the termination, in the form hours_minutes_seconds

AOS/VS only creates a break file if the terminated process has write or APPEND access to its working directory and if the working directory has enough disk space for the break file.

The ?ENBRK system call, unlike the ?BRKFL system call, which terminates a process and creates a break file, does not terminate the process. Instead, if the process traps, issues a CTRL-C CTRL-E, or is the target of a TERM/BREAK, the ?ENBRK system call allows AOS/VS to create a break file of whatever user ring you specified as its target ring. The ?ENBRK system call allows AOS/VS to create a break file, it does not explicitly direct it to do so.

Linking Programs Together with the ?CHAIN System Call

The ?CHAIN system call allows you to link together several steps of a long, complex program set, where each program is a separate program file. The programs may be of different types (i.e., 16-bit and 32-bit). This is useful if you're approaching maximum PID counts on your system or if you lack the privilege to create unlimited sons. The ?CHAIN system call actually releases the system resources that one process is using, and then executes a new program. In addition, the ?CHAIN system call transfers the following attributes to the new program:

- The username, process name, PID, console, search list, default ACL, and working directory of the calling process
- The generic file associations of the calling process (for example, the filenames associated with generic files @INPUT, @OUTPUT, @LIST, and @DATA).
- The privileges, process type, and priority of the calling process.

When a process chains to a new program, AOS/VS performs the following steps:

- Unloads all of the process's inner user rings.
- Terminates all son processes that were previously created by ?PROC system calls issued from the inner user rings.
- Breaks the connection, which in turn, causes AOS/VS to revoke access privileges to protected shared files.

Inner Rings

To load program files into a specific ring, you can issue the ?RINGLD system call. Then, to find out what program was loaded into the ring, you can issue the ?RNGPR system call. If you want to prevent the ?RINGLD system call from loading a runtime routine into a particular ring, you can issue the ?RINGST system call.

To cross from an outer ring to an inner, a program must have access to the proper gates; that is, entry points to the code in the inner ring. When you write a program to execute in Rings 4,5, or 6, you must define an array of the legal entry points (gates).

In the module in which you define your gate array, you must declare the gate entry points as .EXTG (external gate). Also, in your source module, you must declare gate entry points as .ENT (entry point). The 'Principles of Operation of ECLIPSE 32-Bit Systems' manual explains how to reference gates and how to set up gate arrays.

Figure 5-3 shows how a process can span rings. For the purpose of the figure, assume that the main program has used the ?RINGLD system call to load a program file into Ring 6.

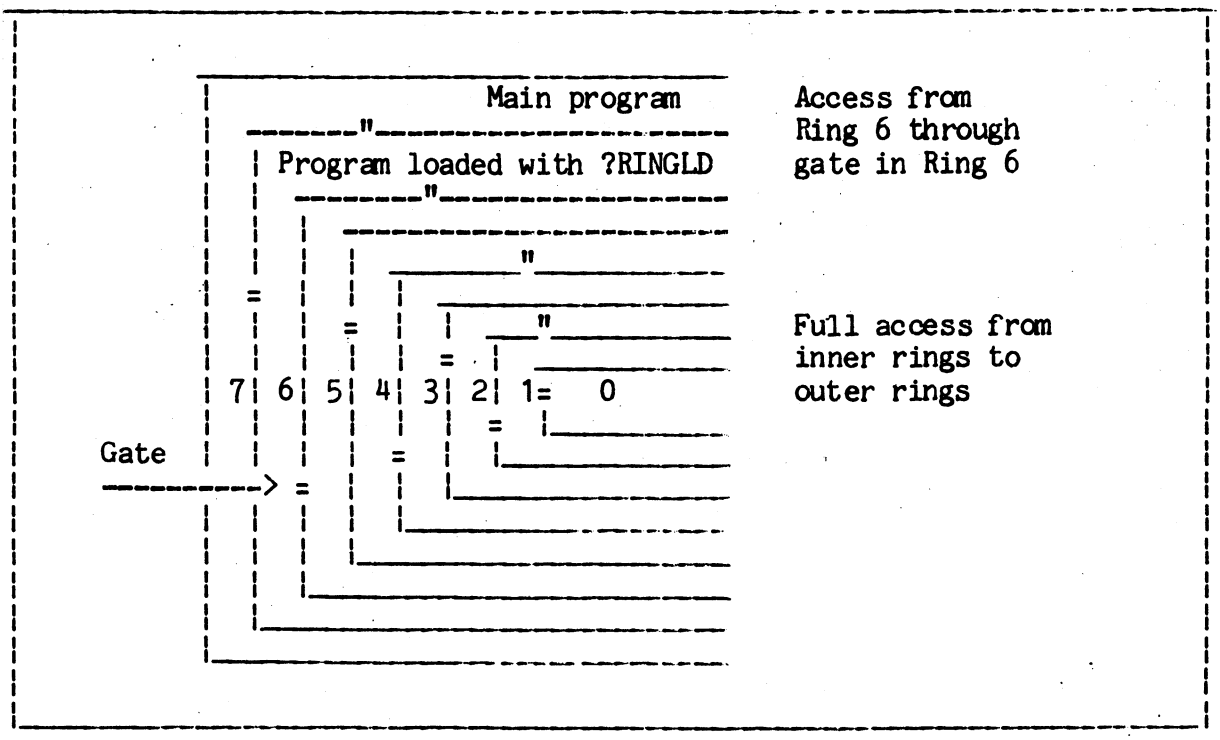


Figure 5-3. Ring structure

File Creation and Management

A file is a collection of related data that is treated as a unit. "File" also refers to the disk blocks used to store files. Each file has a filename by which you and AOS/VS address that file. You can create files and assign them filenames by using the ?CREATE system call, the CLI, or one of the text editors AOS/VS supports. Or, you can create files as you assemble, compile, and link your source code. In the latter case, the utilities assign the filename.
There are two general types of devices that allow you to store and retrieve file information. YOu can use multifile devices, such as disks and magnetic tape. to perform file I/O and to store and retrieve files. Other devices, such as consoles, you can use strictly for file I/O.

Disk File Structure

Each file consists of one or more file elements. A file element is a set of contiguous 512-byte disk blocks. (contiguous disk blocks are blocks with sequential addresses). The default file-element size is four (four disk blocks per element), or whatever file-element size you selected during the system generation procedure. You can also specify a file-element size when you create a file.
AOS/VS always rounds a file-element size to the next higher multiple of the default file-element size. For example, if you create a file with a file-element size of five and the default file-element size is four, AOS/VS rounds the file-element size to eight.

AOS/VS allocates disk space to a file based on its file-element size. For example, a file with a file-element size of four "grows" in units of four contiguous blocks.

The blocks that make up a file element are always contiguous, although the file elements may not be. For example, a file with a file-element size of four may consist of a number of "scattered" 4-block elements.

To keep track of each file's file elements, AOS/VS maintains one or more index levels for each disk file. An index is a single block that lists the address of each file element. As a file exhausts one index, AOS/VS provides a superior index, to a maximum of three index levels. A pointer in each index level links that level with its immediate subordinate.

Files with larger file-element sizes have fewer separate elements and, therefore, require fewer index levels. Files with smaller file-element sizes are easier to store, however, because each block in a file element must be contiguous. (It is easier for AOS/VS to find eight contiguous blocks, for example, than to find 500).

The maximum size for a disk file is 2**23 blocks. You cannot use all the blocks in the total disk storage, however, because AOS/VS must reserve some for index blocks, to store disk bootstraps, and for other purposes.

Directory Creation

Generally, you group related disk files into directories for convenience. A directory is a file that contains information about a particular set of files. For example, you might create a directory called PL_1 to group all PL/1 source files, or a directory called UPD to contain all user profiles. The AOS/VS file name conventions also apply to directory names.

AOS/VS organizes directories into a hierarchial tree structure similiar to the process tree structure. (See figure 5-4.) The initial hierarchy. A colon represents the root.

Directory Entries

Each directory contains a directory entry for every one of its subordinate files. A typical directory entry contains the name of the file, its file type, a list of the access privileges for various users, and other information unique to the file type. For example, a directory entry for an IPC file contains such additional information as the PID of the proces that created the file and the file's local port number. AOS/VS recognizes 256 different types of directory entries, numbered from 0 to 255.

Data General reserves types 0 through 127; the user parameter files PARU.32 and PARU.16 define these types. Users can define directory entry types 128 through 255.

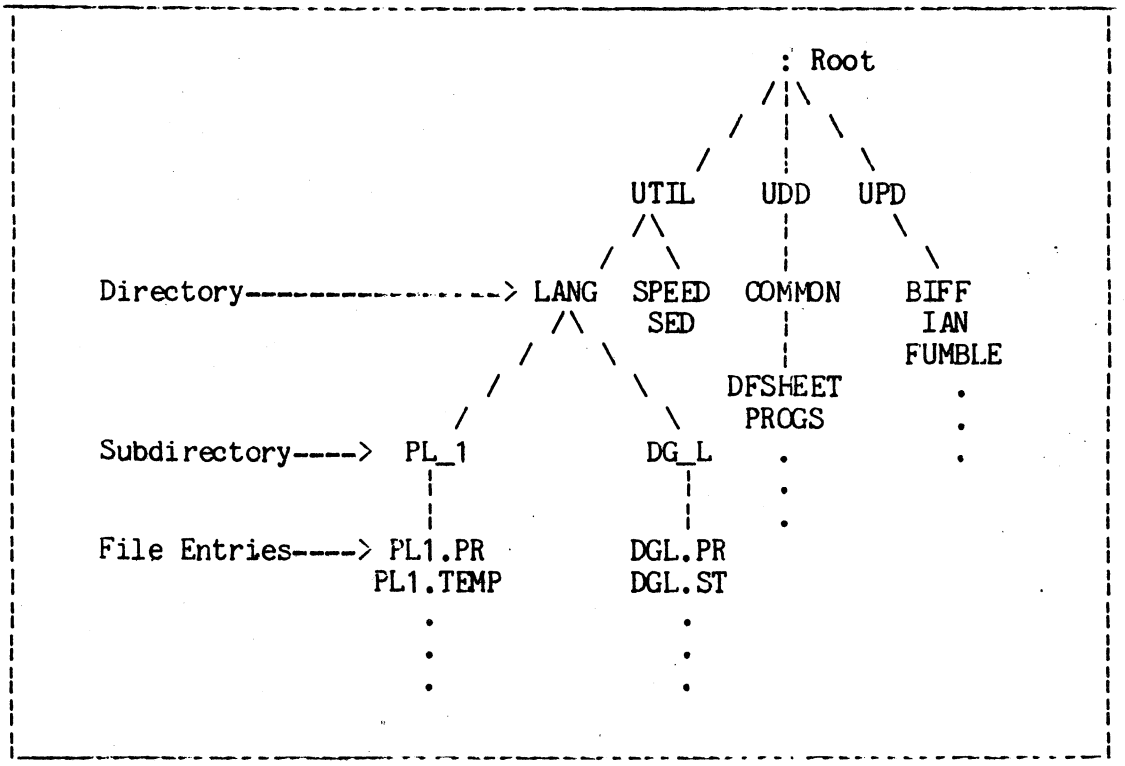


Figure 5-4. Sample Directory Tree

File Types

A file's characteristics and function determine its file type. Table 5-3 lists the AOS/VS file types.

User data files (file type ?FUDF) are not executable files. Typically, you use ?FUDF files to store the object files or text files you create with one of the text editors.

As Table 5-3 indicates, there are three types of program files:

- ?FPRV files, which are developed under AOS/VS.
- ?FPRG files, which are developed under AOS.
- ?FUNC files, which are developed under VS/UNIX.

Table 5-3. File Types

Mnemonic	Type	Comments
?FUDF	User Data File	Usually applies to source or object files.
?FTXT	Text File	Should contain ASCII text.
?FPRG	AOS Program File	Program file for use under AOS (16-bit code).
?FPRV	AOS/VS Program File	Program file for use under AOS/VS (16-bit code or 32-bit code).
?FUNX	VS/UNIX file	File for use under VS/UNIX
?FDIR	Disk Directory	None.
?FCPD	Control Point Directory	(See "Disk Space Control" in this chapter.)
?FLNK	Link File	None.
?FSTF	Symbol Table File	Produced by the Link utility and used primarily by AOS/VS.
?FUPF	User Profile File	Used by PREDITOR (user profile editor) and EXEC.
?FSDF	System Data File	None.
?FIPC	IPC Port Entry	(See section on IPCs).
?FMTF	Magnetic Tape File	None.
?FGFN	Generic Filename	Refers to the generic filenames; that is, @OUTPUT, @LIST, @DATA, etc.
?FGLT	Generic Labeled Tape	None.
?FDKU	Disk Unit	None.
?FSPR	Spoolable Peripheral Directory	None.
?FQUE	Queue Entry	None.
?FLDU	Logical Disk	Cannot create with the ?CREATE system call. (See "Logical Disks" in this chapter.)

Table 5-3 continued File Types

Mnemonic	Type	Comments
?FMCU	Multiprocessor Communications Unit	Cannot create with ?CREATE system call.
?FMTU	Magnetic Tape Unit	Device you use to access magnetic tape files; cannot create with the ?CREATE system call.
?FLPU	Data Channel Line Printer	Cannot create with the ?CREATE system call.
?FNCC ?FPCC ?FFCC ?FOCC	FORTRAN Carriage Control	None.
?FCRA	Card Reader	Cannot create with ?CREATE system call.
?FPLA	Plotter	Cannot create with ?CREATE system call.
?FCON	Console (hard-copy or video display)	Cannot create with ?CREATE system call.
?FSYN	Synchronous	Cannot create with ?CREATE system call.

You cannot execute an AOS-written program under AOS/VS unless you relink it with the AOS/VS Link utility. (In some cases, you must re-assemble or re-compile an AOS program file to execute it under AOS/VS.) If you try to execute an ?FPRG program file under AOS/VS, it returns error code ERIFT (illegal file type).

Directory Access

Each process that runs under AOS/VS has a working directory. A working directory is a process's reference point in the overall directory structure and its starting point for file access. (In other words, your working directory is the directory you are working in.) You can use any directory as a working directory, provided you have proper acces to it.

In most cases, you will probably access files from your current working directory. When you refer to a file that is not in your working directory, you must refer to it by a pathname, unless you've included the file's parent directory in the search list for your process

If you want to change your working directory so that you can access files that are not currently in it, issue the ?DIR system call. Also, the ?DIR system call allows you to return to your initial working directory after you are finished working elsewhere.

A search list is a list of directories that AOS/VS searches if it fails to find the file that you want in your working directory. You can use the ?SLIST system call to create a search list or to change the contents of an existing search list. To examine your current search list, issue the ?GLIST system call.

Filenames

A filename is a byte string that consists of at least one, and as many as 31, ASCII characters. The legal filename characters are:

- Uppercase and lowercase letters
- Numerals 0 through 9
- Period (.)
- Dollar sign ($)
- Question mark (?)
- Underscore (_)

AOS/VS treats uppercase and lowercase letters alike.

To rename a file, issue the ?RENAME system call.

In general, you can use any conventions you like to name files and families of files. Table 5-4 lists the filename conventions used by AOS/VS and its utilities.

Table 5-4. Filename Conventions

File	Filename End In
Assembly language source file	.SR
CLI macro files	.CLI
Object files	.OB
Program files	.PR
Temporary files	.TMP and begin with ?
Library files	.LB

You create source files for a program's source code, and then assemble of compile them to produce object files. One or more linked object modules and/or library files make up an executeable program file. In general, you use temporary files for data that requires only short-term disk storage.

Pathnames

A pathname specifies the exact location of a directory or file in the file structure. For example, you could use the following pathname to locate directory EAGLE, an entry in the superior directory PAT:

:UDD:PAT:EAGLE

Directory PAT is inferior to directory UDD, which , in turn, is inferior to the system root, which the colon (:) represents.

A pathname can consist of:

- A prefix alone (such as a colon to indicate the system root).
- An optional prefix followed by the name of a directory or file.
- Pairs of prefixes and directory names or filenames.

The prefix directs AOS/VS to a particular point in the file structure. Table 5-5 lists the valid pathname prefixes.

Table 5-5. Valid Pathname Prefixes

Prefix	Meaning
:	Start at the system root directory.
=	Start at the current working directory.
^	(Uparrow) Move up to the immediately superior directory. (You can use more than one uparrow in a pathname.)
@	Start at the peripheral directory (:PER).

The peripheral directory (:PER), which is inferior to the root, contains the names of generic filenames, which refer to classes of I/O devices, and the names of system devices.

The = prefix directs AOS/VS to search only the working directory. Generally, when a pathname has no = prefix and the file that you want is not in the working directory, AOS/VS checks the search list. The = prefix prevents AOS/VS from doing this.

To construct a pathname to a directory other than your working directory, use either a single prefix, or one or more pairs of prefixes and directory names. For example, the prefixes ^^ cause AOS/VS to move to the directory two levels above your current working directory. The pathname :UDD:PAT explicitly directs AOS/VS to directory PAT, which is subordinate to botheUDD and the root.

A full pathname traces the path of a particular file all the way from the root to the file's parent directory. The last entry in a full pathname is :filename, where filename is the name of the file you want to access. The following is a complete pathname to the file GLOSSARY, which is an entry in directory EAGLE:

:UDD:PAT:EAGLE:GLOSSARY

Figure 5-4 illustrates the use of pathname strings for a sample directory structure.

Many system calls require pathnames as arguments. When you supply a pathname as an argument, you must terminate it with a null <000> byte. similarly. the system uses this format when passing pathnames to your programs. Remember to allow sufficient buffer space to hold the filename and the null terminator whenever you use a system call that returns a pathname or filename.

The ?GNAME and ?CGNAM system calls both return a file's complete pathname, starting with the root. However, they are not the same in that the ?GNAME system call requires a filename or portion of a pathname as input, while the ?CGNAM system call, requires the file's channel number as input.

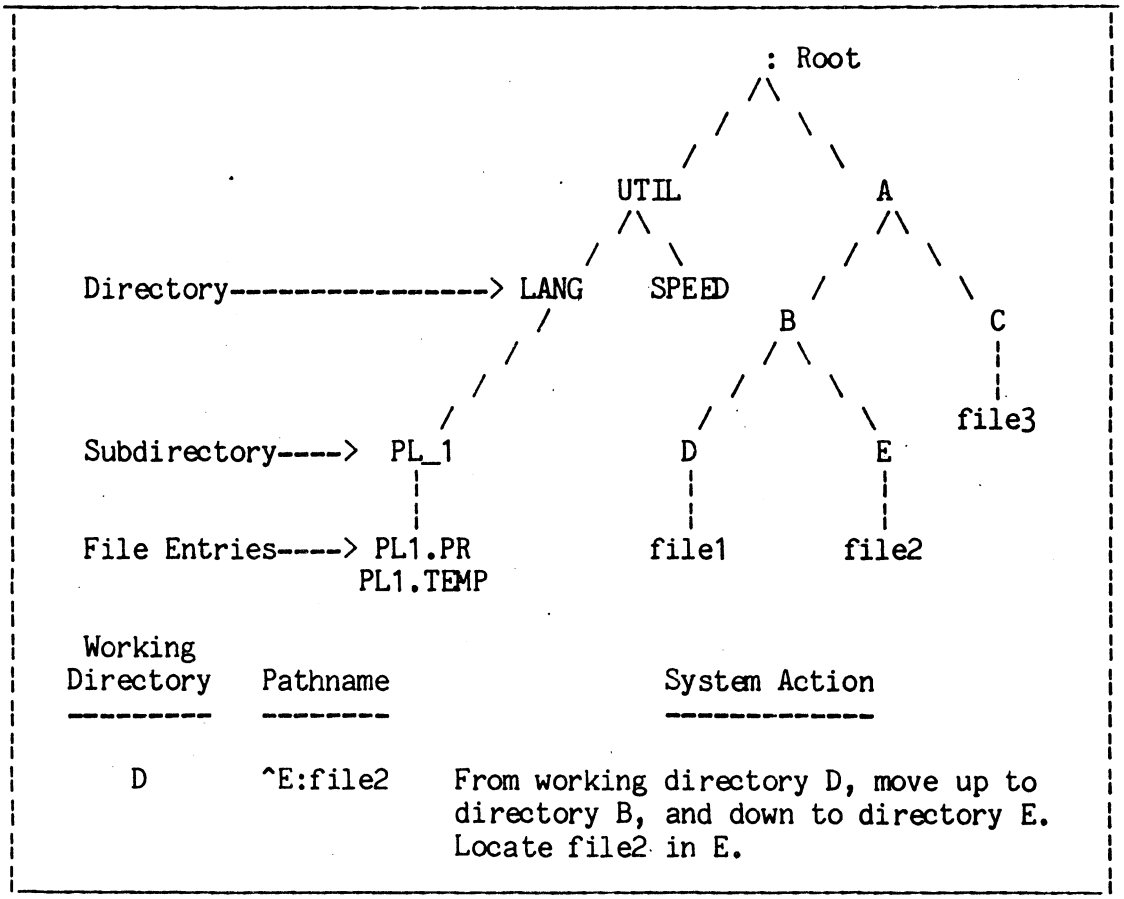


Figure 5-4. Directory Structure

Assuming that the directory structure is the one shown in Figure 5-4, and that D is the working directory, issuing the ?GNAME system call would yield the following results:

Your Input	?GNAME Output
file1	:A:B:D:file1
^E	:A:B:E
^	:A:B
^E:file2	:A:B:E:file2

The ?GRNAME system call is similar to the ?GNAME system call, except that it returns the complete pathname of a generic file. YOu cannot use the ?GNAME system call to get the "true" pathname of a generic file. For example, given the input pathname @DATA, the ?GNAME system call would return :PER:DATA as the complete pathname, even though the complete pathname of the file is actually :UDD:USER:DATA. In this case, the ?GRNAME system call would return :UDD:USER:DATA.

Link Entries

A link entry (file type ?FLNK) is a file that contains a pathname to another file.

Link entries act as a pathname shorthand. When you specify a link entry in a pathname, AOS/VS substitutes the contents of the link for its name. In figure 5-4, for example, you can create a link called G that contains the pathname :A:B:D. Thereafter, whenever you refer to link G, AOS/VS resolves that link to :A:B:D. Link entries work differently as input to the system calls ?CREATE and ?DELETE. The next section discusses two exceptions.

A prefix is optional in a link-entry pathname. If there is a prefix, AOS/VS starts resolving the pathname at the directory that the prefix specifies. If there is no prefix, AOS/VS starts resolving the pathname at the link entry's parent directory.

In addition to acting as pathname abbreviations, link entries serve another purpose. A process can access a file without copying the actual file into its working directory. To do this, the process must include the appropriate link entry in its working directory.

Another way to avoid copying a file is to include the directory that contains the file in a search list. This works only if no other directory in the search list contains a file with the same name. The ?SLIST system call sets a search list for the calling process. Note that a search list cannot contain more than eight pathnames.

One of the entries of a link can be another link. This is called a link-to-link reference. Too many link-to-link references can cause the system call that is referencing the link to overflow its stack. If a stack overflow does occur, AOS/VS returns the stack overflow error message, ERSTO.

Because the number of link-to-link references that you can use depends on both your program and AOS/VS, it is impossible to predict how many link-to-link references will cause a stack overflow. Therefore, if a stack overflow occurs while you are using a pathname, examine the pathname. Then, if the pathname contains link-to-link references, remove them.

To find out what a particular link entry represents, issue the ?GLINK system call. The ?GLINK system call is particularly useful if you cannot decide whether to delete an existing link entry and/or create a new one.

Use of ?CREATE and ?DELETE System Calls on Link Entries

You can use the ?CREATE and ?DELETE system calls to create and delete link entries just as you would other files. When you apply these calls to link entries, however, AOS/VS creates or deletes the link itself, not its contents.

For example, suppose in directory :A you create link entry B, which contains the pathname D:E. If you issue ?DELETE against pathname :A:B, AOS/VS deletes link B without resolving its contents. Directories D and E remain intact, however, as does directory A. (Directory A is simply the "path" to link entry B.)

AOS/VS resolves a link if it is simply part of the pathname of a file you wish to create or delete. Consider the preceding example. If you issue ?DELETE against file C in the pathname :A:B:C, AOS/VS resolves link B to :D:E, and then deletes file C in directory :D:E. Again, directories A, D, and E remain intact.

File Access

To read, write, or execute a file, you must have the proper access to it. Under AOS/VS there are five kinds of access for every file:

- Owner access
- Write access
- Append access
- Read access
- Execute access

Table 5-6 lists the access privileges and their meaning for directories and all other file types.

Table 5-6. File Access Privileges

Privilege	For Nondirectory Files	For Directories
Owner Access	Allows you to: o read and change the file's ACL. o read the filestatus and permanence of the file. o set the permanence of the file. o get a complete pathname of the file. o rename or delete the file. o create a User Data Area (UDA) for the file and read or write to it.	Allows you to: o read and change the ACL of the directory. o initialize an LDU if you have owner access to the LDU's root directory. o rename or delete the directory.
Write Access	Allows you to: o modify the data in the file. o read the filestatus and permanence of the file. o get a complete pathname of the file. o create a User Data Area (UDA) for this file and write to it.	Allows you to: o create, delete, and rename the directory's files. o read and change each file's ACL. o read and set the permanence of the directory's files. o initialize and release an LDU in the directory.
Append Access	Allows you to: o read the filestatus and permanence of the file o get a complete pathname of the file.	Allows you to: o add files to the directory. o initialize an LDU in the directory.

Table 5-6. File Access Privileges continued

Privilege	For Nondirectory Files	For Directories
Read Access	Allows you to: o examine the data in the file. o read the filestatus and permanence of the file. o get a complete pathname of the file. o read a User Data Area for the file.	Allows you to: o list the name, filestatus, and permanence of each file in the directory. o use this directory as a working directory. o read each file's ACL. o get the contents of a link entry in the directory.
Execute Access	Allows you to: o execute the file. o read the filestatus and permanence of the file. o get a complete pathname of this file.	Allows you to: o name the directory in a pathname (this is essential if you wish to name the directory or refer to it.) o make the directory your working directory. o resolve a pathname using a link in the directory.

Execute access is the most essential kind of access to directories, because it allows you to use the directory name in a pathname. Without this privilege, all other access privileges to a directory are meaningless.

Owner access to a directory allows you to initialize logical disks in that directory with the ?INIT system call. (See "Logical Disks" in this chapter.)

If you are writing to a file with the ?WRITE system call, you must have both read and write access to it. If, on the other hand, you are writing to it with the ?WRB system call, you only need write access. When reading from a file with ?READ or ?RDB you need read access to it.

Access Control Lists

AOS/VS maintains a unique access control list (ACL) for every file that is not a link entry. An ACL is an ordered list of the users who can access the file and the type of access granted to each user. When you try to read, write, or execute a file, AOS/VS checks your username against each entry in the parent directory's ACL and against each entry in the file's ACL.

For example, if the ACL for file :TJ:GLOSSARY.PR allows username TJ Read and Execute access, as well as Execute access to the directory TJ and the root, then users that log on under username TJ can execute the file GLOSSARY.pr and read its data. However, these same users cannot delete the file GLOSSARY.PR, or change its ACL, unless they also have WRITE access to GLOSSARY.PR's parent directory, TJ.

There are several ways to set an ACL for a file or a directory. One way is to use the CLI command ACL. Another way is to define a file's ACL from your source code via the ?CREATE, ?SACL, or ?DACL system calls. The ?CREATE system call allows you to define the ACL along with the other specifications for the new file or directory. The ?SACL system call allows you to set an ACL for a file or directory.

To determine a particular file or directory's ACL, issue the ?GACL system call. The ?GTACP system call is more specific in that it returns the ACL for a specific file and username. If you are in Superuser mode, the ?GTACP system call allows you to find out if a given user has access to a particular file.

Depending on your input parameters, the ?DACL system call sets, clears, or examines the default ACL mode for one or more processes that have specific usernames. Default ACL mode is process specific, rather than file specific. For example, a process can issue the ?DACL system call to turn on default ACL mode and define a specific ACL for all files it will later create. A default ACL defined with the ?DACL system call exists until the ?DACL caller terminates or until it redefines that default by issuing another ?DACL system call.

The ?CREATE, ?DACL, and ?SACL system calls take the following bit masks as ACL specifications:

Mask	Meaning
?FACA	Append access
?FACE	Execute access
?FACR	Read access
?FACW	Write access
?FACO	Owner access

ACL Templates

When you create an ACL, you can define access privileges for specific usernames, or you can use ACL templates to represent certain username/character combinations. Table 5-7 lists the valid ACL templates and the character combinations they represent.

Table 5-7. Valid ACL Templates

Template	Meaning
+	Matches any character string. For example, the ACL username specification PA+ matches any character string that begins with PA, such as PAT, PAM, PAUL, PA_B, and PA.M.
-	Matches any character string except those that contain one or more periods. For example, PA- matches PAT, PAM, PAUL, and PA_B, but not PA.M.
*	Matches any single character except the period. For example, PA* matches PAT and PAM, but not PAUL, PA_B, or PA.M.

AOS/VS scans ACL entries from left to right. Thus, you should not place the plus sign (+) template first, because it will override more specific templates or usernames. For example, the following ACL specification begins with +<?FACR> (the zeros are delimiters), which gives all users Read access only (?FACR), even though the second element assigns Owner access to a specific username (PAT):

```
+<0><?FACR>PAT<0><?FACO><0>
```

The Permanent Attribute

Any user with Owner access can easily delete a directory of file. Therefore. AOS/VS provides the permanent attribute for additional protection.

The permanent attribute prevents users from deleting a directory or file, regardless of its ACL. The ?SATR system call sets the permanent attribute. or removes it, if the target directory or file already has permanent status. The ?FSTAT system call returns various information about a directory or file, including whether or not it has the permanent attribute.

If you set the permanent attribute for a file, you should also set it for the file's parent directory. Otherwise, a process can delete the file by deleting the parent directory.

Logical Disks

A logical disk (LD) is one or more physical disk units that you treat as a single logical unit. Each file is completely contained within a single LD.

Each LD is a complete collection of disk space that contains a directory tree structure. In fact, each LD has a single directory called the local root. It is the local root that acts as the foundation for constructing a directory structure. You specify an ACL for the local root when you construct the LD.

When you bootstrap AOS/VS, you select one LD as the Master LD. The root of this LD becomes the system root. which is identified by the colon (:).

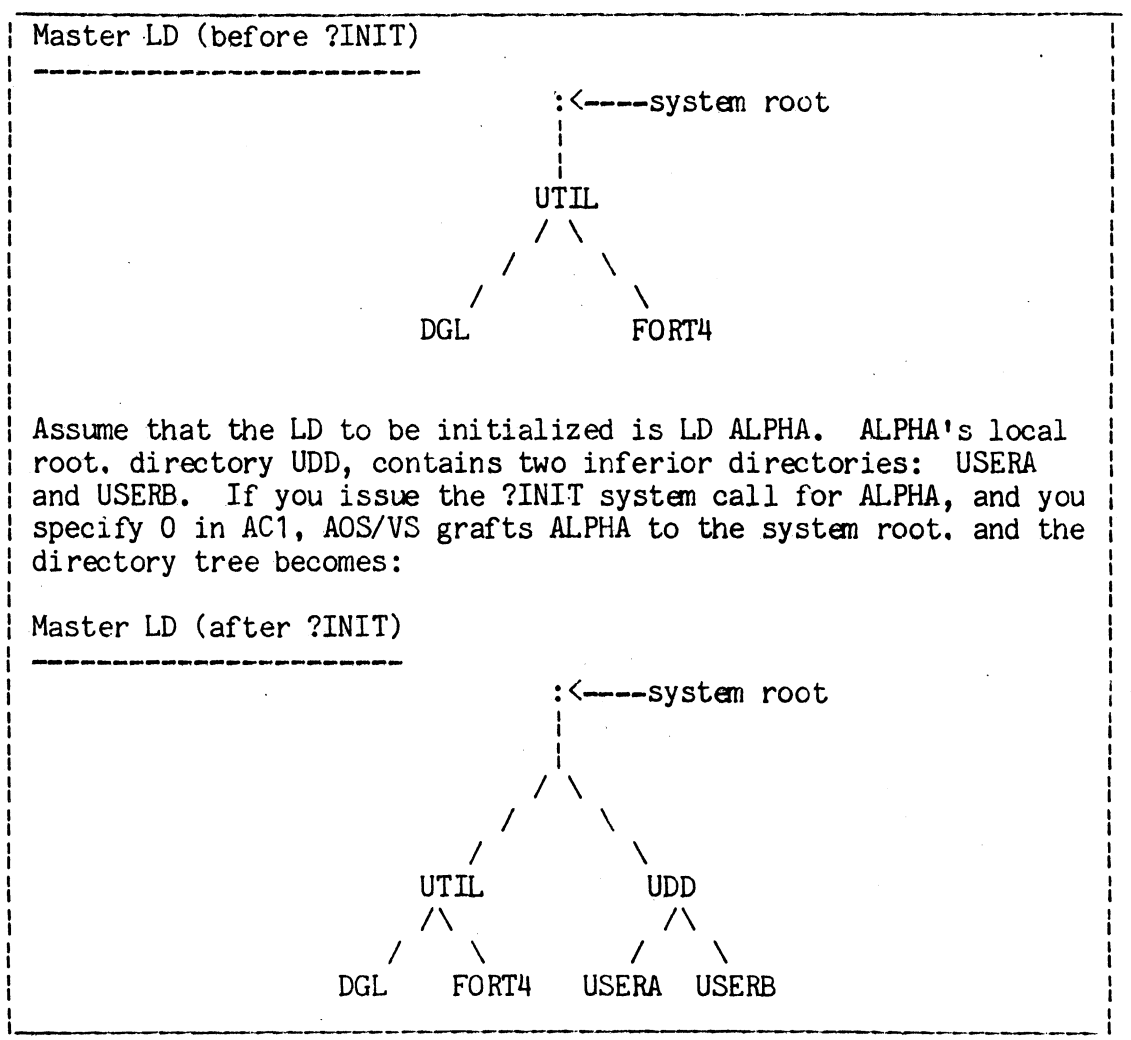


Figure 5-5. Initializing a Logical Disk

Before you can use any LD except the Master LD, you must initialize it with the ?INIT system call or the CLI INITIALIZE command. To use the ?INIT system call, you must have Owner access to the LD's local root directory. The ?INIT system call grafts the LD's local root to a specified directory. (See Figure 5-5.)

The disk structure within each LD can have more than eight directory levels, excluding the local root (directory level zero within that LD). Up to eight different LDs may be grafted upon each other, starting from the system root. If each LD were grafted at the eigth directory level of the previous ld, then the actual maximum directory level attainable under the AOS/VS disk structure is 64 directory levels (excluding the system root).

An LD remains initialized until you release it by issuing the ?RELEASE system call. You may want to release an LD to remove its component volumes from the disk drives and mount other volumes onto those disk drives.

Disk Space Control

You can control how AOS/VS allocates disk space by designating certain directories in an LD as control point directories (CPDs). CPDs function exactly like other directories, but they contain two additional variables:

- Current space (CS), which is the amount of space currently allocated.
- Maximum space (MS), which is the maximum amount of space available in the directory.

Current space (CS) is the current number of disk blocks occupied by the CPD and all its inferior files, except for files in an inferior LD. When you create a CPD, AOS/VS initializes CS to zero. Maximum space (MS) is the maximum number of disk blocks available to the CPD and all its inferior files, except for files in an inferior LD. To specify MS, issue the ?CPMAX system call.

Each LD's local root is a CPD. Thus, a local root's CS is the total space currently used in the LD, and its MS is the maximum number of disk blocks the LD can contain.

CPDs restrict a file's disk space to a predefined limit. When a file requires more disk space, AOS/VS first checks the MS and CS of its CPD. AOS/VS allocates more disk space to that file only if it can do so without causing the CPD's CS to exceed its MS. If a file's pathname contains more than one CPD, AOS/VS compares the CS to the MS at every point. starting with the CPD closest to the file.

Figure 5-6 shows a simple directory structure with two CPDs.

Assume that the LD root and directory CP1 in Figure 5-6 are CPDs. If file1 needs an additional n blocks, AOS/VS first adds n to the CS of CP1, which is the control point closest to file1. If CS+n is greater than the MS for CP1, any attempt to allocate additional space for file1 will fail.

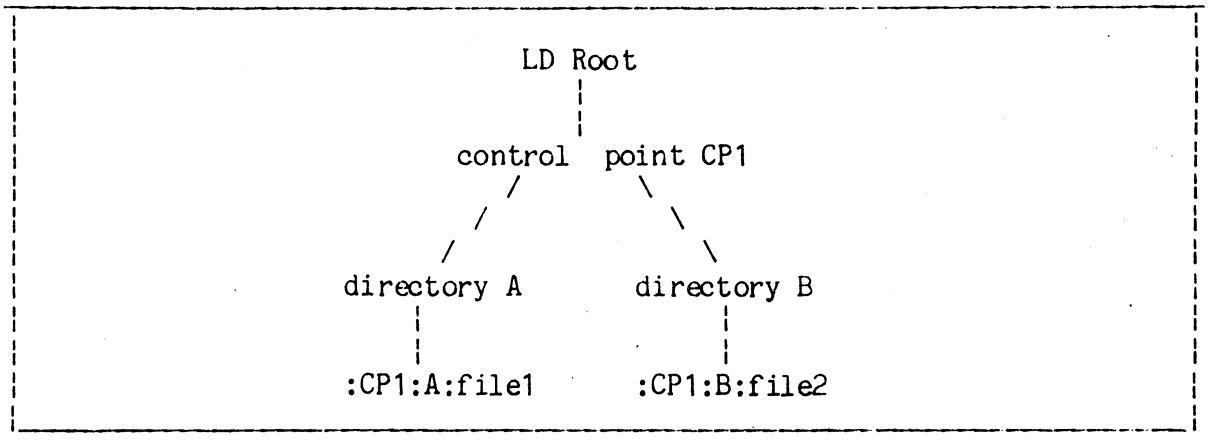


Figure 5-6. Control Point Directories (CPDs)

When you create a CPD, AOS/VS does not initially check its MS against those of the other CPDs in the file tree. In fact, AOS/VS permits oversubscription, as long as the tree's total CS does not exceed the MS in any superior control point, up to and including the local root. Note that you cannot set a CPD's MS to less than its CS.

File I/O

Writing to or reading data from a device is called file input/output (I/O). before you can use file I/O system calls, you must understand file I/O. Therefore, this is divided into the following topics:

File I/O Concepts

This defines blocks, records, and channels, describes how AOS/VS stores and accesses files, and describes the steps that you normally perform to use file I/O.

Blocks and Records

AOS/VS stores files (data) in physical units called blocks. In general, there are two methods of accessing these files.

- Block I/O
- Record I/O

Block I/O system calls allow you to directly access the blocks in which your files are stored. Blocks vary in size from device to device. Therefore. when you access a file using a block I/O system call, you must specify the block size, the starting block number, and exactly how many blocks you want to transfer.

Record I/O system calls allow you to indirectly access the blocks in which your files are stored. WHen you issue a record I/O system call, AOS/VS sees the file as a collection of logical units called records. Then, AOS/VS selects the correct file and records based on the record type that you specified when you created the file. The record type defines the format of a file's records. AOS/VS uses this information along with other parameters, such as the file's pathname, to associated physical blocks on a device with a ceratin file and its records.

Channels

File I/O, which includes both block I/O and record I/O, takes place across paths called channels. When you issue a system call to open a file, AOS/VS assigns the file a channel and a unique channel number to identify that channel. The mnemonic ?LOCHN represents the lowest possible channel number and the mnemonic ?HICHN represents the highest possible channel number

To disassociate a channel number from a file, close the channel. When you close a channel, it becomes unavailable for further file I/O. AOS/VS assigns a new channel number every time you reopen the file.

File I/O Operation Sequence

Table 5-8 summarizes the usual sequence of operations for record I/O and block I/O.

Table 5-8. File I/O Operation Sequence

Operation	Record I/O Call	Block I/O Call
1) Open the file.	?OPEN	?GOPEN
2) Read or write.	?READ/?WRITE	?RDB/?WRB or ?BLKIO
3) Close the file.	?CLOSE	?GCLOSE

Many file I/O system calls require a packet of file specifications. In general, the ?OPEN, ?READ, ?WRITE, and ?CLOSE system calls use similar specification packets, as do the ?GOPEN, ?RDB, ?WRB, and ?GCLOSE system calls. However, some packet offsets and masks apply to certain system calls only. For example, the Exclusive Open option applies to the ?OPEN system call, but not to the ?READ, ?WRITE, or ?CLOSE system calls. At various points in the file I/O cycle, you can change certain information in the file specification packet.

You can open a file repeatedly without issuing a ?CLOSE system call after each ?OPEN system call. AOS/VS maintains an open count for each ?OPEN system call and closes the file only when the open count equals zero.

The creation option in the ?OPEN packet allows you to simultaneously create and open certain file types. Table 5-9 lists the file types you can create with this option. When you select the creation option and default the file type parameter in the ?OPEN packet, AOS/VS creates the new file as a user data file (type ?FUDF). You generally use user data files for storing text, data, and variables. User data files are not executable program files.

Unless you have exclusively opened a file (an option available in the ?OPEN packet), more than one process with write or read access can update any record in the file simultaneously.

Table 5-9. File Types You Can Create with the ?OPEN System Call

File Type	Meaning	Comments
?FUDF	User Data File	This is the default file type. (To take this default, set the right byte of offset ?ISTO to 0.)
?FTXT	Text File	This type of file should contain ASCII code.
?FPRV	32-bit Program File	This type of file is an executable 32-bit program file; it should contain linked, executable code.
?FPRG	16-bit Program File	This type of file is an executable 16-bit program file; it should contain linked, executable code.
?FDIR	Disk Directory	If you use the ?OPEN system call to create this type of file, you can default only the following parameters: hash frame size, maximum number of index levels, and access control list.
?FIPC	IPC File	This type of file directs AOS/VS to create an IPC file or open an existing IPC file to allow full-duplex communications between two processes.
?FCPD	Control Point Directory	Although you can use the ?OPEN system call to create a control point directory, we recommend that you use the ?CREATE system call instead.

By issuing the ?UPDATE system call, you can guarantee the integrity of all previous ?WRITE system calls issued against a file if the system crashes while that file is still open. The ?UPDATE system call flushes memory-resident file descriptor information to disk. Note, however, that the ?UPDATE system call does not write a file's data to disk, just its file descriptor information. File descriptor information includes the file's User Data Area (UDA).

File Pointer

To manage repeated I/O sequences, AOS/VS maintains a separate file pointer for each open channel. The file pointer keeps track of the character position for the next read or write sequence on a file.

When you open a file, AOS/VS positions the file pointer, by default, to the first character (byte) in the file. AOS/VS then moves the file pointer forward as it reads or writes each record or byte string. Three ways to override the default position of the file pointer are:

- Select the Append option in the ?OPEN packet (?APND in offset ?ISTI).

  This option moves the file pointer to the last byte in the file, which allows you to append data with the ?WRITE system call.

- Manipulate the file pointer in the ?READ or ?WRITE packet during an I/O sequence.

- Issue the ?SPOS system call to reposition the file pointer without performing I/O.

The ?GPOS system call returns the current position of the file pointer. The ?TRUNCATE system call deletes all data that follows the file pointer in a disk file, and writes two end-of-file marks after the file pointer in a magnetic tape file.

Block I/O

Block I/O is the process of reading or writing files that exist on a device, in physical units called blocks. The sizes of these blocks vary from device to device. (See "I/O Devices and Generic Filenames" for information on devices.)

The ?GTRUNCATE system call allows you to reduce the size of a disk file that is currently open for block I/O.

The ?ALLOCATE system call allocates blocks for specified data elements and zeroes those data elements that do not actually exist. You can use the ?ALLOCATE system call to make sure that subsequent I/O will not cause a calling process to exceed its control point directory's maximums. (See Chapter 4 for information on control point directories.)

To perform block I/O on a file, you must know the number of blocks you want to transfer (block count), the starting block number, and the block length (number of bytes per block). You specify this information in a block I/O packet. (See the description of the ?RDB/?WRB and ?BLKIO system calls for the packet structures.)

The ?RDB/?WRB and ?BLKIO calls are very similar, except that ?BLKIO includes additional functionality for reading the next allocated data element in the file. ?RDB reads an element whether it is allocated or not. As a result, this ?BLKIO option makes block reading very fast when you have long files with many unallocated elements.

Physical block lengths vary from device to device. To find the block length for a particular device, refer to the 'Programmer's Reference Peripherals' manual. The standard block length for disks is 512 bytes. Magnetic tape block length is whatever length you specify when you issue the ?GOPEN system call. You must specify an MCA unit's block length with each read or write operation.

Physical Block I/O

AOS/VS supports physical block I/O for disks. Physical block I/O is more primitive than block I/O. To perform physical block I/O, you must issue either the system calls ?PRDB (read physical blocks) and ?PWRB (write physical blocks) or the ?BLKIO system call with the physical block I/O option.

Physical block I/O allows you to bypass AOS/VS's usual retries for disk errors. You can also use the ?PRDB/?PWRB and ?BLKIO system calls to check for bad blocks on a disk, or for problems with an I/O device. When AOS/VS encounters a bad block (transfer error) while it is executing one of these system calls, it takes the normal return, but flags the bad block and reports the reason for the error in the packet. When a device error occurs during these system calls, AOS/VS aborts and returns the device error code to the packet.

In summary, physical block I/O differs from block I/O in that physical block I/O has:

- No remapping.

  If a physical block transfer fails because of a bad block, AOS/VS continues to read or write the additional blocks, and then takes the normal return from the ?PRDB/?PWRB or ?BLKIO system calls.

- No retries.

  If a physical block transfer fails, AOS/VS does not try to read or write the block(s) again. (This is different from block I/O in which AOS/VS retries the block read or block write.)

- No ECC corrections.

  If data errors occur during a physical block transfer, AOS/VS completes as much of the transfer as possible, and takes the normal return from the system call.

?PRDB/?PRWB and ?BLKIO with the physical block read option work in conjunction with the assembly language block status instructions DIA, DIB, and DIC. (For details on the syntax and function of these instructions and the error-correction codes for devices, refer to the 'Programmer's Reference Peripherals' manual.)

Record I/O

Record I/O is the process of reading or writing files that exist on a device, in logical groupings called records. There are four types of records:

- Dynamic-length

  When you read to or write from a file that contains dynamic-length records, you must specify the length of each dynamic record in that file.

- Fixed-length

  When you read to or write from a file that contains fixed-length records, you must specify a record length that is common to every record in that file.

- Data-sensitive

  When you read to or write from a file that contains data- sensitive records, you must specify the maximum record length in offset ?IRCL of your I/O packet. Then, AOS/VS transfers data until it either encounters a delimiter or reaches the maximum record length that you specified. In the latter case, your I/O system call fails and returns error code ERLTL (line too long) in AC0.

  The default delimiters are: NEW LINE, CR (carriage return), NULL, or FORM FEED. You can override the default delimiters by specifying a 16-word delimiter table when you open the file or by issuing the ?SDLM system call after you open the file. The ?GDLM system call returns the delimiter table for a file.

- Variable-length

  When you read to or write from a file that contains variable-length records, you must specify the length of each record in a 4-byte ASCII header. This means that each record in a file can be a different length.

Device Names

During system initialization, AOS/VS records the names of all available I/O devices in its peripheral directory, :PER. Because the standard device names are not reserved words, you must precede each one with the prefix @. As a pathname template. @ represents the:PER directory. Thus, when you use @ as a filename prefix, AOS/VS recognizes the filename as either a device name or a generic filename. See Table 5-10 for a complete list of the AOS/VS devices and their device names.

Generic Filenames

The peripheral directory (:PER) also contains generic filenames. Generic filenames are names that refer to devices or files of a particular type, such as input files, output files, and list files.

Generic filenames represent common classes of devices and files. BY coding with generic filenames, you can change the filenames associated with the generic names without recoding the program. For example, you might code a program with the generic filename @LIST to represent the list file. Then, before you execute the program. you can set the list file to a specific filename.

Table 5-10 AOS/VS Devices and Device Names

Name	Device
ALM	Asynchronous Line Multiplexor.
@CON0	System Control Processor (SCP).
@CON2 through @CONn	DASHER display consoles or asynchronous communications lines 1 through n on Lines 0 through n-2 (for example, CON2 is on Line 0, CON3 is on Line 1, etc.).
@CRA and @CRA1	First and Second Card Readers.
@DKB0 through @DKB6	6063 or 6064 fixed-head disk unit 0 through 6.
@DPn0 through @DPn17	Moving-head disk units 0 through 7 on the first controller, and 10 (octal) through 17 (octal) on the second controller where n is a single alphabetic character that indicates the disk unit type. (Refer to the 'Managing AOS/VS' manual for descriptions of these types.)
@LPB, @LPB1 through @LPB7	Data channel line printers 0 through 17.
@LMT	Labeled magnetic tape.
@LFD	Labeled floppy diskette
@MCA, @MCA1	Multiprocessor communications adapter controllers (unit names).
@MTN0 through @MTN17	Magnetic tape controller units 0 through 7 on the first controller, and 10 (octal) through 17 (octal) on the second controller.
@PLA and @PLA1	First and second digital plotters.

Table 5-11 lists the six generic filenames and the files they represent. Like device names, generic filenames requires the @ prefix.

Table 5-11 Generic Filenames

Filename	Refers To
@CONSOLE	Any interactive device associated with a process (usually a CRT console).
@LIST	A mass output file.
@INPUT	A command input file.
@OUTPUT	Any output file.
@DATA	Any mass input file.
@NULL	A file that remains empty.

Like device names, generic filenames reqiure the @ prefix.

For an interactive process, your console usually serves as both the @INPUT and the @OUTPUT file. @NULL is not a strict generic filename, in that you cannot associate it with an actual pathname. When you write data to the @NULL file AOS/VS does not output the data to any other file or device. When you try to read @NULL file, AOS/VS returns an end-of-file condition.

When you create a process with the ?PROC system call, you can set any generic filename except @NULL to a specific pathname. For example, you can set a process's @LIST file to the following pathname:

:UDD:USERNAME:MYDIR:LPT

where:

- o MYDIR is the current working directory
- o LPT is the list file

The ?PROC packet provides the following parameters for generic filename associations:

Offset	Generic Filename
?PCON	@CONSOLE
?PIFP	@INPUT
?POFP	@OUTPUT
?PLFP	@LIST
?PDFP	@DATA

The ?PROC packet also allows the ?PROC caller to pass its own generic filename associations to a newly created son.

Usually. AOS/VS copies the data it reads from the @INPUT file to the @OUTPUT file. However. If @INPUT and @OUTPUT are both consoles, then the @INPUT function echoes data to the @OUTPUT console. The generic filenames @INPUT, @OUTPUT and @LIST acquire all the characteristics of the devices associated with them. For example, if you associate the generic @LIST file with the line printer, a separate listing prints each time you open and close @LIST or any other file.

The @DATA file is similar to the @INPUT file, except that it does not copy data to the @OUTPUT file.

Multiprocessor Communications Adapters

AOS/VS supports type 42006 Multiprocessor Communications Adapters (MCAs). The I/O protocol that AOS?VS uses for these devices is the same as MCA protocol that Data General's AOS,RDOS, and RTOS operating systems use.

Each MCA enables two or more central processing units (CPUs) to communicate across a data channel. The MCA units are connected by hardware links. A single MCA can connect a CPU to as many as 14 other CPUs. By adding a second MCA (MCA1), you can connect another 15 CPUs.

Each MCA link consists of two devices; an MCAT, which transmits data from one processor to another. and an MCAR, which receives the data. The MCA pathname takes the following forms:

@MCAT:n
@MCAT1:n
@MCAR:n
@MCAR1:n

where n is the number of the MCA link, in the range fom 0 through 15

The link number indicates which remote CPU you are communicating with when your local CPU is linked to more than one remote CPU.

Character Devices

Character devices are devices that perform I/O in bytes. CRT and hard-copy consoles are typical character devices.

Charcter devices can operate in one of two modes: binary mode or text mode. Text mode is the default, but you can specify binary mode when you issue a ?READ or a ?WRITE system call against the device. When a character device is in binary mode, AOS/VS recognizes only delimiters. Therefore. AOS/VS passes each byte of any other character without interpretation.

When a character device is in text mode, AOS/VS interprets each byte according to the device's characteristics, or distinguishing features. The device characteristics include:

- The line length of the output.
- Whether the device is ANSI standard of non-ANSI standard.
- Whether the device echoes characters.
- Whether the device uses hardware tab stops of form feeds.

To qualify text mode further, you can set the character device to the Page Mode characteristic. When a character device is in page mode, AOS/VS automatically stops its output at the line length (lines per page) you specify, or when it encounters a FORM FEED character.

To display the next page while the device is in page mode, type the CTRL-Q console control character. (See "Console Format Control" in this chapter for a description of the console control characters.)

The ?GCHR and ?GECHR system calls return the current characteristics of a character device and the extended characterstics of a character device, respectively. The ?SCHR and ?SECHR system calls set or remove device characteristics or extended device characteristics, respectively, depending on your input specifications.

To define characteristics for a character device, you must set certain characteristic flags in a 5-word buffer in AC2 when you issue the ?SCHR system call or the ?SECHR system call. Usually, you will probably set characteristic flags in the first three words of this buffer. If you set characteristic flags in the fourth or fifth words (words 3 and 4), then you are setting an "extended" characteristic.

The extended characteristics control XON/XOFF data flow over console lines. They also control characteristics such as baud rates for Intelligent Asynchronous Controllers (IACs). For details, see the description of ?SECHR.

The initial operator process (PID 2) can override characteristics that were set during the system-generation procedure. However, if you are not PID 2, you can only set the modem control and monitor ring indicator characteristics during the system-generation procedure. (For more information on the system-generation procedure, refer to the "How to Generate and Run AOS/VS" manual.)

The ?SEND system call allows you to pass a message from a process to a console without opening and closing the console. This means that you can pass messages from real-time processes without consoles to a system process, such as OP CLI.

Full Duplex Modems

A full-duplex modem is a communications device that translates analog signals to digital signals, and vice versa, over telephone lines. AOS/VS supports I/O over full-duplex modems, which AOS/VS treats as character devices.

You must define modems and set the modem control characteristic (?CMOD) during the AOS/VS system generation procedure. You cannot set or remove this characteristic with the ?SCHR system call.

AOS/VS supports both auto-answer modems and non-auto-answer modems. Table 5-12 lists the flags used in modem operation.

Table 5-12. Modem Flags

Flag	Meaning
CD	Carrier detect; if set, the communications line is conditioned for data transmissions.
DSR	Dataset ready; if set, AOS/VS is connected to a communications line.
DTR	Data console ready; if set, AOS/VS is ready to connect with a remote user.
RTS	Request to send; if set, AOS/VS has made a request to send data.

Auto-Answer Modems

The following steps summarize the operating sequence for auto-answer modems:

1. During modem initialization, both DTR and RTS are off, which indicates that the modem is off.

2. Upon execution of the first ?OPEN system call, AOS/VS sets DTR and RTS, and changes the modem status to on.

3. No I/O will take place until both DSR and CD are on, which indicates that the modem is connected.

4. The I/O call terminates with an error return if DSR lapses during the I/O sequence, or if CD lapses for more than 5 seconds.

Non-Auto-Answer Modems

If you are receiving data over a non-auto-answer modem, and you are not PID 2, which can override characteristics set during the system generation procedure, you can select the Monitor Ring Indicator characteristic during the system generation procedure. This characteristic appears as parameter ?CMRI in the second device characteristic word. Like the ?CMOD characteristic, you can only set the ?CMRI characteristic during the system generation procedure, unless you are PID 2.

AOS/VS uses the Monitor Ring Indicator to detect incoming calls (rings) to a non-auto-answer modem. If you select the ring-idicator option, AOS/VS begins monitoring the ring indicator as soon as you open the local modem-controlled device. When a remote user places a call to your device, the hardware signals a modem interrupt and sets the ring indicator. AOS/VS then raises the DTR flag and sets a timer. If AOS/VS does not detect a DSR signal and a valid carrrier signal within 5 seconds of the modem interrupt, it posts a disconnect against the line. When this occurs, yu must close the modem-controlled device and re-open it.

The following steps summarize the operating sequence for non-auto-answer modems with Monitor Ring Indicator option:

1. During modem initialization, both DTR and RTS are off, which indicates that the modem is off.

2. Upon execution of the first ?OPEN system call to the modem-controlled device, AOS/VS begins monitoring the ring indicator, provided you selected this characteristic (?CMRI) during the system generation procedure.

3. When a remote user places a call, the MV hardware signals an interrupt for the local modem and sets the ring indicator; AOS/VS then sets the DTR flag and starts the ring indicator timer.

4. AOS/VS begins checking for a DSR signal and a CD signal; if these occur within 5 seconds of the modem interrupt, the modem is connected; otherwise, the system posts a disconnect against the line.

5. No I/O takes place until the modem is connected.

6. I/O terminates with an error return if the modem becomes disconnected during the I/O sequence; this state occurs when either the DSR flag changes from on to off, or the carrier signal lapses for longer than 5 seconds.

NOTE: If you have selected the ring-indicator option, you cannot use the communications line for manual dial-outs. To use the line for manual transmissions, you must generate it again, without the ring-indicator option.

Character Device Assignment

AOS/VS allows you to open a device for the exclusive use of one and only one process by "assigning" the device to that process. You can do this explicitly by issuing the ?ASSIGN system call, or you can do this implicitly by opening the file. You cannot issue the ?ASSIGN system call against a file that is already open.

If you assign a file with the ?ASSIGN system call, you must issue the ?DEASSIGN system call to break the assignment. If you assign a device with the ?OPEN system call, you can break the assignment by closing the device or by terminating the process. A process can open a device more than once without breaking an ?OPEN system call assignment; AOS/VS does not break the assignment until the last ?CLOSE system call (when the ?OPEN system call count drops to zero).

Device assignment works somewhat differently for consoles. All son processes can share their father's console, even if the console was specifically assigned to the father. However, only the most recently created son can actually control the console bye issuing ?OPEN, ?CLOSE, ?ASSIGN, ?RELEASE, ?GCHR, ?GECHR, ?SCHR, and ?SECHR system calls against it. The father process and all other sons can issue only ?READ and/or ?WRITE system calls against an assigned console.

Line-Printer Format Control

When you write a file to a data channel line printer controlled by EXEC, you can tailor the format of the output by creating a user data area(UDA) for the file. The ?CRUDA system call creates a UDA. The ?RDUDA and ?WRUDA system calls read and write UDA infomation, respectively. Typically, you use UDAs to specify file formats, although you can use them for other purposes.

In addition to the ?CRUDA system call, you can also use the AOS/VS Forms Control Utility (FCU) to create UDAs for format specifications. To do this, you must perform the following steps:

1. Create a file with the name of the UDA that you want to create.

   This file can contain format specifications or, if you wish, it can be empty.

2. Execute FCU.

3. Move the newly created UDAs to the :UTL:FORMS directory so that EXEC can access them.

If you want the contents of a particular UDA to override EXEC's default format specification, use CLI switch /FORMS when you print the file on the line printer. If you omit the ?FORMS switch or if the file has no format specifications, AOS/VS uses the current default EXEC format settings.

Console Format Control

Several control characters and control sequences allow you to control the output that prints on your console.

A control character is any character that you type while you press the CTRL key at the same time. By default, AOS/vs does not pass control characters to your program. However, if you want to override this default, set binary mode or type CTRL-P immediately before you type a control character. Either method will cause AOS/VS to pass the control character to your program. Table 5-13 lists the control characters and what they do.

Table 5-13. Control Characters and Their Functions

Control Character	Function
CTRL-C	Begins a control sequence.
CTRL-D	An end-of-file character; terminates the current read and directs AOS/VS to return an end-of-file condition.
CTRL-O	Suppresses output to your console until you type CTRL-O again. (If AOS/VS detects a BREAK condition then its output resumes immediately.)
CTRL-P	Signals AOS/VS to accept the next character as a literal, not as a control character.
CTRL-S	Freezes all output to your console, but does not discard it. (To disable CTRL-S, type CTRL-Q)
CTRL-Q	Disables CTRL-S; if the device is in page mode, CTRL-Q displays the next page
CTRL-U	Erases the current input line on your console
CTRL-T and CTRL-R	Reserved for future use by Data General. (Currently these control sequences do nothing. However, if you precede either one with a CTRL-P, AOS/VS passes them to your program.)

A control sequence is a CTRL-C immediately followed by any control character from CTRL-A through CTRL-Z. What happens when you type the second control character depends on the internal state of the process with which the console is associated. If the process has not explicitly redirected the control character, then AOS/VS ignores the control sequence and treats the second control character as it normally would. However, AOS/VS ignores control sequences that do not have a default action.

Table 5-14 lists the control sequences and what they do.

Table 5-14. Control Sequences and Their Functions

Control Sequence	Function
CTRL-C CTRL-A	Generates a console interrupt (provided you used the ?INTWT system call to define a console interrupt task).
CTRL-C CTRL-B	Generate a console interrupt and aborts the current process
CTRL-C CTRL-C	Echoes the characters ^C ^C on the console, and empties your type-ahead buffer. (This is useful when you want to revoke a command you have typed ahead.)
CTRL-C CTRL-D through CTRL-C CTRL-Z	Reserved for use by Data General

The IPC Facility as a Communications Device

You can use IPC files as a communications device, and perform I/O against them. When you perform I/O against an IPC file, AOS/VS buffers the messages in first-in/first-out order. To use the IPC facility as a communications device AOS/VS performs the following steps:

1. The calling process creates an IPC file entry with the ?OPEN creation option (bit ?OFCR in offset ?ISTI) and sets the file type to ?FIPC (the file type for IPC files).

2. AOS/VS issues a global ?IREC system call for the IPC entry, which indicates that the entry is open. (Note that global ?IREC system calls issued from a particular ring can receive only IPCs destined for that particular ring.)

3. The other process issues a complementary ?OPEN system call on the IPC entry.

4. AOS/VS responds with an ?ISEND system call to synchronize the two processes.

After AOS/VS performs these steps, either a process can issue ?READ or ?WRITE system calls through the established IPC file. When one of the processes closes the IPC entry or terminates, the system sends the other process an end-of-file condition (error code EREOF) when it tries another ?READ system call against that file.

You perform I/O on an IPC entry, AOS/VS synchronizes all ?READ and ?WRITE system calls. Thus, for a process to receive another

process's termination message, it must read it in the proper sequence. Otherwise, the process could repeatedly attempt to write to the closed IPC entry with no results, because in that case, there is no error return.

Note that the process that creates the IPC file (by issuing the firs ?OPEN system call) owns the file.

Task Concepts

A task is a path through a process. It is an asynchronously controllable entity to which the CPU is allocated for a specific time. A task can only execute code within the bounds of the address space allocated to its process.

Each process consists of one or more tasks, which execute asynchronously. You can design your code so that several tasks execute a single re-entrant sequence of instructions, or you can create a distinct instruction path for each task.

You combine program file with other information to define processes. A task is the basic element of a process. Initially. each process has only one task associated with it. However. unlike processes, tasks within a process only exist until you kill them either explicitly or implicitly.

If you are familiar with high-level languages such as BASIC or FORTRAN, you are probably familiar with single-task programs. Single-task programs display one path that connects all branches of logic, no matter how complex. Multitasking is a programming technique that allows up to 32 tasks within a single process to execute.

As a programming technique, multitasking offers several advantages, including:

- Parallelism

  Multitasking is a straightforward way to handle complex parallel events within one program. Thus, it can be useful for time-out and alarm routines, and overlapped I/O. Multitasking gives a program the flexibility to respond to external asynchronous events

- Efficiency

  While one task is suspended, perhaps on an I/O operation, another task can be executing. Each task has a priority level, and AOS/VS schedules tasks based on their relative priorities. The AOS/VS multitasking scheduling facility provides efficient CPU and memory use, especially in an environment with heavy memory contention and devices of varying speeds.

You can design your code so that several tasks execute one re-entrant instruction sequence, or you can create a different instruction path for each task.

Task Protection Schemes

The AOS/VS protection model prevents tasks executing in an outer ring from interfacing with tasks executing in critical inner-ring code paths. AOS/VS uses two classes of protection mechanisms to protect tasks executing in one ring from interference by tasks executing in other rings:

- Ring maximization

  Under this protection scheme, AOS/VS considers a task that is executing in a user ring to be less privileged than another task that is executing in a lower user ring. For all system calls, AOS/VS uses the ring-maximization protection scheme when it validates user-supplied channels, word pointers, or byte pointers.

  This means that a channel opened by a system call issued from one user ring cannot be passed as input to a system call issued from a higher user ring. Also, system calls issued from one user ring cannot be passed as input pointers to lower-ring memory locations.

  The ring-maximization protection scheme parallels the hierarchical protection scheme of the MV-series memory-management hardware.

- Ring specification

  The ring-specification protection scheme protects tasks executing in one user ring from interference by tasks executing in any other user rings. The connection-management facility and the IPC facility use the ring-specification protections scheme in the following ways:

  - The connection-management facility considers connections to be between pairs of process identifier (PID)/ring tandems.

  - The IPC facility now requires a ring field as well as a PID and a local port number field as part of each global port.

    All IPC messages are sent to specific rings within a destination process. Within the destination process, only tasks that issue IPC receive request system calls from the specified ring can receive IPC messages sent to that ring. In this way, interprocess communications paths are secured from both malicious and accidental interference by tasks issuing IPC requests from other rings within the same process.

Task Identifiers and Priority Numbers

When you create a task, you should assign it a task identifier (TID) in the range from 1 through 32. In addition to providing a simple way for you to keep track of each task's actions, several system calls require a TID as input.

If you do not assign each task a TID, AOS/VS assigns the initial task TID 1, but assumes that every other task is TID 0. Although permissible, this is not advisible. Tasks that share TID 0 cannot issue ?IDSTST, ?IDPRI, ?IDRDY, ?IDSUS. and ?TIDSTAT system calls.

In addition to the TIDs that you supply. AOS/VS assigns a unique TID to each task in the system. Therefore. even though each initial task is TID 1 within its own process, it also has a unique TID. This system-assigned unique TID allows you to index into multiple-task databases.

To fine out what the unique TID for a particular task is, issue the ?UIDSTAT system call. The ?UIDSTAT system call returns the unique TID and the contents of the task's status word.

Priority numbers are values AOS/VS uses to determine the order in which tasks execute. Priority numbers range from 0 (the highest priority) through 255 (the lowest priority). AOS/VS assigns the initial task (TID 1) priority 0, highest priority.

To find out the priority and TID of a calling task, issue the ?MYTID system call. If you want to use system calls that require a TID or priority level as an input parameter, you can use the ?MYTID system call to get this information.

Task Identification

The Link utility lets you specify the maximum number of tasks in a process, up to a limit of 32 tasks. Each process is initialized when AOS/VS begins to execute that process's initial task. To initiate other tasks, any executing task can issue the ?TASK system call.

The ?TASK system call requires a packet. This packet allows you to specify several characteristics for the new task, including its TID and its priority.

You can influence task scheduling by assigning a priority level to a task. If you do not assign a priority, aos/vs assigns the new task the same priority level as the calling task (the task that issued the ?TASK system call).

You can use the ?TASK system call to initiate one or more tasks immediately, or you can use it to initiate a task at a later time. Therefore, there are two versions of the ?TASK packet:

o The standard packet. which initiates a task.

- The extended packet. which initiates a task at a particular time and at particular intervals. This is called queued task creation.

When you isssue a ?TASK system call that specifies a starting PC within Ring7, AOS/VS passes control to the ?UTSK task-initiation routine. which places the address of a task-kill routine in AC3 and then returns control to the ?TASK system call. (?UTSK is in the user runtime library URT32.LB.)

You can tailor a task-initiation routine to your own application. For example, you may want to assign system resources to each newly initiated task. To use a tailored task-initiation routine, you must assign the new routine the label ?UTSK and then link it with your program. If you do not do this, AOS/VS passes control to the default ?UTSK routine, which immediately returns control to the ?TASK system call. In addition, if your tailored ?UTSK task-initiation routine pushes anything onto the stack, it must also pop it off the stack before exiting from the routine. Otherwise, if it leaves anything on the stack, the calling task may not return to the proper address in your program.

To abort the ?TASK system call while your ?UTSK task-initiation routine is executing, load AC0 with an error code and return to the address in AC3 (the address of the task-initiation error return). If you do not want to abort the ?TASK system call, increment the value of AC3 by 1 and return to the address in AC3 (the address of the task initiation normal return). This not only causes the ?UTSK task-initiation routine to return successfully, but also causes the ?TASK system call to continue normally.

To use the queued task creation option, you must set the extended ?TASK packet. and you must issue the ?IQTSK system call creates an additional task, the queued task manager. which handles the initiation queue. (The queued task manager is one of the 32 possible tasks in your program.) THE ?DQTSK system call removes one or more ?TASK packets from the queued task manager's initiation queue.

Stack Space Allocation and Stack Definition

Every task that uses the AOS/VS system calls must have a unique stack. A stack is a block of consecutive memory locations that AOS/VS sets aside for task-specific information.

The stack works by a push-down/pop-up mechanism; that is, you store information by "pushing" it onto the stack, and retrieve information by "popping" it off the stack. The 'Principles of Operation 32-bit ECLIPSE systems' manual explains stacks in detail and describes the assembly language instructions for the push and pop functions.

The Link utility allocates the stack for the initial task when you link your program. By default, Link sets up a stack of 60 words for the initial task. You can specify an alternate size by using the appropriate function switch in the Link command line.

You must allocate stack space and define the stacks for all other tasks within the ?TASK packet(s). The stack parameters in the ?TASK packet include the stack base, or starting address of the stack, the stack size, and the address of the stack fault handler.

The stack fault handler is a routine that takes control when there is a stack fault. you can define your own stack fault handler or you can use the AOS/VS default stack fault handler. To specifiy the default stack handler, set the stack fault handler parameter to -1.

A stack base value of -1 means that you will allocate the stack at a later time (that is, after task initiation). If ycu choose this option, you must allocate the stack before the newly initiated task issues any system calls. You must allocate a stack at least 30 double words (60 words).

Inner-Ring Stacks

A task that tries to enter an inner ring via an LCALL instruction cannot succeed unless there is a 32-bit stack (called a wide stack) already defined in the target ring for that task. When you load a segment image into an inner ring, inner-ring stacks must be initialized for all tasks that may want to enter that ring. This section describes the rules that govern the inner-ring stack initialization that AOS/VS performs when you issue the ?RINGLD system call.

Every process begins executing in Ring 7. You can specify the Ring 7 stack for the initial task of the process either when you link or after the initial task begins to execute.

The ?RINGLD system call initializes inner-ring wide stacks on behalf of all possible tasks in an inner ring. You can specify the size of these initial stacks at one of the following times:

o When you compile your program.

  To do this, the compiler initializes locations 20 through 27 (the wide-stack parameters) of the process image. Then, at ?RINGLD time, AOS/VS partitions the region delimited by the stack base and the stack limit into separate stacks of equal size for all of the tasks in the process.

o When you link your program into a program file.

  To do This, you must specify the following in your Link command line:

/STACK=n

where n = (number of tasks) * (stack size per task)

Link allocates n words at the end of your unshared area. At ?RINGLD time, AOS/VS partitions this n-word region into separate stacks for each task in the process. Although n can be as few as 12 double words, we recommend that you allocate at least 60 double words per task for n.

If you specify the segment image's initial stack size when you link, AOS/VS uses that size to override any stack size that you may have specified at compile time.

When you link an inner-ring segment iage, you should also specifiy a value for the/TASKS= switch. The number that you choose must be greater than or equal to the number of possible tasks specified for the (Ring 7) process image. (Note that a general-purpose local server should be linked for 32 tasks.)

When you issue ?RINGLD, AOS/VS performs the following steps:

1. AOS/VS loads the segment image into the inner ring for which it was linked.

2. AOS/VS initializes wide stacks in the specified ring for all tasks of the process.

   AOS/VS gets the size of the total available stack region from locations 20 through 27 (the wide-stack parameters) of the ring. Then, AOS/VS divides tne region into equal-sized wide stacks for each possible task in the process. The size of each stack is the size that was implicitly set at either compile or Link time.

Typically, AOS/VS performs the following steps to initialize the inner-ring stacks:

1. AOS/VS sets the frame pointer, the stack pointer, and the stack base to the start of the task's stack region.

2. AOS/VS sets the stack limit to the end of the stack region minus 2 frames.

3. AOS/VS sets the stack overflow handler address to the address that you specified in page 0 of the segment image.

It is possible to force AOS/VS to initialize a single common inner-ring stack for all tasks in the process. To do this, set the stack pointer within the segment image so that it contains the same value as the stack limit. Then, at ?RINGLD time, AOS/VS initializes all the stacks within the inner ring so that they have the same stack pointer, frame pointer, stack base, and stack limit.

?TASK system calls can be issued from any loaded user ring. If a task in an inner ring issues a ?TASK system call, it can initiate a task in that ring or in any higher, loaded user ring. It can specify new wide-stack parameters for the new task. Offsets ?DSTB, ?DSFLT, and ?DSSZ of the ?TASK packet allow the caller to initialize new stack parameters for the task in the ring specified by the new task's initial PC (offset ?DPC).

The ?TASK system call causes AOS/VS to reset the wide stacks for the new task in all user rings lower than the ring specified in ?DPC. AOS/VS resets wide stacks by resetting thestack pointer and the frame pointer to the stack base. This ensures that the tasks can re-use the same stack sequentially several times in a ?TASK/?KILL/?TASK/?KILL sequence.

Once a new task has been initiated, it is free to allocate a new wide stack for itself at any time. However. it is the responsibility of the task to recycle the old wide-stack memory, if the process wishes to re-use the memory.

Task Scheduling

AOS/VS schedules tasks according to a strict priority scheduling algorithm applied at the task level.

After a process's initial task begins to execute under AOS/VS, you can change its task priority at any time by issuing either the ?PRI or the ?IDPRI system call.

To change a process's own priority, you can issue the PRIPR system call. However. if you want to change the priority of another process, the calling pocess must be in Superprocess mode.

Tasks pass through several different states while a process is executing. a task passes from inactive to the active state when you initiate it with the ?TASK system call. After a task is active, it can become ready or suspended. Figure 5-7 illustrates the task states and the system calls that affect them.

AOS/VS reschedules tasks under the following circumstances:

- When the task that is executing becomes suspended.
- When a suspended task of a higher priority than the task that is currently executing becomes ready to run.
- When there is more than one highest prioroty-level task that is ready to run and a round-robin interval has elapsed.

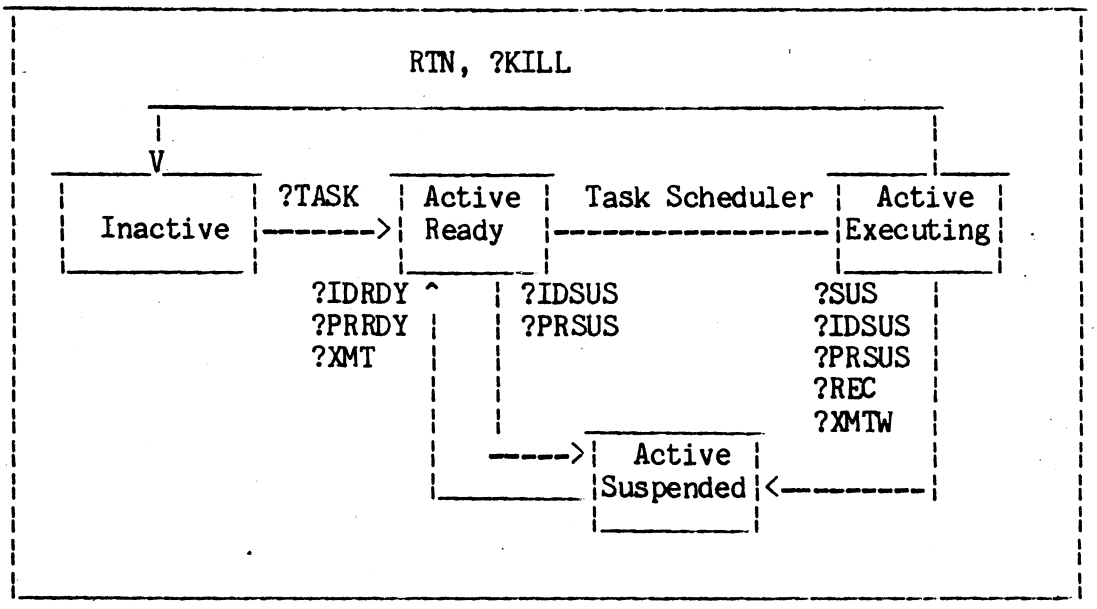


Figure 5-7. Task States

To disable scheduling, you can issue either the ?DRSCH system call, which does not return an indication of the prior state of scheduling, or you can issue ?DFRSCH system call. which does. Both the ?DRSCH and the ?DFRSCH system calls are very dangerous in that they can disrupt the entire multitasking environment. Therefore, do not use these system calls unless you are very certain that they are precisely what you need.

To re-enable scheduling after you have disabled it with a ?DRSCH or a ?DFRSCH system call. issue ?ERSCH.

Task Suspension

Several different events, including some system calls, will suspend an active task. To explicitly suspend a task, issue one of the following system calls: ?SUS. ?IDSUS. or ?PRSUS. Certain other system calls suspend the calling task while they perform their functions. Sysytem calls of this kind include the I/O system calls ?READ and ?WRITE. system calls to acquire system resources, and system calls that depend on another task's response, such as the ?XMTW and ?REC system calls.

Tasks compete for all system resources (including the CPU),. Only "ready" tasks can compete for the CPU. A task is ready if it is not waiting for some event to complete (that is suspend). If a task is not ready. then it is suspended.

A task becomes suspended when it:

- Is part of a proces that the ?BLKPR system call has blocked. To do this, the ?BLKPR system call suspends all tasks within the process.
- Issues an explicit request to suspend itself or another task within the same process (via the ?IDSUS and ?SUS system calls).
- Issues an explicit request to wait for a message from another task within the same process (via ?REC and ?XMTW system calls).
- Issues certain (most) system calls. A system call is usually a request to use some system resource.

If every task in a process is suspended, then the process is blocked. To block a process (that is, suspend every task), you must issue the ?BLKPR system call. When you have explicitly blocked a process with the ?BLKPR system call. you must issue he ?UBLPR system call to unblock that process.

The ?WDELAY system call suspends a task for a specific amount of time. This allows you to synchronize tasks or to temporarily suspend a task until some asynchronous event has completed.

Task Readying

Atask remains suspended until the event that caused the suspension completes or until the suspended task is "readied" by AOS/VS or by another task.

Tasks become ready when:

- A task that was suspended by a ?blkpr system call against its process is explicitly unblocked by the ?UBLPR system call and the task is not suspended for any other reason.

  The ?BLKPR and ?UBLPR system calls work together. Therefore, the ?UBLPR system call can only unblock processes that were blocked by the ?BLKPR system call.

- A task issues an ?IDRDY or a ?PRRDY system call to explicitly request that AOS/VS ready another task. (The ?IDRDY system call readies a task of a given TID and the ?PRRDY system call readies all tasks of a given priority).

  In this case, the task that is being readied must have been previously suspended by a ?SUS, ?IDSUS, or ?PRSUS system call. In addition, the task that is being readied must belong to the same process as the task that issues the ?IDRDY system call.

- A message for which the task was explicitly requested to wait

becomes available. In this case, the tsk only becomes ready when the message is from another tsk within the same process.

- o A system resource becomes available after an implicit wait for that system resource during a system call.

- o A task issues a task-kill system call (?IDKIL or ?PRKIL) or a redirection system call (?IDGOTO) against a suspended task. (Before AOS/VS executes the system call, it automatically readies the target tsk.)

Task Redirection

To redirect a task's activity without killing it, you must issue the ?IDGOTO system call. The ?IDGOTO system call stops the task's current activity (or readies the task, if the task was suspended) and then directs the task to a new location. The task begins executing at the new location as soon as it regains control of the CPU. The task's priority remains the same.

Typically. you use ?IDGOTO to interrupt a task after a CTRL-C CTRL-A console interrupt sequence. (A CTRL-C CTRL_A sequence interrupts console output.)

Inner-ring Task Redirection Protection

Tasks executing in critical sections of an inner ring cannot tolerate being redirected by tasks executing in outer rings. However, task redirection is a common method of responding to external events. In fact, typing a CTRL-C CTRL-A console interrupt sequence frequently causes an ?IDGOTO system call to perform task redirection on the main task(s) of a process. Therefore. to solve this problem, AOS/VS provides you with the ?TLOCK and ?TUNLOCK system calls, which allow you to control whether a task can be redirected by a task-redirection system call. (The task redirection system calls are ?IDGOTO. ?IDKIL, ?PRKIL, ?IDSUS, and ?PRSUS.)

The ?TLOCK system call allows a task that is executing in an inner ring to lock itself against task-redirection system calls issued by another task that is executing in a higher ring of the same process. The ?TUNLOCK system call unlocks a previously locked task.

A task can issue a ?TLOCK system call to protect itself from being redirected by ay task that is in a higher ring or, optionally. in the same ring. The ring-maximization protection scheme governs which tasks can and cannot redirect a task. (In other words, only a task-redirected system call that originates from the same ring or in a lower ring can redirect a locked task.)

If a task issues a task-redirection system call, but the task it wants to redirect (the target task) is locked, the calling task waits until the target task issues enough ?TUNLOCK system calls to unlock

the rings that are lower than the ring in which the calling task resides.

If a task issues a ?PRKIL or a ?PRSUS system call whose input priority specifies more than one protected task, AOS/VS makes a note of all tasks of that priority when the ?PRKIL or ?PRSUS system call occurred. If the redirecting task must wait because one or more target tasks are locked, the task will only wait until all the noted locked tasks issue enough ?TUNLOCK system calls to allow the redirection to occur. If a redirecting task specifies more than one task, the redirection may occur separately (depending on whether one or more of the target tasks are locked). However, in this case, the task-redirection system call will not complete until all the specified tasks have been redirected.

As input to the ?TLOCK system call, you can specify a double word mailbox in AC2, if you want AOS/VS to inform your protected task when another task is trying to redirect it. AOS/VS will set a nonzero flag in this mailbox if another task's redirection request is waiting.

To protect a task from being redirected by another task within the same ring, set the ?TMYRING flag in AC0 when you issue the ?TLOCK system call.

If a task in an inner ring is redirected to a higher ring, then AOS/VS resets the stack pointer and frame pointer for each affected inner ring to the stack base of that ring on behalf of all loaded user rings that are less than or equal to the redirected higher ring. This means that if a task in Ring 5 is redirected to Ring 7, AOS/VS resets the task's stack and frame pointers for Rings 5 and 6.

Task Termination

You can kill (terminate) a task explicitly or implicitly. To explicitly kill a task, issue one of the following system calls:

?IDKIL	Kills a task of a certain TID.
?PRKIL	Kills all tasks of a certain priority.
?KILL	Kills the calling task.

To kill a task implicitly, begin the new task with a WSSVS or WSSVR (wide-save) instructions, and end it with a WRTN (wide-return) instruction. As AOS/VS executes the initial wide-save instruction, it saves the contents of AC3 as the return address for the task. At this point, AC3 contains the address of the task-kill routine (placed in AC3 during task initiation). When AOS/VS executes the WRTN, it passes control to the return address in AC3; that is, the task-kill routine

Because killing a task does not guarantee an orderly release of its user-related resources, you may want to define a kill-processing routine for this purpose (for example, to close the task's currently

open channels).

You can define either a unique kill-processing routine for each task or a general kill-processing routine for all tasks within a process. ?KILAD, which you issue after task initiation, defines a unique kill-processing routinethat is then invoked when you isssue ?IDKIL or ?PRKILL. If you define a general kill-processing routine, assign the routine the label ?UKIL and link it with your program. You can use both ?KILAD and user-defined ?UKIL kill-processing routines within the same program.

If there is no user-defined ?UKIL routine to kill a task, AOS/VS uses the dummy ?UKIL routine in URT32.lb. This routine returns control to AOS/VS, which then kills the task. ?UKIL kill processing is only invoked on behalf of tasks that initiated processing within Ring 7 (that is, tasks whose initial PCs are Ring 7 addresses).

Task Creation and Termination Detection

Typically, a local server needs to maintain accurate task-specific databases. Therefore, to keep those task-specific databases accurate, a local server must be able to keep track of when tasks are created and when they terminate. This action describes how AOS/VS helps an inner-ring server to detect when a task is created and when it is terminated.

All active tasks have distinct Unique Storage Position (USP) pointers associated with Rings 4 through 6. Tasks within 32-bit processes also have a USP pointer associated with Ring 7. A double-word pointer at location ?USP within a ring specifies the USP pointer for a given task within the ring. The USP pointer allows tasks to keep track of task-specific databases associated with a particular ring.

When a process issues a ?TASK system call to create a task, AOS/VS initializes all the USP pointers associated with that task to zero. When a customer issues LCALL to enter a local server, the local server can examine the USP pointer to that inner ring. The local server can interpret a zero USP pointer to mean that this is the task's first visit to the local server. In this case, the local server can initialize any task-specific databases for that initially entering task.

AOS/VS uiquely identifies every task within a process to aid in identifying task-specific databases with their tasks. The ?UIDSTST system call returns the unique TID associated with a given task.

When a task terminates, AOS/VS serially invokes a ?UKIL posprocessor for each loaded user ring whose ring number is less than or equal to the ring specified by the task's initial PC. Local servers can use the ?UKIL postprocessor to update or deallocate task-specific databases, as appropriate. The ?UKIL routine should not issue system calls.

Several ?UKIL postprocessors (one per ring) can be associated with a process. However, only one ?UTSK postprocessor can be associated with a process. AOS/VS only invokes a ?UTSK postprocessor on behalf of tasks that are to be executed in Ring 7. The ?UTSK postprocessor must reside in Ring 7.

Task-to-Task Communications

AOS/VS provides an intertask communications facility that you can use to synchronize tasks or pass messages among them. The following system calls allow tasks to communicate with one another:

?XMT	Transmits an intertask message
?XMTW	Transmits an intertask message and awaits its reception
?REC	Receives an intertask message; suspends the ?REC caller if there is no message currently available
?RECNW	Receives an intertask message; does not suspend the ?REC caller if there is no message currently available

Tasks deposit messages in and retrieve them from 32-bit locations called mailboxes. Before you can send a message with an ?XMT or ?XMTW system call, you must initilize the appropriate mailbox to zero.

Timing is a factor for both the ?XMTW and the ?REC system call. If a sending task issues an ?XMTW system call before another task issues a complementary receive, AOS/VS suspends the sender until the receive occurs. Likewise, if a task issues a ?REC system call against an empty mailbox (the sender has not transmitted the message yet), AOS/VS suspends the receiver until the transmission occurs.

THe ?XMT and ?RECNW system calls maintain the calling task in the ready state, regardless of the timing of the transmit and receive sequence. If a task issues a ?RECNW system call against an empty mailbox, the system call fails, and AOS/VS returns an error code to ACO.

You can use the ?XMT and ?XMTW system calls to "broadcast" a message; that is, to send the message to all tasks currently waiting for the message. If you do not select the broadcast option and more than one task is waiting for the message, AOS/VS sends the message to the receiver with the highest priority.

Critical Region Locking

You can use the intertask communications system calls to lock or unlock a critical region. A critical region is a procedure or database that all tasks share, but that is available to only one task at a time. To protect a critical region, you must define a mailbox to synchronize task execution within the critical region. A task gains control of a critical region by issuing a successful receive against that mailbox. The procedure for locking and unlocking a

critical region is as follows:

- First, a task initializes the locking facility, by either setting the mailbox to a nonzero value or by issuing the ?XMT system call "without broadcast" from the initializing task to the mailbox. (The ?XMT system call message may specify the address of the critical region.)

- Second, a task locks (gains exclusive control of) the critical region by issuing an ?REC system call against the mailbox. AOS/VS suspends other tasks that issue subsequent ?REC system calls against the mailbox.

Once a task has locked a critical region, it remains locked until the task issues another ?XMT system call to unlock it. If more than one task is waiting for control of a critical region (that is, more than one task was suspended by a ?REC system call to the mailbox), the second ?XMT system call readies the highest priority receiver, which then gains control of the critical region.

You can also lock a critical region implicitly by issuing a ?DRSCH system call. which disables all task scheduling in the calling process, or a ?DFRSCH system call. which not only disables all task scheduling in the calling process, but also returns indication of the prior state of scheduling. If you use a ?DRSCH or a ?DFRSCH system call to lock a critical region, you should use a ?ERSCH system call to unlock it. However. ?DRSCH and ?DFRSCH system calls can be dangerous because they disable all multitask scheduling for the calling process.

Unless absolutely necessary, you should avoid the ?DRSCH and ?DFRSCH system calls. Although there are may be times when you need to issue one of these system calls, such as to control a "race" condition between two tasks that are competing for the same critical region, you must use them with discretion. Disabling task scheduling, even briefly. can disrupt the entire multitasking environment.

The ?ERSCH system call re-enables multitask scheduling for the calling process.

The Interprocess Communications (IPC) Facility

AOS/VS allows processes to communicate with each other through the Interprocess Communications (IPC) facility, which allows you to:

- Transmit variable-length free-form messages from one process to another
- Synchronize processes during execution.

You can use the IPC facility to pass arguments from a father process to a son process and return the results to the father before the son terminates. If there is a delay between the father's receive request terminates. If there is a delay between the father's receive request and the son's message, AOS/VS pends the father process until the son process responds, thereby synchronizing the two processes. AOS/VS uses the IPC facility to send messages to father processes to notify them of their sons' termination.

The following primitive system calls allow you to send and/or receive IPC messages:

?ISEND	Sends an IPC message
?IREC	Receives an IPC message
?IS.R	Sends and then receives an IPC message

For each of these system calls, you must supply a header (packet) that includes the origin and destination of the message, its length, its address, and other information about the connection.

During each IPC transmission, portions of the sender's header overwrite portions of the receiver's header. In fact, some transmissions consist solely of passing header information from the sender to the receiver.

To use the primitive IPC system calls, ?ISEND and ?IS.R, the calling process must have privilege ?PVIP, which is one of the optional privileges you can specify when you create a process with the ?PROC system call.

If the calling process does not have ?PVIP privilege, it must use the IPC facility as a standard peripheral device, which it can then access by device-indepenent I/O techniques. Also, you can use the connection-management facility, to establish communications between processes. (Note that is a process is a declared customer under the connection-management facility, it does not need the ?PVIP privilege to issue the ?IS.R system call.)

Sending Messages Between IPC Ports

AOS/VS send IPC messages between ports. Ports are full-duplex communications paths that a process identifies by port numbers. There are two types of port numbers:

- Local port numbers

  Local port numbers are values that the IPC caller (either the sender or receiver) defines to identify its own ports.

- Global port numbers

  Global port numbers uniquely identify each port currently in use system wide. Global port numbers are made up of a process's PID, its local port number.and its ring number. When a process refers to its local port in an IPC system call, AOS/VS translates the local port number to its global equivalent. The ?TPORT system call performs this translation.

When a process sends an IPC message, it defines a local port number for the connection, then it specifies that port number and the destination's global port number in the IPC header. The receiving process issues a complementary receive system call and, like the sender, defines its own local port number and specifies the sender's global port number. If the port specifications on both ends match (including target ring), AOS/VS sends the message.

a process must use a global port number to refer to another process's port. However, because global port numbers depend on the system environment, they frequently change during subsequent process execution. To circumvent this problem, potential IPC users can issue ?CREATE system call to create IPC files, which serve as ports. Then, these same users can define the local port numbers before they issue IPC system calls. As AOS/VS executes the ?CREATE system call, it translates the local port numbers into glob11 port numbers. Potential senders and receivers can then issue ?ILKUP system calls against the IPC file to determine its global port number.

When you issue the ?CREATE system call to create an IPC file, AOS/VS saves the number of the ring from which the system call was issued in the new IPC file. The global port number, which ?ILKUP returns, incorporates this same ring number. AOS/VS interprets all global port numbers as containing ring fields.

The ?ISEND and ?IS.R system calls interpret ring fields (within global port numbers) as follows:

> Offset ?IDPH(the global port number) must always contain a valid user ring number. The ring number specifies the ring to which the message will be sent. However, the caller must have appropriate privileges to send a message to that ring within that particular process

The ?IREC system call interprets ring fields (within global port numbers) as follows:

> Offset ?IOPH (the global port number) can contain either a valid user ring number or a zero ring number. A nonzero ring number indicates that ?IREC returns a message only from sends issued from the specified origin ring within the specified origin process. A zero ring number indicates that ?IREC will return a message from any ring within the specified origin process that sends a message destined for the ?IREC caller's ring. (You can use the ?IMERGE system call to construct a global port number with a zero ring field.)

When you include ring fields as part of global port numbers, the ?IREC port-matching rules are affected in that if the receiver specifies a non-zero ring field in an otherwise zero global header, a ring-specific global receive takes precedence after explicit matches.

To identify the PID that is associated with a particular global port number, you must issue the ?GPORT system call. Conversely, if you know the name of the PID of a console's associated process, you can identify its console port-number by issuing the ?GCPN system call.

The ?ISPLI system call extracts the ring field from a global port number. while the ?IMERGE system call permits bothe16- and 32-bit users to modify the ring field within a global port number.

Connection Management

AOS/VS allows you to establish a customer/server relationship (called a connection) between processes, and then use the server process to perform certain functions on behalf or its customers. Typically, a server process performs general routines that customer processes can access. For instance, you can create a server process to build files or perform I/O.

Connection management allows servers to move bytes to and from their customers' buffers.

Connection Creation

To make a connection between two processes you must define one process as the server and the other as the customer. To do this, issue the ?SERVE system call to define the calling process as a server. and issue the ?CON system call to define a customer and establish the logical connection between the customer and an existing server.

AOS/VS maintains a connection table, which manages exchanges between customers and servers. When a customer makes a connection (via the ?CON system call) with a declared server, AOS/VS writes an entry in the connection table that specifies the PID of the server, the PID of the customer. and the customer's ring field. Each ?CON system call generates one connection-table entry.

A process can act as a server for other processes and can also act as a customer of other servers as long as it issues the appropriate number of ?SERVE and ?CON system calls. A process that acts as both a server and a customer is called a multilevel connection.

You can make a double connection between two processes. A double connection allows each process to act as ether the customer or the server of the other. depending on the action to be performed. A double connection requires that two ?SERVE system calls and two ?CON system calls. AOS/VS creates two connection-table entries, one for each ?CON system call.

Fast Interprocess Synchronization

Frequently identical local servers loaded into different processes will use a common shared memory file for global synchronization. AOS/VS includes a fast interprocess synchronization facility that common local servers can use to pend and unpend tasks, depending on the state of data bases in that shared memory.

The fast interprocess synchronization mechanism, which uses the ?SIGNL, ?WTSIG, and ?SIGWT system calls, provides you with another way of synchronizing processes. Unlike the IPC system calls, the fast interprocess synchronization system calls do not move any data. Instead, they allow a task within a process to send and receive

task-specific signals to and from the same or another process. Because they do not move data. ?SIGNL, ?WTSIG, and ?SIGWT are very fast, and they require very little system overhead.

When a task issues a ?SIGNL or a ?SIGWT sytem call, the target does not have to be waiting to receive the signal. Instead, AOS/VS remembers the task-specific target. A subsequent ?WTSIG or ?SIGWT system call issued by the target task causes the target task to proceed immediately. A ?WTSIG system call, however, will pend the caller if a signal for the task is not outstanding.

Unlike the IPC system call ?IS.R, the ?SIGWT system call does not force the calling task to wait for a signal from the same task that it signaled. Any signal that specifies the pended task will wake up that task.

No privileges are necessary to issue the ?SIGNL, ?WTSIG, or ?SIGWT system calls.

User Device Support

AOS/VS supports a wide variety of user devices, such as magnetic tape drives, disk drives, and line printers, which you usually define during the system-generation procedure. However a process that has the ?PVDV privilege can define and enable devices at execution time.

Devices that you define and enable during the system-generation procedure are called system-defined devices. Devices that a process with the ?PVDV privilege defines and enables at execution time are called user-defined devices. This section describes those system calls that allow you to use both system- and user-defined devices.

AOS/VS supports a maximum of 64 user (that is, system-defined and/or user-defined) devices per I/O system. You can use any device code in the range from 1 to 63, as long as you do not use codes that are already in use. (Note that AOS/VS reserves many device codes for its own use.)

To introduce a user-defined device to AOS/VS at execution time, you must issue the ?IDEF system call. As input to the ?IDEF system call, you must supply:

- A device code for the new device.
- The address of the device control table (DCT) you defined for the new device.

The DCT specifies the address of the user-defined device's interrupt service routine. You must supply to AOS/VS the following information:

1. The address of the interrupt service routine
2. The address of the power-failure/auto-restart routine or, if you do not want to use such a routine, -1.
3. the interrupt service mask.

To remove a user device, issue the ?IRMV system call.

?IDEF System Call Options

When you issue the ?IDEF system call. you can select any of the following options:

- o Burst Multiplexor (BMC) I/O
- o Data Channel (DCH) map A.
- o DCH map B, C, or D.
- o Neither BMC or DCH I/O.

You can select either burst multiplexor (BMC) I?O or data channel (DCH) I/O for a user-defined device by selecting certain options when you issue the ?IDEF system call. (In general your choice depends on the device's design.)

If you want to use the BMC map or DCH B, C, or D maps, you must use an extended map table. However, you can define (issue the ?IDEF system call against) a device that is on DCH map A without using an extended map table. To do this, you must specify that you do not want to use the extended map table in the accumulators when you issue the ?IDEF system call. This option, does not allow you to specify the first acceptable map slots your application needs.

However, if you do not want to use either DCH or BMC I/O, you must specify this option in the accumulators when you issue ?IDEF. Also, if you do not want to use either DCH or BMC I/O, you do not have to use the extended map table.

Burst multiplexor I/O requires program control only at the beginning of each block transfer. Therefore, BMC I/O is generally faster than DCH I/O. Typically, the BMC rate is about half the memory rate, although the exact transfer rate varies from implementation to implementation. Note that not all user-deffined devices have BMC hardware.

If you use extended form of ?IDEF, you can select specific DCH or BMC map slots. Each MV/8000 DCH map consists of 32 map slots, numbered 0 through 37 (octal). The MV/8000 BMC map consists of 1024 map slots, numbered 0 through 1777 (octal).

Each map slot (DCH and BMC) address 1K memory words. The hardware uses three map slots to map data from the device to memory during data transfers.

To select a particular DCH map or the BMC option, you must perform the following steps:

1. Set up a map definition table in your logical address space.

2. Issue the ?IDEF system call.

When you issue the ?IDEF system call, AOS/VS allocates--but does not initialize--map slots. To initialize these map slots, you must issue the ?STMAP system call.

If you issued the ?IDEF system call with the DCH map-A-only option, then you can issue only one ?STMAP system call to initialize the map slots allocated on DCH map A. However, if you issued the ?IDEF system call with the extended-map-table option, you can issue more than one ?STMAP system call. Each ?STMAP system call, in turn, initializes a different group of map slots. (The map definition table entries define each group of map slots.)

When you issue the ?STMAP system call, you can initialize part of a group of map slots that is defined in a map definition table entry. For example, if entry 2 has allocated 10 map slots on the BMC, an ?STMAP system call only initializes 5 of the 10 map slots.

(For a detailed description of BMC I/O, DCH I/O, and the DCH maps, refer to the 'Principles of Operation 32-Bit Eclipse systems' manual.)

User Interrupt Service

To define a user device to AOS/VS, you must issue the ?IDEF system call. Each device, such as a disk, is programmed to do a particular job. When a device starts doing its job, the CPU and AOS/VS ignore that device. As soon as the device completes its job, it signals the CPU that it is done. This signal is called an interrupt.

When the CPU detects an interrupt, it stops doing whatever it is doing, so that it can "service" the interrupt. Servicing an interrupt means that AOS/VS passes control to the appropriate interrupt service routine. To do this, AOS/VS must pass a vector through the interrupt vector table, which is a hardware-defined index.

The interrupt vector table contains an entry for each device. Each entry points to a DCT, which contains the address of the interrupt service routine that will service a particular interrupt.

The ?IDEF system call directs AOS/VS to build a system DCT and enter it in the interrupt vector table. Conversely, the ?IRMV system call clears the device's DCT entry from the interrupt vector table.

The device's DCT also contains the current interrupt service mask. The current interrupt service mask is a value that specifies the devices that can interrupt the user-defined device

Before AOS/VS transfers control to an interrupt service routine, it performs the following steps:

1. Loads AC2 with the address of the interrupting device's DCT.

2. Loads AC0 with the current interrupt service mask.

3. Takes the current interrupt service mask and incluseively ORs it with the interrupt service mask in the DCT.

4. Saves the current load effective address (LEF) state. LEF mode is a CPU state that allows AOS/VS to correctly interpret MV/8000 LEF instructions.

5. Turns off LEF mode.

The inclusive-OR operation establishes which devices, if any, can interrupt the interrupt service routine that is currently executing.

AOS/VS restores LEF mode when you issue ?IXIT system call to dismiss your interrupt.

A process in an interrupt service routine can issue only three system calls:

- ?IXMIT. which sends a message to a task outside the interrupt service routine.
- ?SIGNL, which signals a task within your own or another process.
- ?IXIT. which exits from an interrupt service routine.

AOS/VS does not save the state of the MV/8000 floating-point registers when a process enters the interrupt service routine. If necessary (for example, if you want to use floating-point instructions), you can save the state of the floating-point registers when the interrupt service routine receives control and restore that state before you issue the ?IXIT system call.

User Stacks

Each user task has its own user stack. A user stack is a data structure to which the contents of certain Page 0 hardware locations point. The contents of these hardware locations are called stack pointers. Whenever a task runs, AOS/VS must first load that task's stack pointers into hardware locations octal 20 through 26. This allows the task to use its stack.

When user-defined device interrupt handler receives control at interrupt level, the stack that AOS/VS loads into the Ring 7 hardware registers is the stack of the last user task that was running. AOS/VS does not set up a stack for your interrupt service routine. Therefore, to use a stack, you must set uup your own.

Before you issue the ?IXIT system call to exit from the interrupt service routine. you must perform the following steps:

1. Save the current hardware stack pointers.
2. Restore the current hardware stack pointers to the hardware.

Communicating from an Interrupt Service Routine

Multitasking halts when a device interrupt occurs. However, an interrupt service routine can communicate with an outside task by issuing the ?IXMT system call. The ?IXMIT system call transmits a message of up to 32 bits from the interrupt routine to a specific receiving task outside the sending routine. There is a location in the system DCT that serves as a mailbox for the message. The external task receives the message by issuing a ?IMSG system call against the DCT associated with the interrupt routine.

You can issue ?IXMT and ?IMSG system calls in any order. If the ?IMSG system call occurs before the ?IXMT system call, AOS/VS suspends the receiving task until the ?IXMT system call occurs. If the ?IXMT system call occurs first, AOS/VS posts the message in the mailbox until the receiving task issues the ?IMSG system call.

You cannot use the ?IXMT system call to broadcast a message.

Enabling and Disabling Access to all Devices

Processes can issue I/O instructions from their tasks to all system and user devices. When a process issues a ?DEBL system call, AOS/VS enables device I/O and disables LEF mode, which allows tasks within the calling process to issue I/O instructions. Note that the I/o enable and LEF mode states are process wide, and therefore, affect all tasks.

The ?DEBL and ?DDIS system calls work in parallel with the LEF mode system calls ?LEFE, ?LEFD, and ?LEFS.

No device I/O can occur while the CPU is in LEF mode, because LEF instructions and I/O instructions use the same bit patterns. Similiarly, when LEF mode is disabled, AOS/VS executes LEF instructions as if they were I/O instructions. Thus, the deciding factor for executing LEF and I/O instructions is the state of the CPU; that is, whether it is in LEF mode or I/O mode.

Note that the ?DDIS system call, which disables I/O mode, does not automatically re-enable LEF mode. To disable I/O mode, and re-enable LEF mode, you must issue the ?LEFE system call. Also, the ?LEFD system call, which disables LEF mode, does not automatically re-enable I/O mode. To perform these two functions, you must issue the ?DEBL system call.

LEF Mode

LEF mode (load-efective-address mode) is the CPU state that protects the I/O devices from unauthorized access. I/O instructions and LEF instructions use the same bit patterns. AOS/VS decides how to interpret these instructions by checking the LEF mode state and the state of the complimentary I/O mode.

LEF mode and I/O mode are mutually exclusive. When the CPU is in LEF mode, all I/O instructions execute as LEF instructions; therefore, I/O cannot take place in this state. Conversely, the CPU must be in LEF mode to execute LEF instructions properly.

AOS/VS provides the following system calls to check and alter LEF mode:

?LEFD	Disable LEF mode
?LEFE	Enable LEF mode
?LEFS	Returns the current LEF mode status.

Each process begins with LEF mode enable. AOS/VS disable LEF mode when a process enters a user device routine, and restores LEF mode when the process exits from the routine.

Power-Failure/Auto-restart Routine

If you specify an extended DCT within the ?IDEF system call--provided you have the necessary battery backup hardware--AOS/VS will restart your user devices after a power failure. The DCT extension (offset ?UDDRS) points to a power-failure/auto-restart routine. When a power failure occurs, AOS/VS transfers control to the auto-restart routine, with the DCT address in AC2, and the current system mask in AC0.

AOS/VS checks to see if there are any user-defined devices that have associated power-failure/restart routines if auto-restart is enabled.

(During the auto-restart routine, AOS/VS enables interrupts and masks out all devices. This allows AOS/VS to recognize only power-failure interrupts.) AOS/VS transfers control to the auto-restart routine with a system mask of -1, which cannot be changed. The states of both the devices and the data channel map are undetermined after a power failure.

CHAPTER 6 - DISKWORLD
(AOS/VS revision 5.00)

This chapter explores the diskworld of AOS/VS. Its databases are detailed here along with the structure of the physical disk. The modules which make up the file system are also listed.

ACLST - process ACL's
entry points
- DACL.P - set up default ACL
- SACL.P - set ACL
- ISACL.P- search a system file and system ACL
- JSACL.P- search a user file and user ACL
- KSACL.P- search a user file and system ACL
- VSACL.P- search an IDP to file's PIB in system ACL CCB must be locked
- GACL.P - get a files ACL

ACPRV - set up access privleges
entry points
- ESTAC.P - establish access privileges user name in process table
- PESTAC.P- establish access privileges user name in process table
- UESTAC.P- establish access privileges user name in user space
- TMATCH.P- check if a string matches template
- GTACP.P - given user name get ACL privilege for file
- SATR.P - set a file's attributes
- ASCAN.P - verify that ACL's are correct

BUFIO - disk buffer I/O
entry points
- BUFIO - start of module
- NQBHR - enqueue buffer hdr on LCB
- NQBH1 - enqueue on UDB
- NQUCN - enqueue a BH to UNICORN type device
- BWAIT - wait for BH request
- STUNT - disk starter routine
- IODON - request done processing
- PPBSY - post process for system buffer I/O
- PPBMD - post processor for system buffer I/O readies LCB's waiting for buffers
- CTLIN - checks for controller interference

CACHE - use system cache buffers
entry points
- ONLCB - add buffers on FCB to LCB
- SRCLCB- look for buffer on LCB
- OFFLCB- remove buffer from LCB list
- SLCB - search LCB
- CLRLCB- remove buffers from LCB
- DELLCB- delete deallocated blocks from LCBs

```
CLOSE     - file closing support
              entry points
                        ICLOSE.P - internal close
                        RCLOSE.P - close a file and reserve the channel
                        SCLOSE.P - close a shared file
                        GCLOSE.P - plain user close
                        ECLOSE.P - special system close
                        KFCB.P   - kill a user FCB
                        KCCB.P   - kill a system CCB
                        KCCBE.P  - kill a system CCB and exit
CPDCK     - control point directory hierarchy
              entry point
                       CPDCK - check control point hierarchy to se if it
                               will allow a file to grow
CPDUP     - control point directory update
              entry points
                        CPDUP - update control point directory  to
                                reflect the growth of a file
CREATE    - create a file
              entry points
                        CREATE.P - user call to create a file
                        ICREATE.P- internal call to create a file
                        VCREATE.P- internal call to create a volume
                                   entry LDU
                        BCREATE.P- internal call to create a breakfile
                        PCREATE.P- internal call to create a page/swap
                                   file
DE        - delete a file
              entry points
                        DELETE.P - delete a directory entry
                        IDELETE.P- internal directory entry delete
                        UDELETE.P- internal directory entry delete
                        VDELETE.P- delete a volume directory entry
                        CLDEL.P  - delete a directory entry that has
                                   just been created
                        DIPCF.P  - delete process' IPC directory entries
DEBKS     - de-allocate blocks
              entry points
                        DEBKS    - de-allocate 'n' blocks
                        DEBLK    - de-allocate 1 block
DINIT     - disk initialization
              entry points
                        DINIT.P - disk initialization segment 1
DIRST     - directory manipulation
              entry points
                        DIR.P    - change working directory
                        RDIR.P   - release terminating proc's working
                                   directory and default directory
                        CPMAX.P  - set control point directory's max
                                   size
                        FSTAT.P  - get file status
```

DPANQ - moving head disk enqueue
entry points
DPANQ - moveing head disk enqueue routine

DRLSE - disk release
entry points
DRLSE.P - release a disk
IRLSE.P - internal disk release
SDOWN.P - shutdown the system

DSKIO - disk I/O module
entry points
DSKIO - start of module
NQCCB - enqueue CCB
RUNLC - run any ready requests
CWAIT - wait until a CCB request finishes
LCBWU - wake up IOCBs waiting for a buffer
IOWLK - wake up IOCBs waiting on a locked buffer
RIOCB - release an IOCB to free chain
WAIT - wait for an event to occur
IOWU - wake up IOCBs waiting on a bit map
IOWUNU- wake up IOCBs waitng on a bit map, the lock is already free though
IWWU - wake up IOCBs wainting on LCB withdraw lock
PPCUS - post processor for user CCB requests
PPCUB - ?BLOCKIO post processor for user CCB request
PPCBI - post processor for CCB blkin requests
PPCSY - post processor for system CCB requests
PPCSR - post processor for system shared reads

DUMPR - core dump routine
entry points
DUMPR - dump start address
WTAPE - tape write routine

ESD - emergency shutdown procedures
entry points
ABORT - ESD processing jumped to 50-51
ESDQ - ESD shutdown processing
STOP1 - "DRLSE" jumps here on shutdown
DCHON - data channel mapping on all channels
PZERO - running copy of page zero
SIPZLN- length of page zero for sinit
ESDFLT- page fault handler entry
CLR - clear memory

KFSIN - kernel file system interface
entry points
NQLDRQ - enqueue a logical disk request
RECAL - enqueue a recal request
GSTS - enqueue a disk status request
NQCRQ - enqueue a channel request
CH.ACC - return a channel's access privileges
CH.EOF - return a channel's EOF
CH.FTY - return a channel's file type
CH.LDA - return a chanel's logical disk address

OPEN - open file processing
entry points
GOPEN.P - standard user open
SOPEN.P - user shared open
SOPPF1.P- protected shared file open, 1st open
ROPEN.P - AGENT only reserved open
IOPEN.P - internal open
XIOPEN.P- internal exclusive open
EOPEN.P - internal open with user name
XEOPEN.P- internal exclusive open

RESL2 - network resolves
entry points
NRSLV.P - networking resolve continued
REXTN.P - read the extender
RNAME.P - ?RNAME system call
FFCB.P - find an FCB
LOOKUP.P- look up a filename in a directory

RESLV - resolve a filename
entry points
GRSLV.P - ?GNAME resolve
RESLV.P - search for disk files in the pathname
WRSLV.P - search for disk files in path execept last which cannot exist
DRSLV.P - search for files in path last must be a directory
RRSLV.P - ?GFNAME resolve
LKRTN.P - return from link resolution

SGSB3 - file system and system call processing subroutine
entry points
SGSB3 - start of module
DQBCN - dequeue buffer header from LRU
IMTB - (interrupt level) move bytes
FPLOK - pend lock an FCB
FPULK - un-pend lock an FCB
PLOCK - lock cxpblk for locking disconnects while PMTPF's are on-going and vice-versa
PUNLCK - unlock "CXPBLK"
CLOCK - lock a CCB
CULCK - unlock a CCB
CULCKNF - unlock a CCB don't flush FCB
RAID - read a directory block
REDEL - release a directory element
CVTL - convert address to intra-directory pointer
RELB - release buffer
RELD - release buffer and destroy
RELM - release buffer and set modified
RELF - release buffer and set flush
CBLKIN - read a block from a channel
BLKIN - read a block from a file
BLKINW - read a block from a file (no wait)
ASBUF - assign a buffer (no wait)
BLASB - assign a buffer

FLFCB - flush FCB to FIB
FLFCW - flush FCB to FIB and wait
FLFCWL - flush FCB to FIB and wait, the parent CCB is already locked
LKBH - lock a buffer header
ULKBH - unlock a buffer header
FLBFW - flush a buffer and wait for I/O completion
UPFIB - update a FIB from an FCB
VALCID - validate a CID

SOV10 - file system common subroutines
entry points
JELLO.P - allocate a free directory element
JELUDA.P- allocate a UDA
GNFN.P - get next file name
IGNFN.P - internal get next file name
SGNFN.P - special internal get next filename
GEIB.P - search directory block for free element

UNIT - device dependent routines for disk tape line printer and mca
entry points
UOPEN.P - unit open
UCLOSE.P - unit close
GTUDB.P - get a UDB
RLUDB.P - release a UDB
GDUDB.P - get a disk UDB

WDCBK - write to disk and get free blocks
entry points
WDCBK - allocate "n" contiguous disk blocks
WDBLK - allocate 1 disk block
MWAIT - wait for bit map FCB lock with withdraw
MSWAT - wait for bit map lock for de-allocate
MWLCK - lock out LCB bit map withdraws
MWULK - allow LCB bit map withdraws
TRTBL - null terminator table

XINIT - disk initialization segment 2
entry points
XINIT.P - disk initialization segment 2
EINIT.P - disk initialization error processing

The Physical Disk

Physical Layout

Physical disk units (PU)

The fundamental element of a disk unit is the disk block or sector. AOS/VS addresses theses blocks sequentially, so that a unit with N blocks has an addressing range of 0 through N-1. These sequential addresses must be broken down into head, sector, and cylinder addresses in order for the disk unit to access the desired blocks. This translation is performed by the disk drivers.

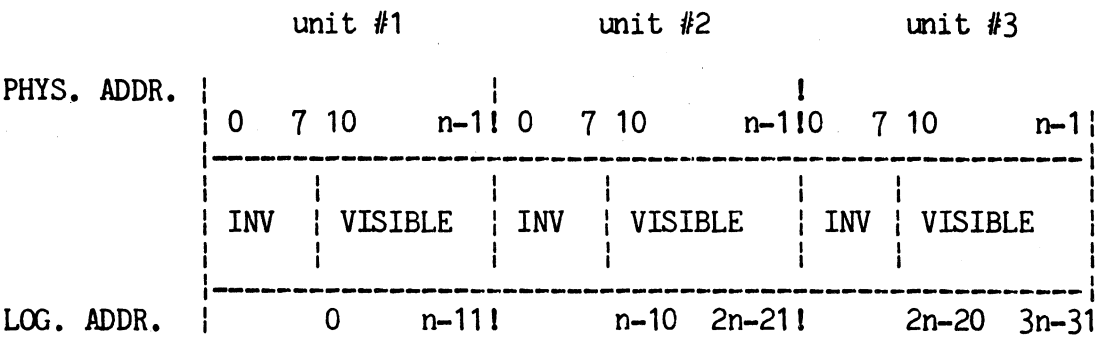
Logical Disks (LDU)

A logical disk is an association of physical disks which are made to appear as a single large disk. The purpose for this is to allow a more extended addressing space. A LDU can be composed of from 1 to 8 disk units of any mixture of AOS/VS disk types.

In order to identify the structure of a logical disk, AOS/VS requires a certain amount of each disk's address space to be reserved for the system's use. This reserved space is invisible to the user and therefore does not have an AOS/VS logical disk address.

A logical disk has a number of logical blocks equal to the sum of the numbers of physical blocks on all the disks in the LDU minus the total invisible space on all those disks. Logical disk addresses must be broken down into physical disk unit and the physical disk address on that unit. This translation is done by the disk block I/O routines.

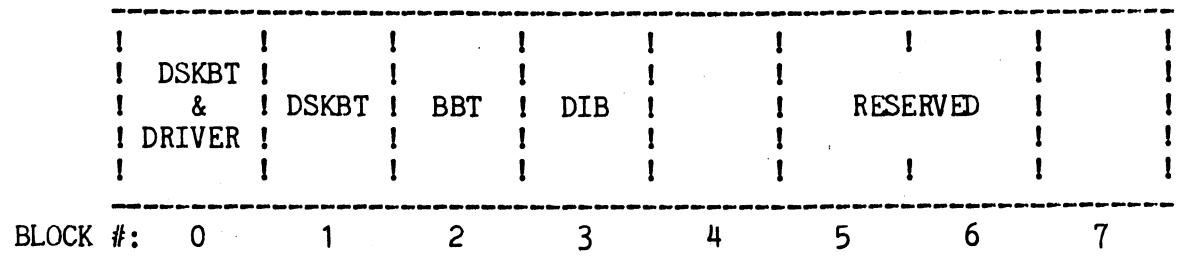
The following example compares physical and logical addressing of disks:



AOS/VS disk format

As mentioned before, each PU in a LDU is divided into two main areas. These are the INVISIBLE SPACE which is the first 8. blocks and contain information needed by the system and the VISIBLE space which is the remainder of the disk and contains both user and system data.

The invisible space is in the following format:



DSKBT and DRIVER:

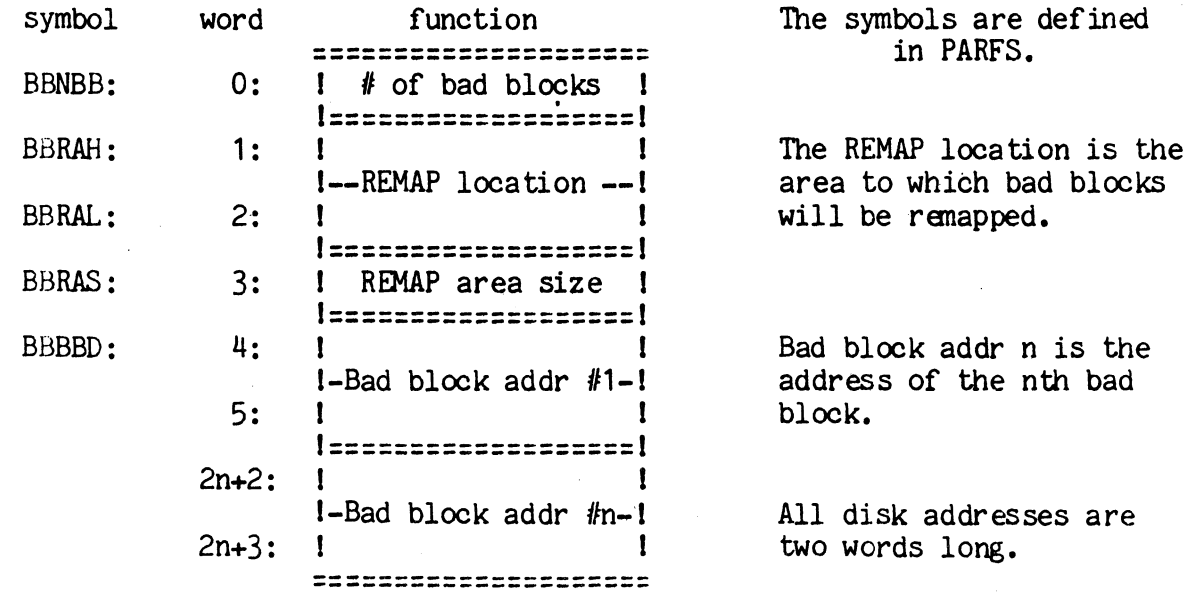
This contains the information read in by the program load switch. This contains a disk driver and the logic to read in block 1 (DSKBT).

DSKBT:

Block 1 contains the code needed to read in the DIB (block 3) and fetches the location of the system bootstrap area. It then reads in the system bootstrap and transfers to the routine.

BBT:

This block contains the AOS/VS bad block table for the PU. The table is set up as follows:



DIB (Disk Information Block)

The DIB's primary purpose is to link the units of a LDU together. At logical disk initialization time, considerable checking is done to ensure that the specified disks form a complete logical disk. In addition the DIB contains unit sizing information and pointers to system data bases.

The DIB is defined in PARFS starting at the symbol IBREV.

Briefly, the DIB contains the following information:

Disk Format Revision number
Per unit status flags
LDU unique I.D.
Sequence number of this PU in LDU
# of physical units in LDU
# of heads, # of sectors/track, # of cylinders on the PU
# of visible disk blocks on the PU
Physical disk address of the BBT (always 2 for now).

If the unit is the first of a LDU the DIB also contains the following information:

Per LDU status flags
Logical disk address of LDU name and Access control blocks
Logical disk address of bit map.
System bootstrap disk address and length
Overlay area disk address and length
Installed system pointer
LDU current and maximum sizes
"Funny FIB" of root directory (FIB = File Information Block)

VISIBLE SPACE

Several of the AOS/VS system data bases are in the visible portion of the disk. These locations are allocated by either the formatter (DFMTR) or the installer (INSTL).

BIT MAP -- indicates which disk blocks have or have not been allocated (1 bit per block -- set if in use). There is one bit map per LDU and the location is setup by DFMTR

REMAP AREA -- is the area to which bad disk blocks are remapped. There is a remap area on each PU, the location of which is setup by DFMTR.

BOOTSTRAP AREA -- is the area which contains the code for SYSBOOT.

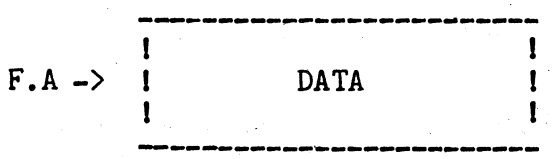
OVERLAY AREA -- contains AOS/VS system overlays; setup by DFMTR.

INSTALLED SYSTEM -- is a file that lives in the root (:) but has no name. The file is created by INSTL

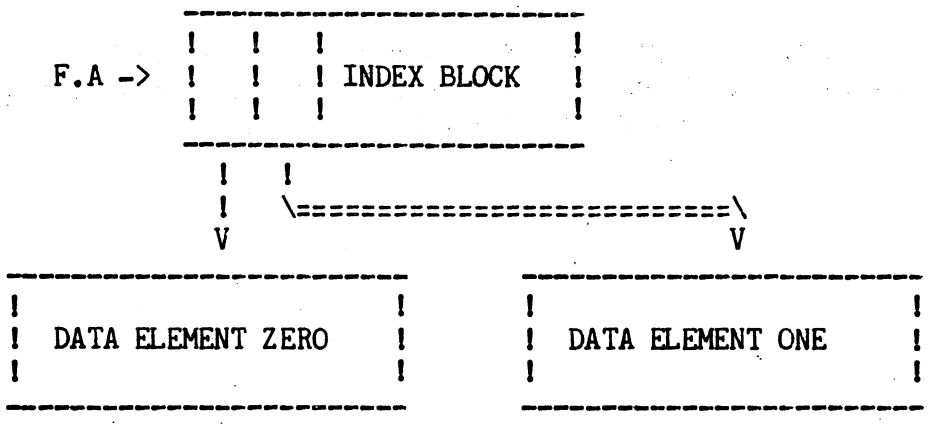
Files in the AOS/VS disk world.

Index blocks

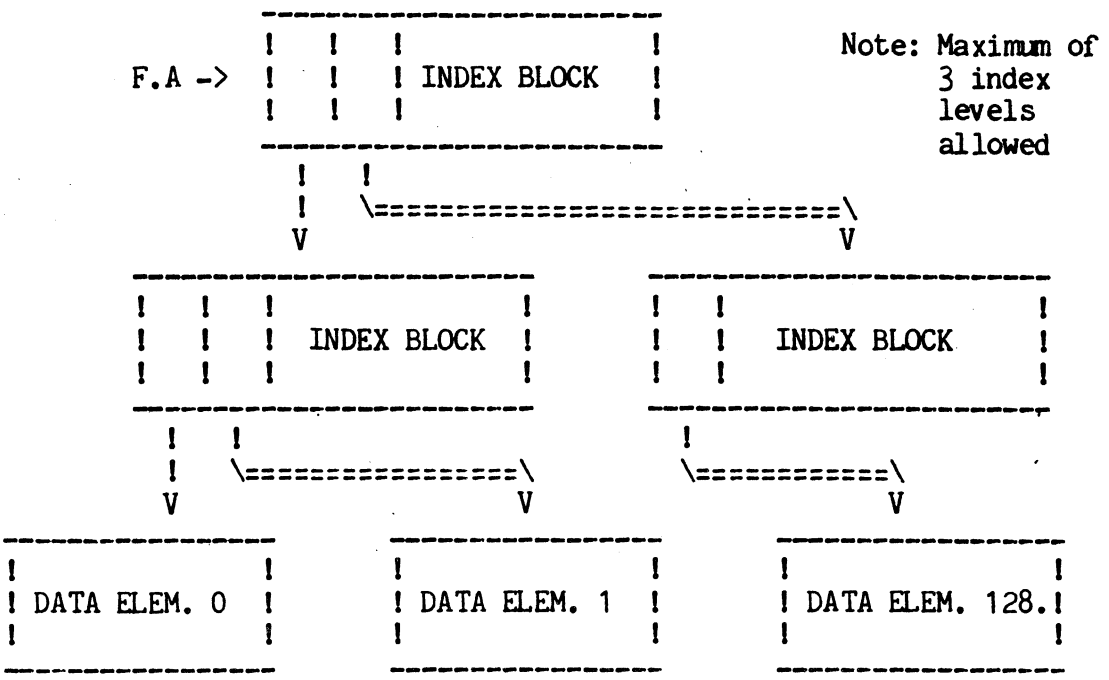
The following diagrams trace the growth of the index tree for an AOS/VS file. F.A indicates the 'first address' or where AOS/VS looks when looking for the file. Each index block contains 128 double word pointers. After data element zero is written:



After data element one is written:



After data element 128. is written:



AOS/VS disk directories

Building Blocks

The directory file (standard AOS/VS file, element size of 1) is arranged in 256 word Directory Data Blocks (DDB) which consist of various Directory Data Elements (DDE). Consecutive DDEs are not chained together. together. The DDBs can be chained together using relative pointers that consist of single precision relative block numbers within the directory. The first eight words of each DDB is a header of the following format.

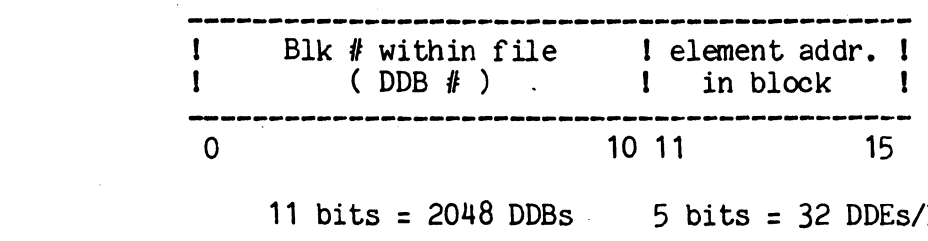
Word	Description
0	Forward Link (to other DDBs)
1	Backward Link (to other DDBs)
2-7	0

Subsequent words are allocated in the eight word elements to form the DDEs. There are six types of DDEs. They are:

Symbol	Element type code	Description
DEFNB	1	FNB (File Name Block)
DEFAC	2	FAC (File Access Control)
DEFIB	3	FIB (File Information Block)
DEFLB	4	FLB (File Link Block)
DEFUD	5	FUD (File User Data)
DEFUI	6	FUI (File Unique ID)
DEFRE	0	(Not really a type of DDE, 0 indicates free element)

The main database is the FIB. It contains or points to all vital information about a file. The FIB will point to the first FNB, the FAC and, if defined, the FLB and the FUD. All of these databases in turn point back to the FIB.

The pointers between the FIB and other Directory Data Elements are called IDPs (IntraDirectory Pointers). This format is as follows:



11 bits = 2048 DDBs 5 bits = 32 DDEs/DDB

The offset 0 (DETAS) of any of the DDEs is special. The left byte contains the element type (0-5) and the right byte contains the size in words. Note that the size of a free element (type 0) is 0 (i.e. offset 0 is 0).

The remainder of the offsets vary from DDEs. These are defined on the following pages.

File Information Block (FIB)

Symbol	Offset	Description
FINLP	1	Pointer to FNB (IDP)
FIACL	2	Pointer to FAC (IDP)
FILBP	2	Pointer to FLB (link only) (IDP)
FIUID	3	Unique ID
FITCH	4	File creation time (hi)
FITCL	5	File creation time (lo)
FISTS	6	File status
FITYP	7	File type (RH) and format (LH)
FICPS	10	File control parameters
FIHFS	10	Hash frame size (directories)
FIDCU	10	Device code (left), Unit number (right) Unit type only
FIHID	10	Host ID - Network type files
SFIBL=FICPS-DETAS+1=11		Short FIB length

FIB extension for data files and directories:

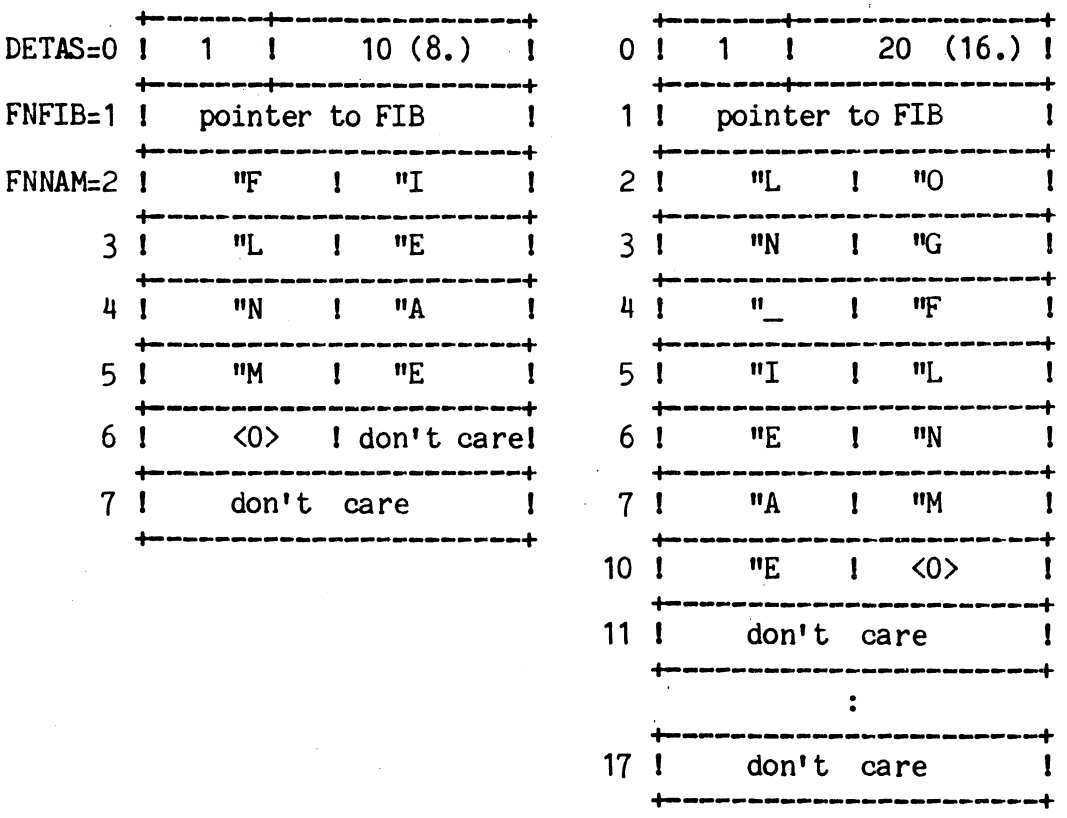
FIFW1	11	Extension for EOF in future
FIFW2	12	" " "
FIEFH.W	13	Last logical byte (EOF) (.W)
FIDFH.W	15	Data element size (.W)
FIDFL	16	Data element size (low)
FIFAH.W	17	First logical address (.W)
FIIDX	21	Current index levels (left) Maximum index levels (right)
FIIDR	22	Count of inferior directories
FIFUD	23	Pointer to FUD
FITAH.W = FITAH		Time last accessed (.W)
FITAH	24	Time last accessed (high)
FITAL	25	Time last accessed (low)
FITMH.W = FITMH		Time last modified (.W)
FITMH	26	Time last modified (hi)
FITML	27	Time last modified (lo)
FIFCB.W	30	FCB address or zero (.W)

```
FIBLT=FIFCB.W-DETAS+2=32  Full FIB length
FCOML=FIIDR-FISTS+1=15    Length of common data between FIB and FCB

FIB extension for Control Point Directories:

FICSH.W          32          Current size (.W)
FIMSH.W          34          Max size (.W)
LFIBL=FIMSH.W-DETAS+2=36     Long FIB length
```

Offsets 6 - 23 (enclosed by the dashed lines) are common to FIBs, DIBs, FCBs and 'funny FIB'.

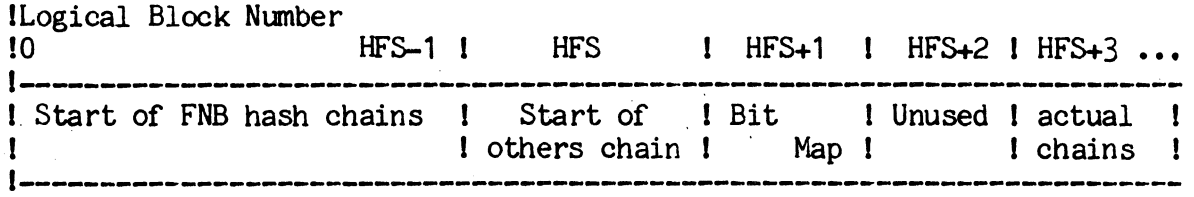
Symbol	Offset	Description
File Name Block (FNB)		
FNFIB	1	FIB pointer
FNNAM	2	File name offset
FNBLT=FNNAM-DETAS	2	FNB header length
Access Control Block (FAC)		
FAFIB	1	Pointer to FIB
FAACL	2	Access Control List offset
FACLT=FAACL-DETAS	2	FAC header length
File Link Block (FLB)		
FLFIB	1	Pointer to FIB
FLLCN	2	Link data offset
FLBLT=FLLCN-DETAS	2	FLB header length
File User Data block (FUD)		
FUFIB	1	FIB pointer
FUFFL	2	FUD forward link
FUFBL	3	FUD backward link
FUUDA	4	User data offset
FUDLT=FUUDA	4	FUD header length
File Unique ID (FUI)		
FDFIB	1	FIB pointer
FDUID	2	ID Data offset
FUILT=FDUID	2	FUI Header Length

Sample FNBs (FLBs are in the same format)



Directory format:

The following is a diagram of the internal organization of a directory file:



HFS = Hash Frame Size

The first HFS blocks of a directory contain the first filenames for files that hash to the same value. Using the standard DDB links, additional DDBs are linked to these first blocks. Each DDB so linked will contain only filenames that hash to the same value. Only specific DDBs are allocated for FNBs. The easiest way to describe which uses the bit map. Every fourth word in the bit map is used to mark DDBs used exclusively for FNBs. (16 DDBs for FNBs, then 48 DDBs for other chains etc.)

The hash value is computed as follows:

1. Take the sum of the ASCII values of the characters in the name
2. Divide by the HFS (Hash frame size)
3. The remainder of step 2 becomes the hash value for the name.

Example for the filename AOS and a hash frame size of 7.

```
A = 101
O = 117
S = 123
   -----
    343 --> 343/7 = 40 with a remainder of 3.
```

Therefore the hash value is 3.

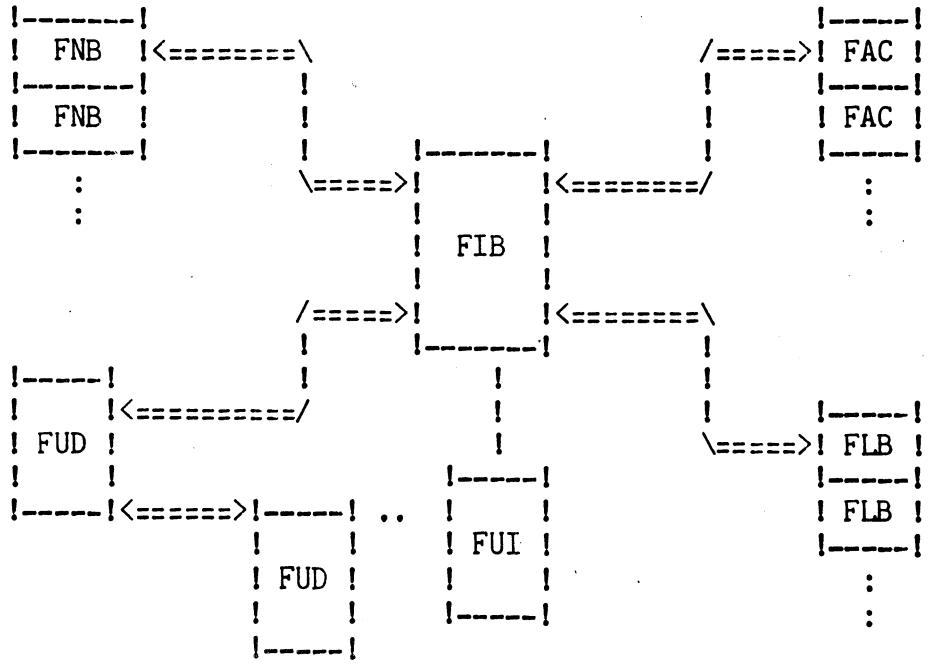
No ordering is attempted of DDBs within a chain or of FNBs in a DDB. A filename, once a hash chain has been determined, is searched for sequentially.

The next block of the directory is the anchor DDB of an unordered chain of DDBs that contains the remainder of the DDEs that make up the directory.

The directory bit map is used in allocation of blocks within a directory. It contains one bit per DDB. As mentioned above, every fourth word of the bit map is reserved for FNB allocations.

The blocks that follow the unused block (HFS+3 and on) contain the other DDBs that make up the chains.

FNBs must be in a DDB reserved for them. FACs, FIBs, FUDs, and FLBs are mixed together in the other DDBs. Pointers are bi-directional and are in IDP format.



In-core databases

The following database reside in memory. The length of time that the databases are around varies. The LDBs are around from INIT to RELEASE, the UDBs from OPEN to CLOSE, while the DCTs are permanently allocated.

LCBs and UDBs are used to convert a disk request for a logical disk into a physical request to a specific disk unit. The UDB and the DCT are used for processing the physical request.

Logical disk Control Block (LCB)

The LCB contains the following information:

Pointer to the chain of UDBs that make up the LCB (.W)
Pointer to the next LCB in the system (.W)
Buffer cache list pointer (forward and backward, both .W)
Various LDU bit map lock words
Bit map FCB address (.W)
Bit map buffer address (.W)
LDU's root CCB addr. or 0 (.W)
Current size (.W)
Max size (.W)

There is one LCB per logical disk that has been initialized. An LCB is LBBLT words long (22.). LCBs are allocated when a LDU is inited and returned when the LDU is released.

Unit Definition Block (UDB)

DCT address
Device unit number
Unit request list
Last logical address (.W)
UDB forward link (logical) (.W)
UDB forward link (physical) (.W)
Unit start addr (.W)
Number of blocks to move
Unit status word
Cylinder size in sectors (UDNSC*UDNHD)
Number of sectors per track
Flags (left byte) Number of heads (right byte)
Data address (.W)
DOA word (temp)
DOC word-used as running cylinder number
Error counter, flags, status, retry count
Temp block counter

Unit status (DIA or DIC only)
Bad block table pointer and remap address (each is .W)
Pointer to metering locations (if metering is defined)

Certain offsets are used by UNERR, the unit error report routines while other are used for fatal (hard) errors and in times of PANICs, etc.

Moving head disk UDB states (UDSTS)

Symbol	Bit	Description
DPIDL	0	Idle
DPSKR	1	Seek ready
DPSKP	2	Seek in prog
DPSKD	3	Seek done
DPIOR	4	I/O ready
DPIOP	5	I/O in prog
DPIOD	6	I/O done
DPRCR	7	Recal ready
DPRCP	10	Recal in prog
DPRCD	11	Recal done
DPSKE	12	Seek error
DPIOE	13	I/O error
DPFTE	14	Fatal error
DPGST	15	Get status

There is one UDB for each disk, LPT and tape unit that has been opened. (There are different symbol definitions for the tape not discussed here.) A UDB is UDBLT words long (41.), and allocated when an LDU is inited and released when the LDU is released.

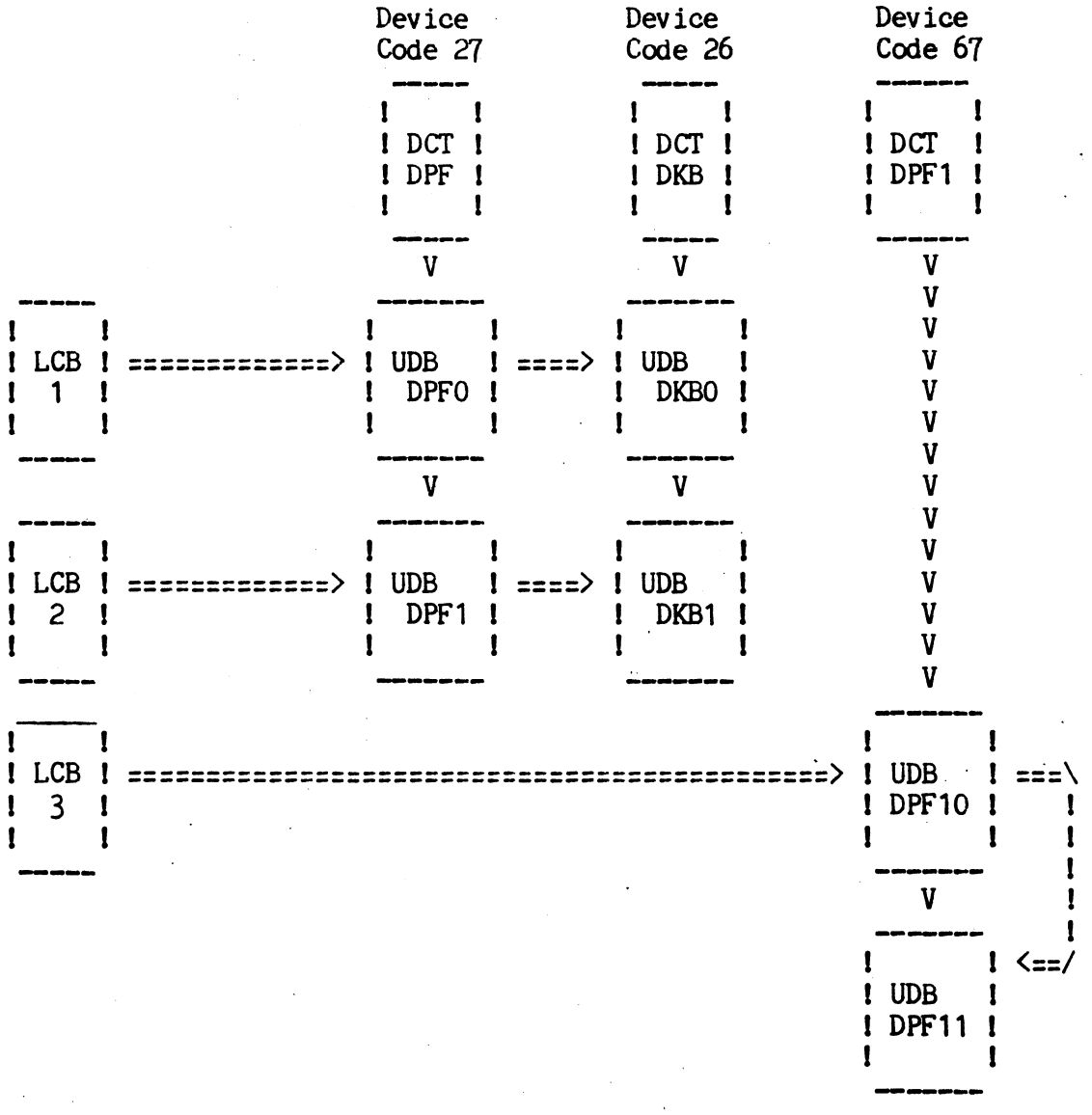
Device Control Tables (DCT)

The DCTs contain the following information:

Address of the interrupt service routine (.W)
Interrupt mask (.W)
PSR state (.W)
Device code
Status
Map slot assignments
Address of device specific routines (Initialization, powerfail, enqueue, timout ...) (.W)
UDB list (for disk, tape, and LPB/LPD)

DCTs for AOS/VS are bound into the .PR file at VSGEN time.

The following is a diagram of how some database are connected. It assumes three LCBs, two containing a fixed head disk and 6060 (96 mb) and a third containing two 6060s.



File Control Block (FCB)

There is one FCB for each file opened regardless of how many users open it. FCBs are 43. words long and contain the following information:

Pointers to LCB, CPB, UDB (.W)
Pointer to the FIB on disk
"Funny FIB" for the file (file type, status, EOF, element size
Hash frame size, first address, index levels)
Open count
Parent CCB pointer (.W)
File level counter
Pointers to buffer and shared page chains (.W)
PID and ring of first opener for SOPPF opened files
Forward and backward link for PPB chain

FCBs are loaded from the disk FIB when the file is first opened. The use count is incremented for each user that opens a file, decremented when they close it. When the use count returns to zero, the FCB is released. FCBs are found on dedicated pages in AOS/VS.

Channel Control Block (CCB)

There is one CCB for each channel (user or system) open. System CCBs are in ring 0, and user CCBs reside in ring 1. All CCBs are 27. words long and contain the following information:

Link word for when enqueued to CCBWQ
Pointer to the FCB (.W)
Parent CCB pointer (.W)
Priority / Number of blocks to transfer
Process table or CB address (.W)
User buffer address (.W)
Last block byte count
Retry count (MTA/MCA)
Logical address of most recent index blocks referenced
Relative block number of most recent index blocks referenced
post processing address
command word (READ/WRITE/DELETE/READ SYSTEM BUFFER/
TRUNCATE/ALLOCATE)

There are: 27. words/CCB
37. CCBs/page
7 CCB pages/process (256 channels)

Channel IDentifier (CID)

In order to make the kernel less dependent on the diskworld databases, the concept of the CID was introduced. Currently, the CID is simply the address of the system CCB. A kernel routine passes a CID to the Kernel/FileSystem interface routines which in turn process the request. The kernel builds a special packet for the request and does not need to know the format of the system CCB.

Control Point Directory Block (CPB)

There is one CPB for each Control point directory or LDU. Each is 6 words long and contain the following information:

Current size in disk blocks (.W)
Maximum size in disk blocks (.W)
Pointer to parent CPB (.W)

Every time a disk block is allocated or deallocated for a file, its parent CPB (pointed to by the FCB) is incremented or decremented. The CPB's parent CPB is also modified recursively.

Buffer Header (BH)

Buffer headers contain information about buffer level I/O. They contain:

LRU chain pointers (forward and back) (each .W)
Buffer header chain pointers (forward and back)(each .W)
Use count
Buffer Address
Data address on disk
Number of blocks (if not a system buffer request)
PTBL address or physical block number

Protected file permission blocks

Protected file permission blocks are allocated each time a user does a ?PMTPF system call targeted at a PID that does not already have any access to the specified file. Permission blocks are enqueued off of the file's FCB (FBPPB). Permission blocks are 14. words long, and are found in pages dedicated to this type of database. The first PPB on each page acts as a header. There are 72. normal PPBs on each page plus the header PPB. The normal PPB contains:

PPB forward and backward links (both .W)
The PID/Ring of the first opener
The PID of the process that can use the file
The FCB address of the file that this PPB controls.
Five words of permitted access control (one per rings 3-7).

The header PPB on each PPB page is in the following format:

A pointer to the next and previous PPB page (both .W)
A forward and backward link (both .W) of free PPBs on this page
A count of free PPBs on this page
The page address of the page

Input/Output Control Block (IOCB)

IOCBs contain the following information:

Forward IOCB chain link (for IORUN.W) (.W)
Backward IOCB chain link (for IORUN.W) (.W)
CCB address (.W)
IOCB status word
Save levels (IOCB routines do not use the stack)

The following locations are used by the diskworld as a buffer header to enqueue the request (until the dashed line)

Data address (.W)
Status
Link to next BH (forward and back) (each .W)
LRU forward and backward links (each .W)
The LRU forward chain is used to hold the IOCB address when running an IOCB request
#blks to transfer this element
physical memory block or PTBL address
unpend(post process) address
TCB address
Data address (.W)

Q=file element #=blk#/element size IOQLO (.W)
Remainder from Q computation(.W)
I/O error code
Counter of #levels of indexing needed
Indexing word (<0><x1><x2><x3>) (.W)
Current #levels in file
Data element size (.W)
Temp variable IOICB(=IOVAR)
Amount of bytes transferred so far
FCB address

Block count
Buffer header address

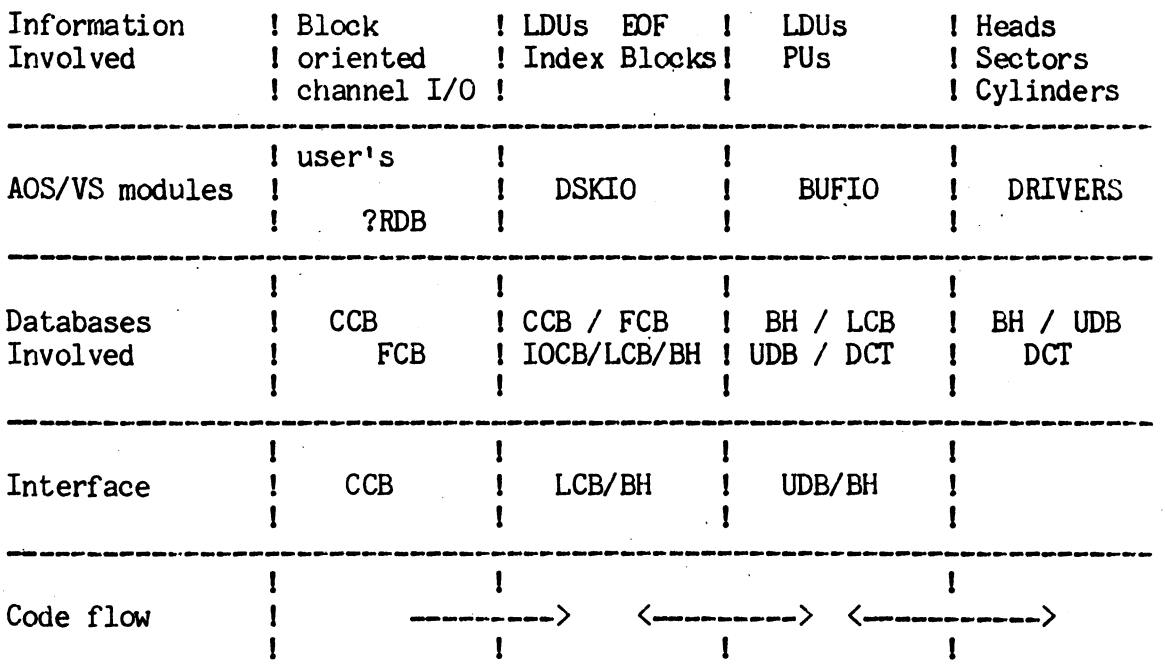
The save levels are used to hold return addresses from subroutines. IOCBs are allocated as they are needed (see later in chapter) and are 64. words long.

IOCB status states:

IOCB dormant
Waiting for I/O
IOCB ready
Waiting for a buffer
Waiting on the bit map
Waiting on file IOP

I/O Processing

The following diagram is an attempt to follow a ?RDB call through the various system levels.



Notes:

1. The code can bounce back and forth between DSKIO and BUFIO in the case when BUFIO returns an index block instead of a data element to DSKIO, which then must ask BUFIO for the next level.

2. The code can bounce back and forth between BUFIO and DRIVERS in the case that a request reaches the end of a PU but not the end of a LDU.

IOCB Processing

1. Initialized by NQCCB. This routine will allocate an IOCB, place the location of RUNRD into the IOCB at offset IOSPC.W, and flag the IOCB ready to run.

2. RUNLC will find the IOCB ready to run and begin execution at @IOSPC.W

3. Code path will have to pend at points waiting for disk req. This is done with a call to WAIT, which will put the pending address in the IOCB at offset IOSPC.W and jump back to RUNLC.

4. When the disk request is completed, the interrupt will wake up the IOCB by setting its ready to run flag.

5. Loop back to 2. This will continue until the IOCB processing is completed.

6. IOCB allocation:

   a. The max number of IOCBs for the system is determined by available memory (if the diskworld can get an IOCB, it will).

   b. The min number of IOCBs is defined by SCMNI which is 2.

   c. The system will dynamically grow the pool of IOCBs. When an IOCB is done and at least one IOCB is on the free chain, AOS/VS will return the completed IOCB to the free memory pool (i.e AOS/VS always attempts to keep at least one IOCB on the free chain, and always SCMNI IOCBs allocated.)

   d. When attempting to allocate an IOCB, GIOCB first attempts to assign one from the free chain. If this is empty, GIOCB will attempt to grow the IOCB pool. If it can, it does, and returns the new IOCB to be associated with a CCB. If it cannot, the it takes an error return which forces the CCB to be enqueued off CCBWQ.W.

Some words about WAIT

1. Called with a XJSR WAIT.

2. Store away the return address (AC3) into offset IOSPC.W of the IOCB

3. Store away the status (AC0) in offset IOSTW.

4. Enable interrupts.

5. Try to run any other IOCB requests (Jump back into RUNLC)

NQCCB (Module: DSKIO)

NQCCB enqueues CCB requests. It attempts to allocate an IOCB. If successful, the IOCB is enqueued to IORUN.W. If not, the CCB is enqueued to CCBWQ.W

1. NQCCB - Convert the ring 1 CCB address to a ring 0 address (LPHY)

2. Store the post processor address (in AC0 at time of call) into the CCB

3. Reset the CCB error flag in the CCB.

4. Try to allocate an IOCB. If this fails, goto step 10.

5. Store the CCB and FCB addresses in the IOCB.

6. Set the IOCB status to 'ready to run' and put location RUNRD in offset IOSPC.W of the IOCB. (RUNRD is the address at which IOCB processing begins).

7. Enqueue this IOCB in I/O priority order on the list of IOCBs (IORUN.W) (disk priority = PNQF for resident and preemptible processes, = 377 for all swappable processes)

8. Call DWAKE to ready the disk manager control block.

9. Return (this was called with a XJSR).

10. We must wait for an IOCB. Enqueue the CCB in priority order onto the CCB waiting list (CCBWQ.W). Increment the waiting CCB counter (CCWC), and if necessary the max count of waiting CCBs (CCMX).

11. Return

RUNLC (Module: DSKIO)

RUNLC runs all readied IOCBs for all LCBs. RUNLC is entered when the scheduler runs the DMTSK control block. It is also entered when an IOCB pends (PEND).

1. Enable interrupts

2. Clear DWAKE called flag.

3. Set LCBP.W (current LCB pointer) and IOCBP.W (current IOCB) to -1.

4. Find a ready IOCB. Determination is done by examining the IOCB running list (IORUN.W) to see if any IOCBs are ready.

5. Save the IOCB address in IOCBP.W (IOCBP.W always contains the address of the currently executing IOCB).

6. Save the address of the LCB associated with this IOCB in LCBP.W.

7. Execute the code specified by the IOCB. (XJMP @IOSPC.W).

RUNRD (Module: DSKIO)

All individual IOCB code paths start here. This address is loaded into offset IOSPC.W of the IOCB when it is initialized. RUNLC jumps through this offset which always contains the IOCB restart address.

Upon starting:

AC2, AC3 = IOCB address
The LCB is in LCBP.W (put there by RUNLC)
The IOCB is in IOCBP.W (put there by RUNLC)

All values stored as temporaries are stored in the IOCB.

1. Initialize some IOCB temporaries - zero the error code and the last block correction (number of bytes in last block).

2. Wait for any IO to this file to stop by checking the IO in progress bit in the FCB.

3. Store the following in the IOCB:

PTBL address	IOMAP.W
current # of index levels	IONLV
data element size	IODEH.W

4. If the file type (from the FCB) is a disk unit type (?FDKU), then if the beginning block number of the request is greater than the element size, signal an error (EOF). Otherwise, store the block number in the file element offset location of the IOCB (IOREH.W) and set the element number (IOQHI.W) to zero.

5. If this is not a disk unit file, then calculate the element number (Q) that the request starts in by dividing the starting block number by the element size (store the results in IOQHI.W) and store the remainder in IOREH.W.

6. Calculate the number of blocks that can be transfered from this element. If the complete request cannot be satisfied, save the count of how many blocks can be transfered in offset IOBNL, and the remaining number of blocks in the request in offset CBNBK.

7. Dispatch (LJSR) to the appropriate READ/WRITE/DELETE/ALLOCATE/ TRUNCATE. (The command code is in the CCB (CBFLG)). See below for each individual code path

8. Upon returning from the element processing, if an error has occured, set the error flag in the CCB, invalidate the last index level entry, and goto step 12.

9. Update the in memory data buffer address (increment by 256 multiplied by the number of blocks transferred.

10. Update the count of total bytes transferred.

11. If there are any blocks remaining to be transfered (CBNBK<>0), then save the updated byte count, increment the data element number (IOQHI.W), and zero the IOREH.W (look at the first block in the new element), and go back to step 6.

12. Copy the error code (if any) from the IOCB to the CCB

13. Call the CCB unpend processor (offset CBUPD.W in the CCB points to the code path).

14. Wake up anyone waiting for this I/O to complete.

15. Dequeue this IOCB from IORUN.W.

16. If any CCBs are waiting for an IOCB, get the first on the list, and assign this IOCB to that request. Initialize the IOCB by copying the new CCB and FCB addresses into the IOCB, put the address of RUNRD in offset IOSPC.W, and mark the IOCB ready to run. Dequeue the CCB from the CCBWQ.W list, enqueue the IOCB to IORUN.W (in priority order), and jump back to RUNLC.

17. If no CCBs are waiting to run, return the IOCB to the free pool or free memory pool depending on the state of IOFRE.W, and the total count of IOCBs

18. Jump back into RUNLC to run any other IOCBs.

The following routines are called from the IO processing routines documented below:

INDEX - Loops through the index levels. It takes the normal return if it reaches the data level, the zero return if it reaches a hole in the index, or the error return if an error is encountered during the index reading.

INDEX calls LBLKN to get an index block. LBLKN determines if the block is still in memory as a system buffer, and if it is, waits for any IO to the buffer to complete and returns. If the block is not in memory, LBLKN assigns a buffer for the request, NQs the buffer header block, NQs the buffer header, waits for the IO and takes the good return. If no buffer is available LBLKN waits for one.

GROW - Allocates space on the disk for a file. Allocates necessary index blocks until lowest level, updating the previous level index blocks, and then allocates a data element. GROW calls WDBLK (allocate 1 blk) and WDCBK (allocates n contiquous blocks).

GROFL - Decides if the number of index levels must and can grow (the file might already have the max number of index levels). If so, the address of the new index index block is put in the file's first address, and the old first address is put in the new zeroed index block.

CKEOF - Checks for the EOF condition. On writes, shared reads and system buffer reads, the EOF is extended. On normal reads, the number of blocks to read is decremented, and the EOF flag is set. On all calls to CKEOF, the request is checked against the absolute maximum of 2^32, and truncated if beyond that.

PARSQ and MATCH - Each request can be broken down into a maximum of 3 index levels, and each level can have 128. different pointers to the next level, therefore, each request can be represented by three 7 bit numbers, designated x1, x2 and x3. PARSQ will determine the three 7 bit values and MATCH will compare the current x1, x2, and x3 with the previous requests values. If a match occurs, then the request can be optimized. If a match occurs on all three levels, there is no need to transverse the index levels, if a match occurs on two levels, then only the last level must be read. The occurance of matches is recorded in the metering locations X1MATCH.W, X2MATCH.W, and X3MATCH.W (see the end of this chapter for more information)

The following routines are dispatched to by RUNRD (step 6)

RDDEL - Reads one or part of one data element

1. Check EOF condition, but do not extend file.
2. Check number of blocks to read. If zero, just return.
3. If shared read, jump to SHRD.
4. Compute indexing offsets (XJSR PARSQ), and try to use matched data
5. If the file first address is 0, then goto step 12.
6. Loop through the index levels (call INDEX)
7. If INDEX took the zero return, we tried to read a hole.. Jump to step 11.
8. Set up the buffer header offsets of the IOCB. These contain the data buffer address in memory, and flags to indicate that this is a user request, and a read.
9. NQ the buffer header, and wait for the IO to complete. (XJSR GIOWT which will call NQBHR in BUFIO, and the wait for IO completion).
10. Return (Jump back into RUNRD at step 7).
11. Release BH of last index block (residue from INDEX)
12. Clear core indicated by the TCB using the routine IMCLR. An error from the routine will cause a PANIC 14033. (At the point at which we came to step 12, we either tried to read an empty file or attempted to read in a hole, either of these operations will return zeroes to the user).
13. Invalidate the index offsets that were not actually read.
14. Return (Jump back to RUNRD at step 7).

SHRD - Shared read

1. If we are reading in a hole, or past the end of file, we must allocate new elements and their associated indices so that anybody else requesting the read will get the same information.
2. Compute the indexing (PARSQ), and try to optimize the read (MATCH)
3. Fill the new areas allocated with zeros.
4. Read in the data element (jump to step 8 of RDDEL).

WRDEL - Write a data element

1. Check the end of file condition and truncate the request if necessary
2. If a zero length request, just return
3. Set the modified and the flushed bits in the file's FCB
4. Grow the file past EOF if necessary
5. Call PARSQ and attempt to optimize the index reads
6. If necessary, loop down the index levels (INDEX)
7. If a hole in the index was found allocate and zero the new element
8. If really a write (as opposed to a allocate) then set up the request in the buffer header and enqueue the request
9. Update EOF if necessary
10. Return to RUNRD

DELFIL - Delete file's space from disk.

DELFIL steps through indexes deleting blocks with calls to DEBLK (delete single block) or DEBKS (delete n contiquous blocks)

TRUNCFIL - truncate a disk file

1. Check to see if the new EOF is before the old EOF. If not, the request is either illegal or not necessary
2. Call PARSQ to get the index structure
3. Call index to step down the index levels if necessary
4. Release all data elements after EOF
5. Shrink the number of index levels if possible and return any unnecessary index blocks to the general disk pool

This ends the documentation on RUNRD.

CCB post processing

PPCUS/PPCUB - CCB post processing for user I/O requests

1. Pass the number of bytes transferred and error code if any back to the TCB and unpend the TCB

2. If this is a ?BLKIO call then get the packet length and address save it on the stack, set up byte count in packet and restore it unwire it and pass it back.

3. If this is not an allocate request, then update the modified and referenced bits because the data channel and BMC maps cannot.

   If this was a read request, set the modified and referenced bits for each page

   If this was a write request, set the referenced bit for each page

4. Unwire the pages involved in the transfer

5. Decrement the active call count for the user

6. Wake up the process (IWKUP)

7. Unlock and unwire the CCB

8. Return

RNA - Read Next Allocated block (?BLOCKIO)

This routine reads the next allocated data element in the file. By "next" we mean starting the search at the index offset immediately following that of the unallocated data element requested. The algoorithm searches th rest of the current index block, goes to its parent index block, searches the rest of it, and so on until either an allocated data element/index block is found or there are no more parent index blocks to search in which case EOF is returned. If an allocated slot is found, then we travel down the right hand side of the index structure (we had essentially traversed the left hand side when trying to find the next allocated index block/data element) until we get to the data level and have the LDA of the allocated data element. If, as we travel down the right hand index structure, we ever encounter an index block which has nothing in it, then we panic since we should never have allocated an index block without putting something in it. once we have found the allocated data element and have its logical disk address, we can figure out its logical file address by using its index block offset values and the file's data element size. before returning, we turn on the phase II RNA request bit in the iocb so that if we hit another unallocated data element later in the request, it won't call this routine again but instead will terminate the read request.

PPCBI - post processor request for the system blockin requests
PPCSY - post processor for system read and write request
PPCSR - post processor for system shared read CCB requests

1. If PPCBI then return the buffer header address to caller in the CCB

2. If PPCSR copy disk address from IOCB to CCB for waiter.

3. Reset the CB or PTBL not ready to run bit and zero the unpend code

4. Disable interrupts go try and set next to run and enable interrupts.

5. Return

NQBHR / NQBH1 (Module: BUFIO)

NQBHR enqueues a buffer header to the UDBs. It takes as input a LCB address. NQBH1 takes as input the actual UDB address.

1. Save unpend address in the buffer header.

2. Set IO in progress in BH. If already set, PANIC 14050.

3. If this was a NQBHR call (not NQBH1) then translate the LCB address into a UDB address. This is done by scanning down the chain of UDBs associated with the LCB until the request address is less than the maximum address on the unit. If we come to the end of the chain and have not found a unit we will PANIC 14047

4. Mask out interrupts from the diskworld

5. Meter the number of requests on this unit.

6. If nothing is on this unit's request list, enqueue this request, and start the device (XJSR STUNT)

7. If something was already on the unit's request list, just enqueue the buffer header to the UDB; and meter the fact that the disk was active at the time the request came in.

8. Increment NQDBHRS (number of currently enqueued buffer headers)

9. Restore the interrupt mask

10. Return with AC0=UDB address

IODON (Module: BUFIO)

This is called from the interrupt service routines.

1. If this was a physical I/O request, copy the DIA, DIB, ECC ... words into the request state block and goto step 4

2. If an error has occured, store the information into the error locations in the UDB, and wake up the system manager (which will report the error)

3. If there are more blocks to transfer, check to see if this is an LDU or single PU. If this is the last unit in a LDU or a single PU signal an error (if PU signal EOF -- if LDU signal PANIC 14052). If it is not the last unit of a LDU, enqueue the request to the next UDB in the LDU and start the transfer.

4. Call the post processor for the request.

5. Decrement NQDBHRS

6. If there are anymore request on this UDB, start the unit.

Disk drivers

Disk drivers are the base level support of disk in AOS/VS. There are eight major routines for each driver. Each DCT will point to the relevant driver modules.

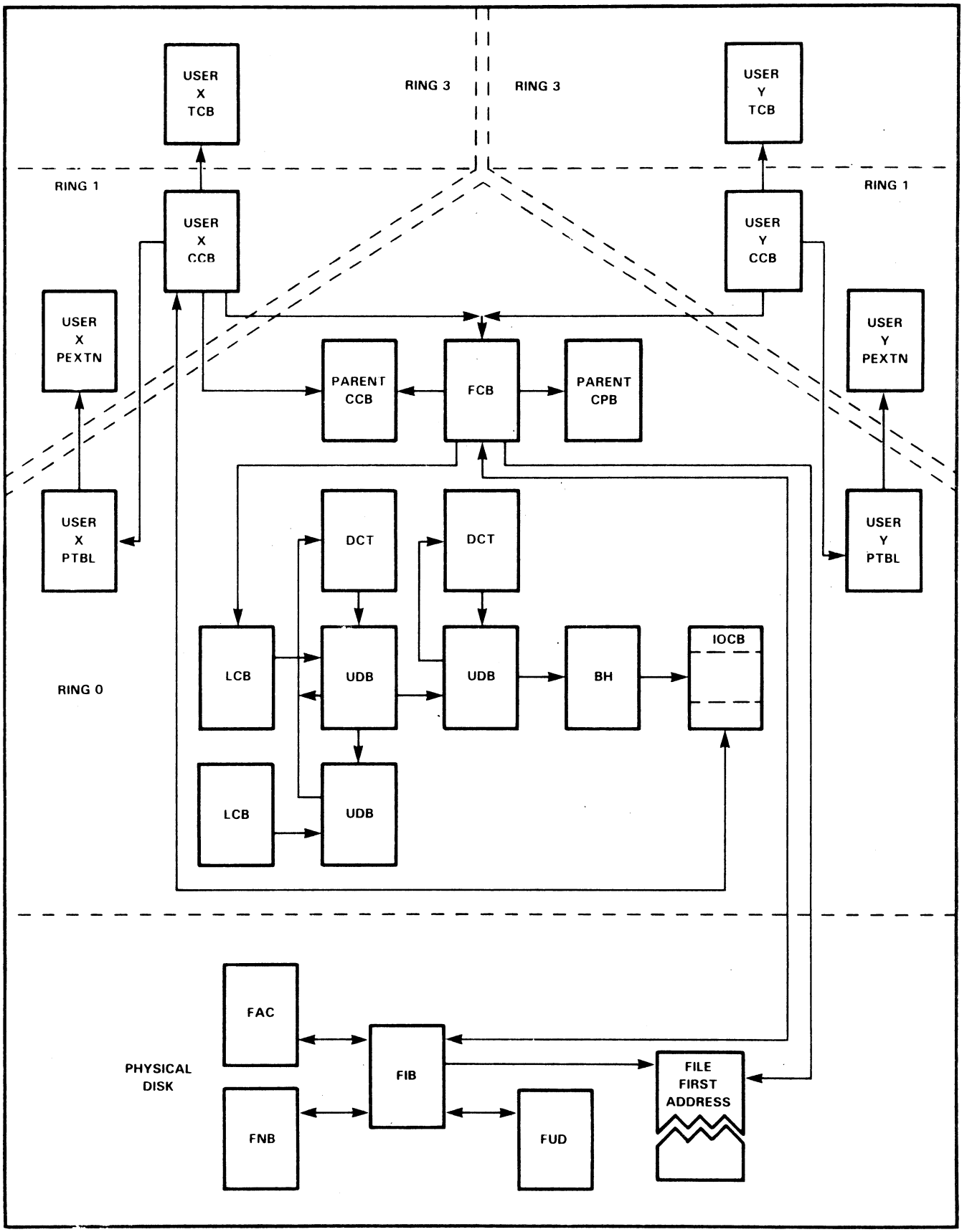
Routine	Function	DCT offset
a. Interrupt Service	- device interrupts, restart, errors	DCINS.W
b. Start Up	- start current request	DCSTR.W
c. Set Up	- setup next request, calculate seeks.	DCSUP.W
d. Initialization	- Init UDB	DCTIU.W
e. Enqueue	- Enqueue new request to UDB. (DKBNQ enqueues in FIFO order DKANQ attempts optimization)	DCENQ.W
f. Check timeout	- Check timeout block routine	DCCTO.W
g. Time out routine	- Routine to handle device timeout	DCRTN.W
h. Powerfail restart	- Device dependant restart code	DCPRS.W

For quick reference, the following demonstrates the relationships between device name and modules that handle that device.

Device Name	a	b	c	d	e	f	g	h
DKB	DKBIS	DKBST	DKBSU	DKBIU	DKBNQ		DKBTM	DKBRS
DPD	DPAIS	DPDST	DPDSU	DPDIU	DPANQ	DPACT	DPATM	DPDRS
DPF	DPFIS	DPFST	DPFSU	DPFIU	DPANQ	DPACT	DPATM	DPFRS
DPG	DPAIS	DPDST	DPDSU	DPGIU	DPANQ	DPACT	DPATM	DPGRS
DPI	DPAIS	DPDST	DPDSU	DPIIU	DPANQ	DPACT	DPATM	DPGRS
DPJ	DPJIS	UNCST	DPJSU	UCNIU	UCNNQ	IODON	UCNTM	UCNRS
DPM	DPMIS	DPMST	DPMSU	DPMIU	DPANQ	DPACT	DPATM	DPMRS

Note that many routines are shared. a-h refer to functions above ie a=interrupt service, b= startup etc.

The overall disk world diagram



Kernel / File system interface

In an attempt to make the kernel less dependent on specific file system databases and routines, the following concepts have been included in the operating system.

NQCRQ -- Enqueue a request (via NQCCB) and wait or its completion (via CWAIT). The caller passes a packet and the file's CID.

NQLDRQ -- Enqueue a request to a logical disk unit (via NQBHR)

CH.ACC -- Return a file's access control bits

CH.EOF -- Return a file's EOF pointer

CH.FTY -- Return a file's file type

CH.LDA -- Return a file's first logical disk address

Shared pages

Shared pages under AOS/VS exist in one of three states. These are:

1. Not in use and not in memory
2. Not in use and in memory
3. In use (and in memory)

AOS/VS maintains two chains to manage shared pages. These are:

1. LRUCH - the system LRU chain - this chain (of CMEs) contains pages that have been released from a processes working set. A shared page with a use count of 0 will be on this chain

2. FCB chain - this chain (of SH[shared page headers]) contains shared memory pages associated with a file (regardless of use count)

AOS/VS also maintains a database called the CDE for each shared data page in the user's address space. Bits 0 - 7 contain the CCB number that describes the file that the page belongs to, and bits 8-31 contains the logical page address (block number/4).

When a user executes a ?SPAGE, AOS/VS will simply build a CDE entry for that shared page. When the page is eventually accessed,the user will fault on the page, and control will pass to the routineREADSH. READSH will

calculate the disk file address of the page if the page is in a .PR file, or it will lookup the file address and CCB # in the appropriate CDE. Now that we know the logical disk address and CCB, the corresponding FCB's SH chain is searched for a match. If a match is found, then the page does not have to be read in, the page frame for that page is put into the PTE, and if the frame was on LRUCH, it is dequeued. If the page is not in memory, a shared header is built and locked; a page is allocated from GSMEM, and the 4 blocks are read into it. Then the new page frame value is placed into the PTE. AOS/VS will increment the use count (in the CME) for that page. In both cases, an LPA will be allocated linking the page and the logical address in the user's space

When a user (or the system when the user terminates) releases a page, AOS/VS will decrement the use count. If the count is not now 0 we are done. If the count is now 0, AOS/VS will put the page (CME) on LRUCH, where it will sit until either:

a) Someone needs that shared page again (in which case we take the page off LRUCH, and increment the use count

or:

b) Someone needs the page for something else. In this case the page will be flushed if necessary and the page will be removed from LRUCH, and the FCB chain.

Key Diskworld page zero locations

The following locations in page zero (defined in SZERO.LS and STABLE.LS) are useful in examining the disk world. Some locations are pointers to chains, others are metering locations, counter or flag words.

SZERO.LS locations:
STABLE.LS locations:

LCBP.W	Pointer to current LCB, used by DSKIO
IOCBP.W	Pointer to current IOCB, used by DSKIO
.MLCB.W	Pointer to a linked list of defined LCBs for the system
.ELCB.W	Pointer to the end of the LCB list.
IORUN.W	IOCB running list head
IOTAIL.W	IOCB running list tail
IOFRE.W	IOCB free list head
CCBWQ.W	CCB wait queue head
RLCFL	RUNLC flag - non zero if there is something for RUNLC to do.
WFLAG	<>0 - Base level waiting for system buffer
BFLRU.W	Head of free buffer chain, in least recently used order
BTAIL.W	Tail of free buffer chain
NOBUFS	Number of times that no buffer was available
BUFCN	Number of buffers currently on the BFLRU.W
BUFLO	Lowest number of buffers ever on BFLRU.W

BFMIN	Starting number of system buffers
FCBCH.W	FCB chain, chain of 8 word descriptors defining the physical memory blocks being used for FCBs

Metering locations:

NQDBHRS	Number of buffer headers currently NQd to disks
PRDRQS.W	number of physical disk read requests
PWTRQS.W	number of physical disk write requests
PBLKSR.W	number of physical blocks read
PBLKSW.W	number of physical blocks written
FCBRDS.W	number of reads found on a FCB buffer list
LCBRDS.W	number of reads found on an LCB buffer list
X1MAT.W	number of 2nd level index matches
X2MAT.W	number of low level index matches
X3MAT.W	number of data level index matches
IOTC	IOCB total count
IOAC	IOCB active count
IOMX	Max value of IOTC
CCWC	Count of CCBs waiting for IOCB
CCMX	Max value of CCWC

Logical Disk Structure

The basic concern of this section is to determine how AOS/VS relates a disk filename with the physical disk addresses of the data associated with the file.

A. File Structure

The reader should already be familiar with one of the steps of data location, namely, how AOS/VS finds data elements from random index blocks (RIBS) in progressively larger files:

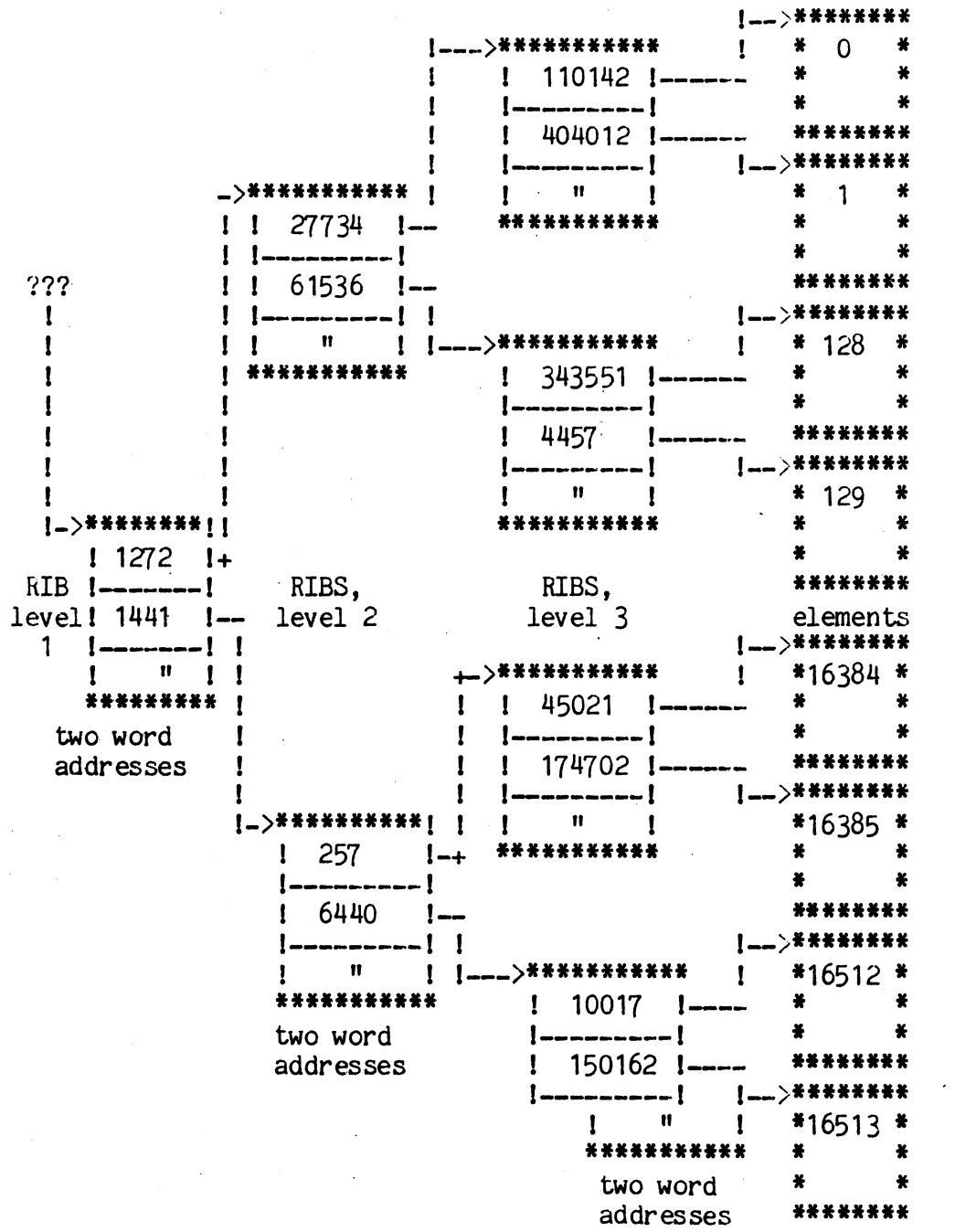


Figure 1. File Structure.

The obvious next step is to determine how AOS/VS finds the first RIB. We will approach this problem over the next several secions by investigating the overhead structures (invisible space) and directories of the logical disk.

B. Physical layout of visible and invisible spaces

The size of the AOS/VS file pictured above is phenomenal; It has the potential of greatly exceeding the physical capacity of any real disk. So, in order to sidestep the file growth limitations imposed by the physical units, AOS/VS supports the concept of a logical disk unit (LDU) by simply grouping multiples of physical disk units (PU's) together and viewing the amassed sectors as one contiguous address space. The overhead structures necessary to tie the units together are written onto the disks as they are serially formatted during a single DFMTR run. In dealing with files, the AOS/VS tasking world sees the disk exactly as above, i.e., with no clues as to which sectors belong to which PU's. It is not until the disk driver is requested to actually do I/O that the details of relating a disk address to drive, sector, surface, and cylinder (via the overhead structures) comes into play.

(One consequence of the 'contiguous address space' advantage is that none of the physical units can be missing when the LDU is brought on-line. To create a larger or smaller LDU requires re-formatting. This should not be confused with grafting, where the address space of an LDU is presented to another LDU in the form of a Control Point Directory; because of the physical boundaries, the initial LDU cannot let its own file structures spill out onto the grafted LDU.)

Figure 2 shows the basic block allocations which exist on a rev 4.xx logical disk consisting of two single density Zebra (6060) disk packs. NOTE that blocks are referenced in the diagram by either a logical address (LOG xx) or a physical address (PHY xx). The physical addressing is the more primitive method of identifying the series of blocks spanning the two disks; the progression is from unit one physical block 0 clockwise to physical block 556027 (8), then to the first block on the second pack (= physical block 556030 (8)), and finally clockwise to physical block 1334057 (8). The disk driver uses physical addresses to determine sectors, surfaces, and cylinders according to the following formula:

(local block #) = (physical block #) - (# blocks on all packs before this one)

$$\frac{\text{(local block \#)}}{\text{(\# sectors per cyl)}} \bmod 45 = \text{clyinder \#} + \text{remainder1}$$

$$\frac{\text{remainder1}}{\text{(\# sectors per sur)}} = \text{surface \#} + \text{remainder2}$$

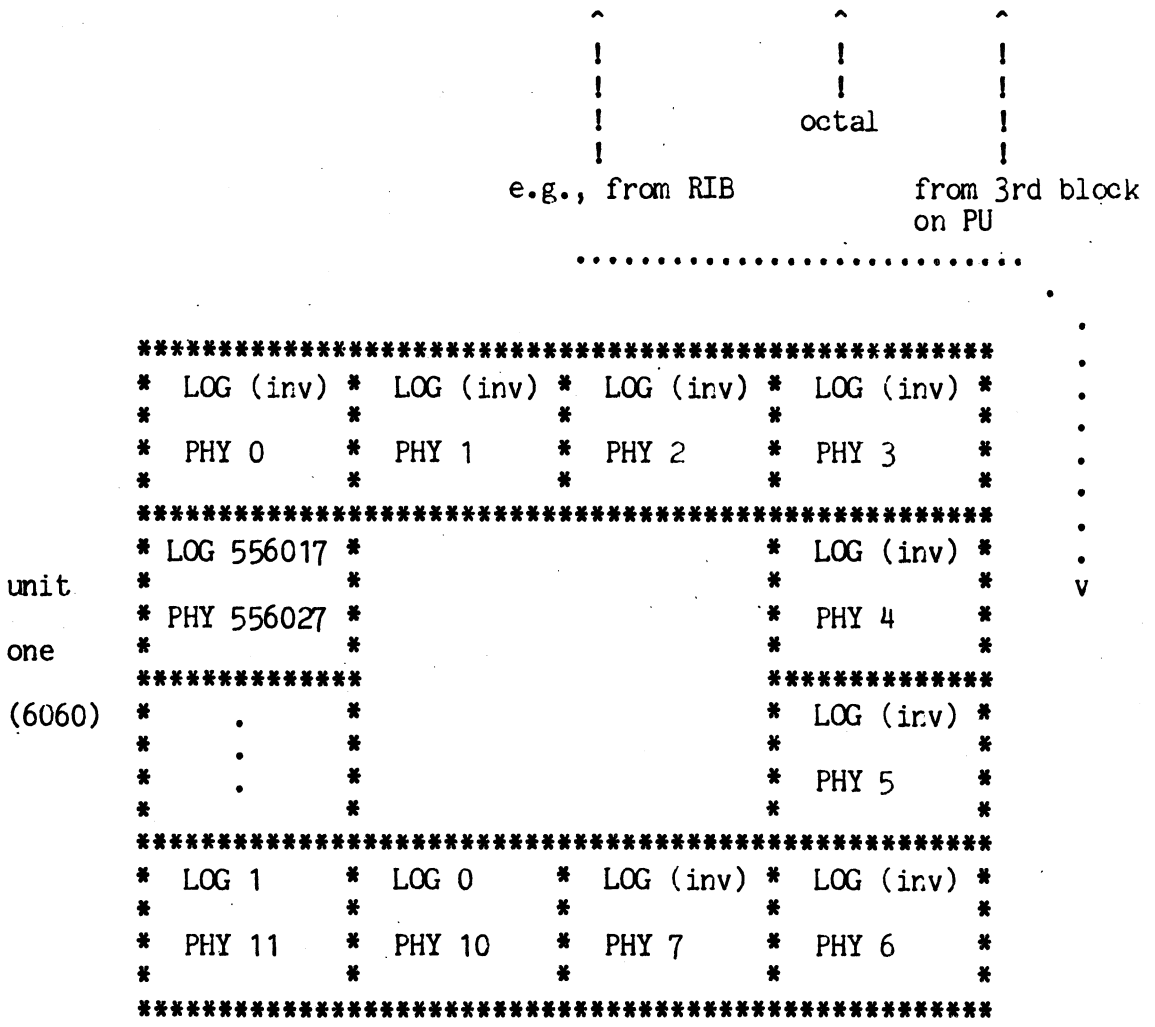
sector # = remainder2

As will be seen shortly, the sequence # of the disk in the LDU, the # of sectors per cylinder, the # of sectors per surface, etc., are kept in each PU's third block.

AOS/VS does not use physical addresses for file transactions because doing so could conceivably result in overwrites of the LDU overhead structures. This could happen, for example, if someone inadvertently changed a RIB pointer while doing some disk editing. NOTE in Fig 2 that the first 10 (8) blocks of each PU are not included in the counting sequence of logical blocks; These collectively comprise the 'invisible space' of the LDU because any algorithm working exclusively from logical addresses (like an AOS/VS system task) can never access information in these areas.

For the disk driver to convert logical addresses to physical, the procedure is:

physical address = logical address + (10 x disk sequence #)



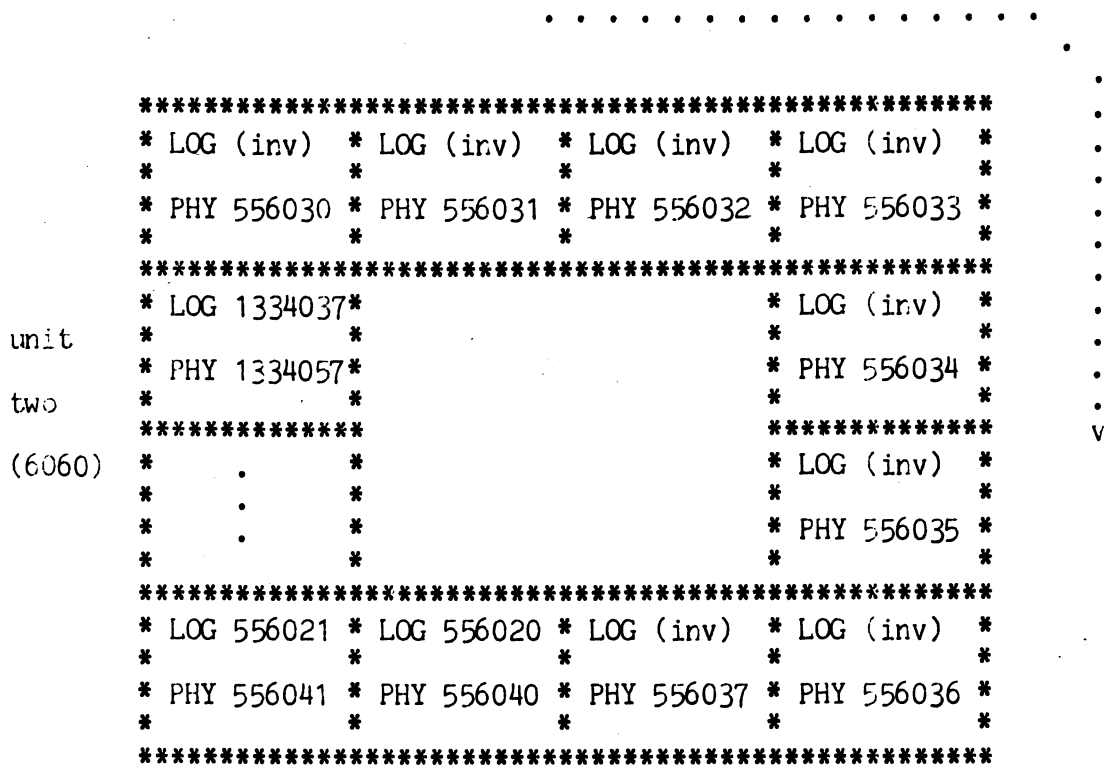


Figure 2. Logical disk addressing.

C. Details of invisibile space for the first PU

The first 10 (8) blocks in Figure 3 illustrate the format of invisible space for the first PU in an LDU after DFMTR and INSTL have run. The information contained in these blocks is fairly straightforward:

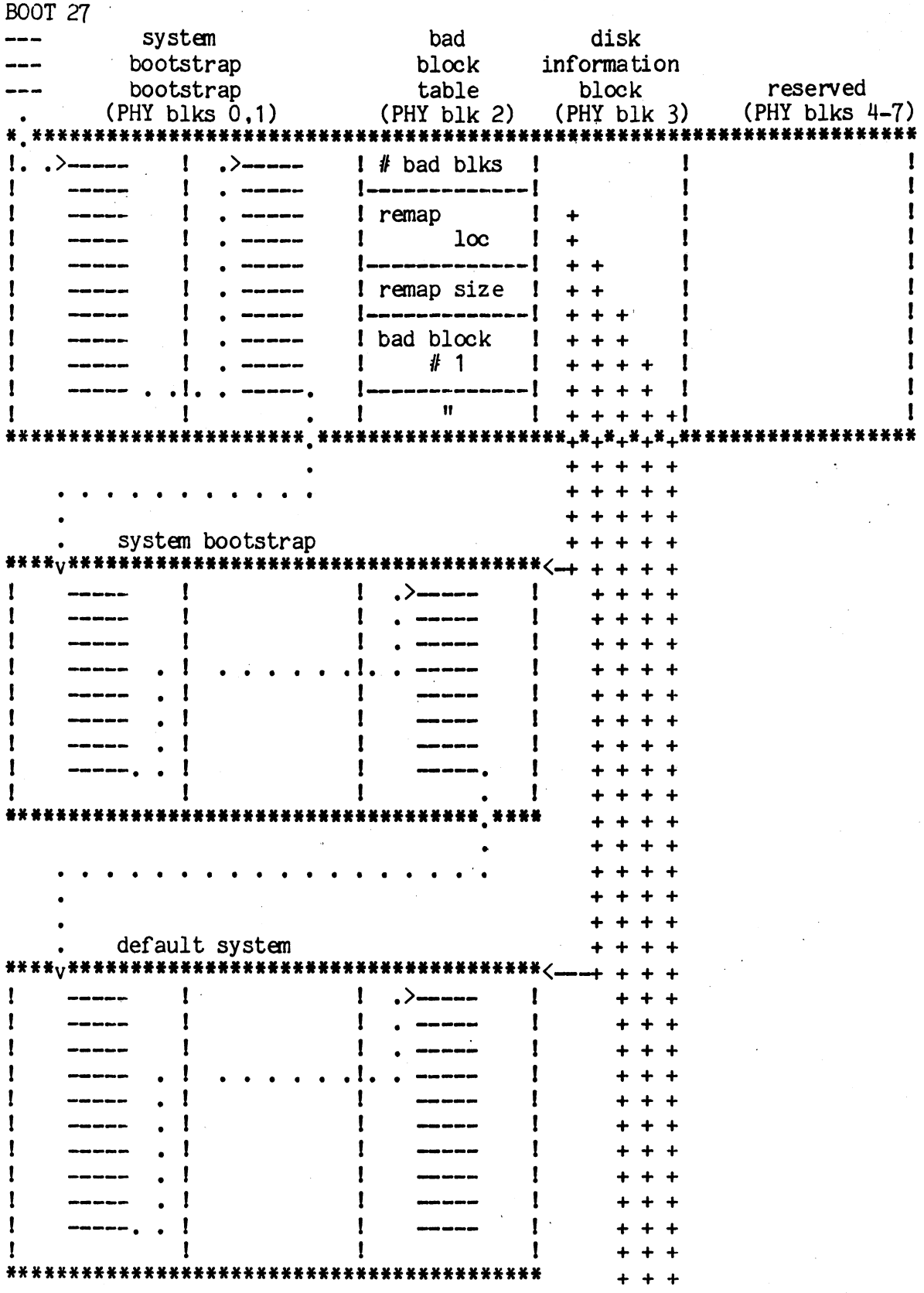
1. Physical blocks 0 and 1 contain the code used to read the system bootstrap program into memory and start it running. Block 0 initially gets into memory via the BOOT command to the SCP-CLI. It also homes the Zebra heads to track zero and clears the controller registers to effectively set up a 'read block 0 into memory starting at location 0' disk transfer. The BOOT command activates a ROM which loads memory with a 32(8) word program and starts the CPU; The program starts the controller (NIOS DEV), whose device code is given in the BOOT command, and hangs in location 377 (contents of 377 = JMP 377). The DMA transfer eventually overwrites location 377 so that the next fetch cycle at 377 results in the first execution of an instruction from block 0. For the purposes of argument, let us assume that this intruction is a 'JMP 0'. Then the net effect of BOOT is as shown in Figure 3, where the BOOT code flows into the first part of the system bootstrap bootstrap. It is an easy matter for these 256(10) instructions to read in block 1, yielding a net 512(10) word program quite capable of reading and starting programs much larger than itself. Unfortunately, it is not sophisticated enough to bring in an entire

operating system; Therefore. it reads and starts an intermediary program (the system bootstrap). The location and size of the system bootstrap as well as the system itself are kept as data in physical block 3, which is further documented below.

2. Physical block # 2 is the bad block table, which contains the disk remapping information necessary to make bad blocks transparent to the file system. One of these tables exists on each PU; hence, from Figure 3, no single disk in an LDU can have more than (256-4)/2 =126 decimal bad blocks. When the LDU is init'ed, the bad block tables of each PU are read into memory so that the disk driver can do fast checking of each logical address involved in I/O.

3. Physical block # 3 is the disk information block, or DIB, which is the most important block in the LDU. Table 1 below shows the structure of the DIB and highlights the locations of disk addresses seen previously in Figure 3. The system bootstrap bootstrap uses the DIB to find out where the system bootstrap is. The system bootstrap uses it to find out where the default system and overlays are. The system uses it to find out where the the disk block usage map is (one bit per logical block -- set => block in use). It also uses it to find out where the first RIB of the root directory is; Since all filenames begin ultimately from the root, this is indeed a crucial step in our objective to relate names to locations. Only the 1st PU is privileged to point to the LDU's bit map, root directory, default system. and system bootstrap.

4. Physical blocks 4-7 are currently unused.



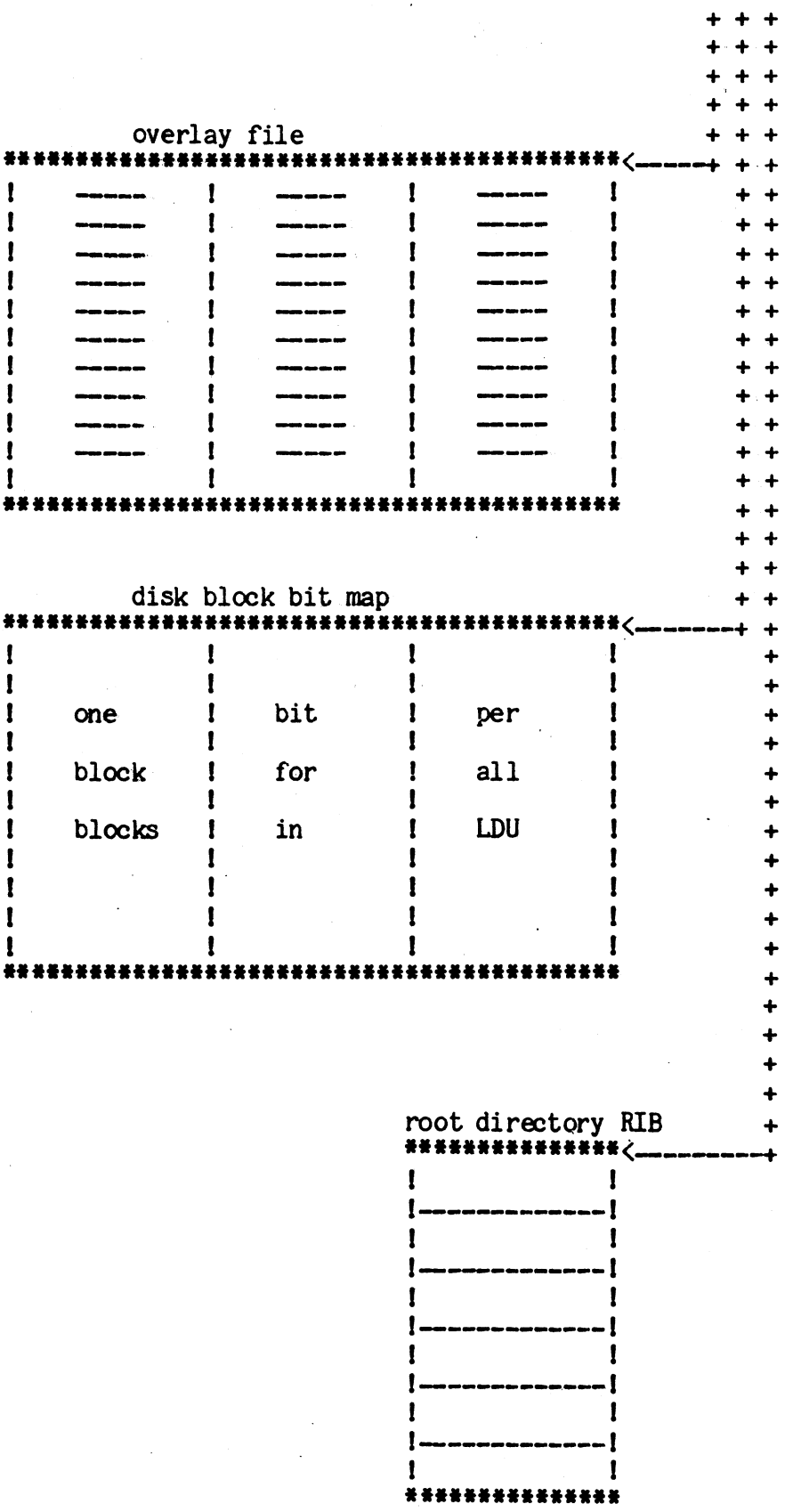


Figure 3. Overhead Structures.

Table 1. DIB Format.

```
        ; disk information block (dib) parameters
000000 .dusr    ibrev=0         ;disk format file sys rev number
000001 .dusr    ibtyp=ibrev+1   ;disk unit type
000002 .dusr    ibsts=ibtyp+1   ;status word(per unit flags???)
000003 .dusr    ibidh=ibsts+1   ;ldu unique id (high)
000004 .dusr    ibidm=ibidh+1   ;ldu unique id (middle)
000005 .dusr    ibidl=ibidm+1   ;ldu unique id (low)
000006 .dusr    ibsnp=ibidl+1   ;sequence number of this PU in ldu
000007 .dusr    ibnpu=ibsnp+1   ;number of PUs in ldu
000010 .dusr    ibnhd=ibnPU+1   ;number of heads
000011 .dusr    ibnst=ibnhd+1   ;number of sectors per track
000012 .dusr    ibncy=ibnst+1   ;number of cylinders
000013 .dusr    ibvis=ibncy+1   ;disk addr start of visible space
000014 .dusr    ibnbh=ibvis+1   ;number of vis disk blocks (high)
000015 .dusr    ibnbl=ibnbh+1   ;number of vis disk blocks (low)
000016 .dusr    ibbth=ibnbl+1   ;PHYs addr of bad block table (hi)
000017 .dusr    ibbtl=ibbth+1   ;PHYs addr of bad block table (lo)
000020 .dusr    ibuid=ibbtl+1   ;10 word unique id for n.c.

    ; the following dib offsets are only valid on unit 1 of the ldu

000032 .dusr    ibldf=ibuid+10. ;ld flags
000033 .dusr    ibnmh=ibldf+1   ;disk address of name block (hi)
000034 .dusr    ibnml=ibnmh+1   ;disk address of name block (lo)
000035 .dusr    ibach=ibnml+1   ;disk address of acl (hi)
000036 .dusr    ibacl=ibach+1   ;disk address of acl (lo)
000037 .dusr    ibbah=ibacl+1   ;disk address of bitmap (hi)
000040 .dusr    ibbal=ibbah+1   ;disk address of bitmap (lo)
000041 .dusr    ibsbh=ibbal+1   ;system bootstrap address (hi)
000042 .dusr    ibsbl=ibsbh+1   ;system bootstrap address (lo)
000043 .dusr    ibssb=ibsbl+1   ;size of system bootstrap
000044 .dusr    iboah=ibssb+1   ;address of overlay area (hi)
000045 .dusr    iboal=iboah+1   ;address of overlay area (lo)
000046 .dusr    iboas=iboal+1   ;size of overlay area
000047 .dusr    ibfbp=iboas+1   ;idp to fib of installed system

        ;funny fib for root directory starts here (fcoml words)
000050          ; file status
000051          ; file type (rh) and format (lh)
000052          ;hash frame size (directories)
000053          ; extension for eof in future
000054          ;       "       "       "
000055          ; last logical byte (eof) (hi)
000056          ; last logical byte (eof) (lo)
000057          ; data element size (hi)
000060          ; data element size (lo)
000061          ; first logical address (hi)
000062          ; first logical address (lo)
000063          ; current index levels (left)
                ; maximum index levels (right)
000064          ; count of inferior directories
        ;funny fib ends here
```

```
000065 .dusr     ibcsh=ibffb+fcoml  ;current size of ld (hi)
000066 .dusr     ibcsl=ibcsh+1      ;current size of ld (lo)
000067 .dusr     ibmsh=ibcsl+1      ;max size of ld (hi)
000070 .dusr     ibmsl=ibmsh+1      ;max size of ld (lo)

000071 .dusr     iblen=ibmsl+1      ;dib fixed length

       ; dib status word mask definitions
       ; valid only on first disk of ld

100000 .dusr     ibsin=  1b0              ;logical disk initialized

040000 .dusr     ibsbi=  1b1              ;sysboot has been installed
         ;(the two status flags appear in word 32)
```

D. Details of invisible space for the other PU's

The other PUs' invisible space appears exactly as the first PU's invisible space, except as just noted in (3) above concerning the DIB's. When all the pointer information is taken away, what remains in the DIB is basicly disk characteristic information. For instance, the unique id and # of PU's in the LDU are inspected during init time to ensure that the complete LDU is present. The sequence #, starting address of visible space, and size of visible space determine which logical address belong to which PU's. The physical characteristics are also given in order to convert physical addresses to sectors, surfaces, and cylinders. See the DIB below (from parfs.sr) for specifics.

E. Directory structures

In part 'D' above, it was shown that information in the DIB points to the first RIB of the root directory file. In the following paragraphs, we discuss how information about files is arranged in the root.

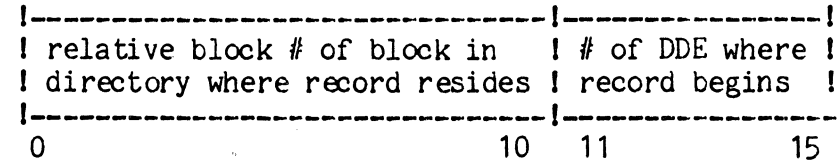
It is important to keep in mind that the root directory (or any directory for that matter) is just an ordinary AOS/VS file from a structural standpoint. What makes the directory file seem different from non-directory files is the not-so-straight-forward organization of its various record types and the prolific use of pointers among them. (The directory file which was just introduced in Figure 3 might lead one to believe that at least all records of a given type are grouped together; However. that diagram's purpose was to reveal how the root directory was related logically to other fundamental disk structures and not how it was internally managed.) The reason for the complex architecture is the tremendous range of storage requirements which might be necessary to describe individual files in a directory. For instance, a file can have a one character filename and zero length ACL or 31 character filename and 511 character ACL. If a single record in the directory were to describe the entire file, it would have to be at least 542 bytes long -- and that doesn't include disk addresses, creation dates, etc! and since filenames and ACL's in general are nowhere near the maximum length, it is easy to see that the one-record-per-file type of directory organization would result in

an unacceptable amount of wasted record space in each descriptor. By breaking the single descriptor into a series of variable length records that point to each other. it is possible to achieve much better space utilization.

The specific record types which describe certain aspects of a file are listed below. To get an exhaustive description of a file requires finding every applicable record (i.e., the FNB and FIB plus the FAC if there is an access control list, the FLB if the file is a link, etc.) and reading them.

type	name	mnemonic	how located
0	free 8w area	n/a	scanning data blocks
1	file name block	FNB	filename match
2	access cont list	FAC	from FIB
3	file info block	FIB	from FNB
4	link block	FLB	from FIB
5	user data area	FUD	from FIB

As in any ordinary file whose records are variable in length. part of the overhead of each record is the record size. In all record types listed above, the word length is kept in the first word of the record, right byte. (The left byte is used to identify the record type.) One departure from the normal overhead of variable length records does exist, however; the records must begin on 8-word boundaries (nuggets). (Each 8-word segment is called a 'directory data entry' or 'DDE'.) The motivation for this rule is the format of the pointers to other records called 'inter-directory pointers' or 'IDPs'. Their format:



Note how this format places an immediate limitation on the size of a directory file; it can only grow to 2**11=2048 relative blocks. If records were kept using the normal AOS/VS byte bookkeeping, then the pointer would require 9 bits to accomodate the byte offset (into a 512=2**9 byte block) and only 7 bits would be left over to describe the record's relative block number. Hence, the directory could only grow to 128 relative blocks!

With the IDP format disclosed, we are now in a good position to actually trace through the directory structures to see how AOS/VS relates a filename to its location and attributes. The discussion refers to Table 2 and Figure 4.

Step 1 -- Use the DIB on PU1 to find the root directory RIB. This was already considered in Figure 3.

Step 2 -- Find the FNB for the file.

A) Hash the filename by taking the sum of the ascii values of all the filename characters and dividing by the frame size for the directory. (For the root, this number is the contents of displacement 52 in the DIB of PU1.) The remainder of this division is the hash value.

B) The hash value is the relative block # in the directory file where
the FNB (hopefully) resides. Therefore, use the hash value as a displacement into the RIB (actually the [hash value]*2 because of the two word addresses) to find the logical address of the relative block.

C) Keeping in mind that the entries are variable length records, scan the FNB's for a match with the desired filename.

D) If the FNB is found, go to 3; otherwise, go to E)

E) Use the forward link to find the next block to scan. Actually, Figure 4 is a little misleading in the fact that it shows the forward link pointing directly at the next block, as though the logical address were kept there. Actually, the link word contains the relative block # of the next block to scan, and the RIB must be used again to find the LOGical address. If there is no link, it can be concluded that the file does not exist. Otherwise, go to c).

Step 3 -- Find the FIB

A) Use the second word in the FNB to find the relative block # of the block containing the file's FIB (bits 0-10 of the IDP) and DDE # (bits 11-15 of the IDP) where the FIB starts in the block.

B) Use the relative block # as a displacement into the RIB to find the logical address of the block.

C) In the block, the FIB begins at offset [DDE #]*10 (octal).

Step 4 -- Find other file information

A) Use the FIB format to find the location of the file and its attributes.

B) Use the IDP's in the FIB to find other records in the directory containing the file's ACL, UDA, etc.

Table 2. Directory file record formats.

FNB -- file name block; to locate, use filename hashing and matching

```
        offset 0     ; left byte = type, right byte = size in words
               1     ; inter-directory pointer to FIB of file
               2     ; filename starts here, one character per byte
               3     ;   left to right packing, no parity
               4
               .
               .
               ?     ; size of record depends upon filename length
                     ;   rounded up to end of DDE.
```

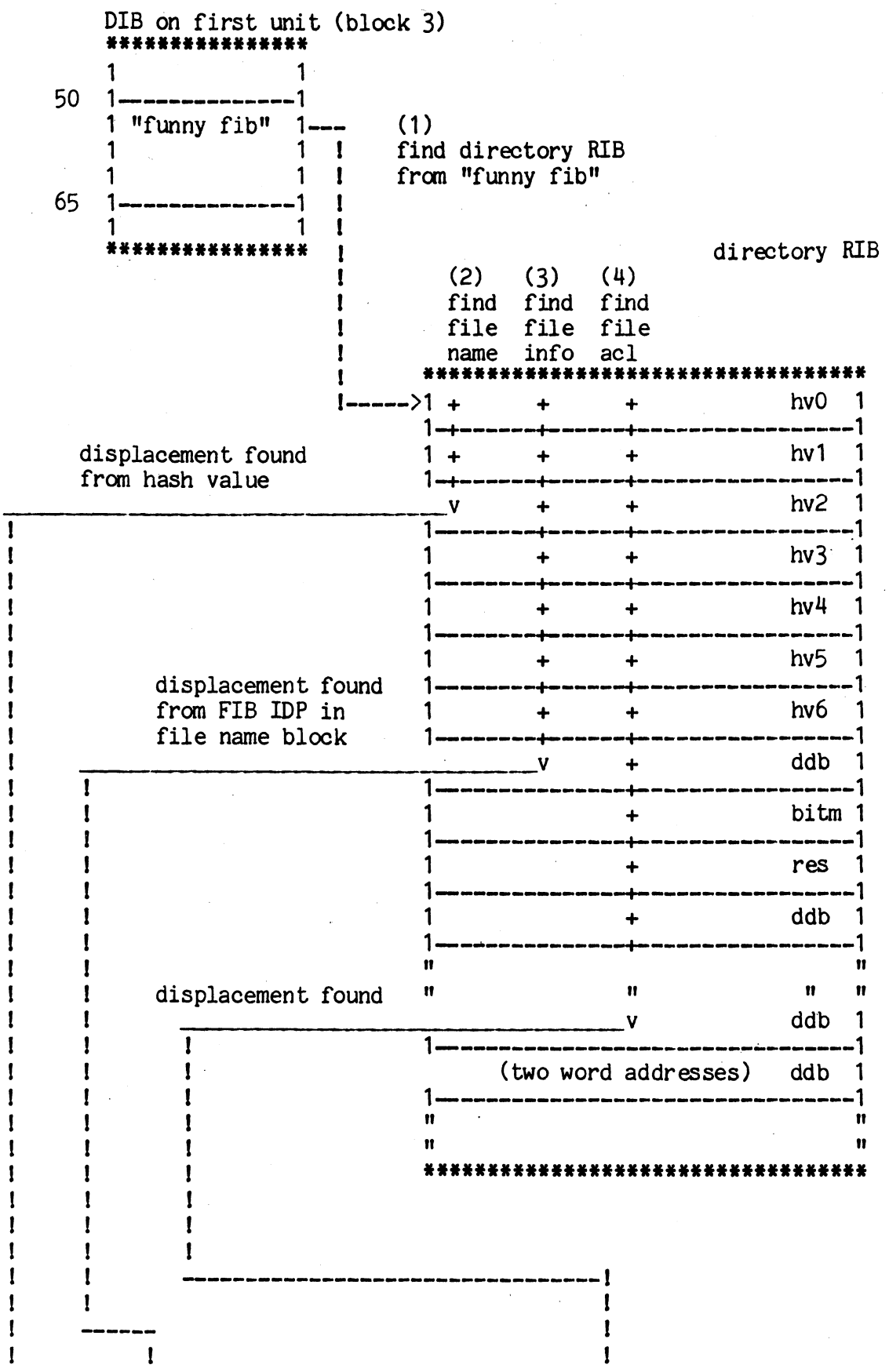
FIB -- file information block; to locate, use offset 1 from FNB

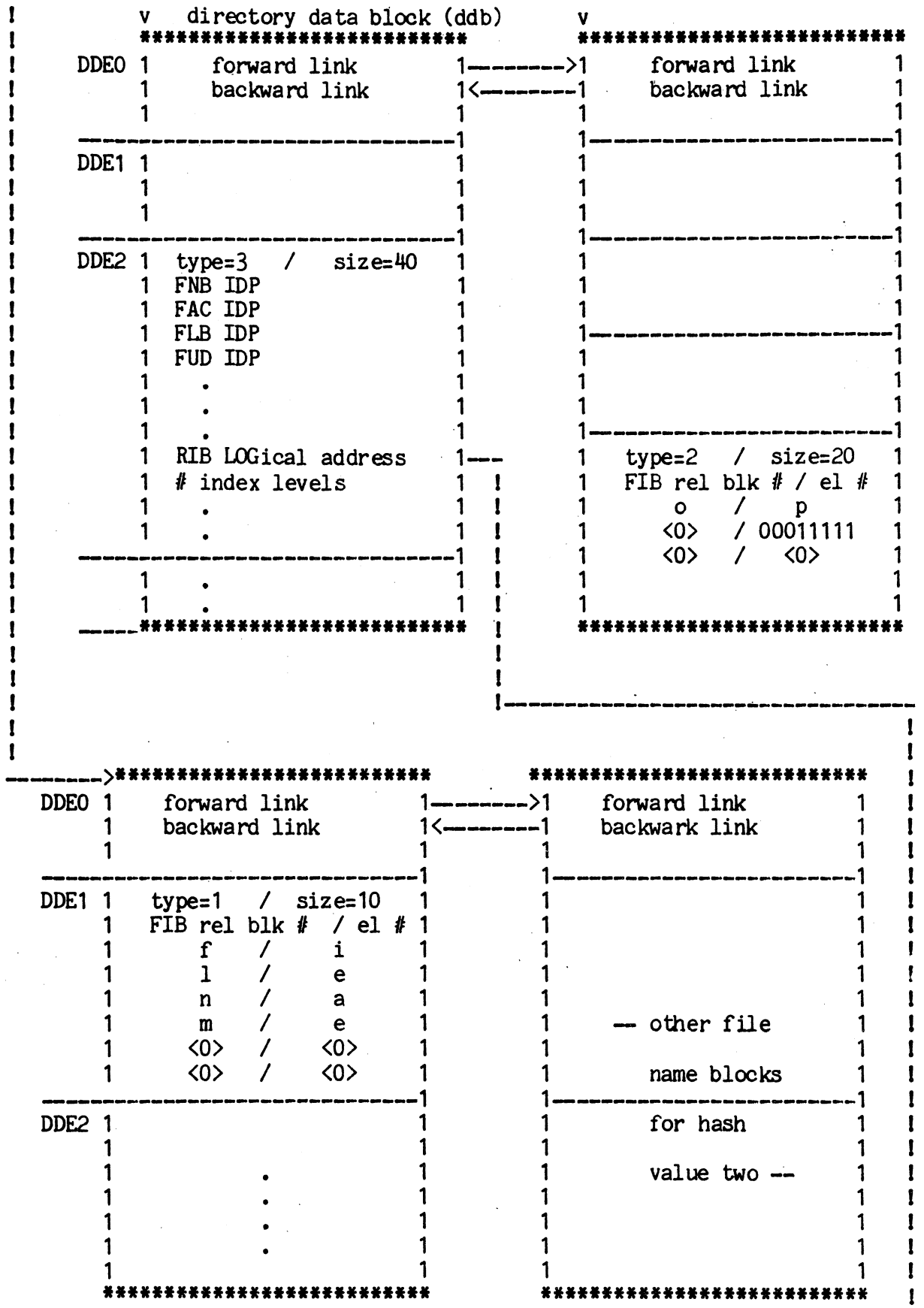
```
        offset 0     ; type of record (left), size in words (right)
               1     ; pointer to first FNB (IDP)
               2     ; pointer to FAC (IDP)
                           * or *
                     ; pointer to FLB (link only) (IDP)
               3     ; unique id  (IDP)
               4     ; file creation time (hi)
               5     ; file creation time (lo)
               6     ; file status -- contains access rights for "+".
                     ; (see FAC below for bit positions)
               7     ; file type (rh) and format (lh)
               10    ; file control parameters
                           * or *
                     ; hash frame size (directories)
                           * or *
                     ; device code (left), unit number (right) for
                     ;     device files only (e.g., grafted LDU's)
                           * or *
                     ; host id - network type files
                     ; FIB extension for data files and directories
               11    ; extension for eof in future
               12    ;       "       "       "
               13    ; last logical byte (eof) (hi)
               14    ; last logical byte (eof) (lo)
               15    ; data element size (hi)
               16    ; data element size (lo)
               17    ; first logical address (hi)
               20    ; first logical address (lo)
               21    ; current index levels (left)
                     ;    maximum index levels (right)
               22    ; count of inferior directories
               23    ; pointer to FUD
               24    ; time last accessed (hi)
               25    ; time last accessed (lo)
               26    ; time last modified (hi)
               27    ; time last modified (lo)
               30    ; extension for FCB address
```

```
            31    ; virtual FCB address or zero
                  ; FIB extension for control point directories
            32    ; current size (high)
            33    ; current size (low)
            34    ; max size (high)
            35    ; max size (low)


FAC -- access control list; to locate, use offset 2 in FIB

        offset 0      ; type of record (left), size in words (right)
               1      ; address of FIB (IDP)
               2      ; ACL -- username (one char/byte) terminated
               .      ;     by null byte, one byte of access,
               .      ;     next username terminated by null,
               .      ;     one byte of access, etc.
               ?      ; end determined by size of ACL (max=256 bytes)

              ;execute: 00000001
              ;read:    00000010
              ;append:  00000100    bit positions of access privileges
              ;write:   00001000
              ;owner:   00010000
```

```
FLB -- link; to locate, use offset 2 in FIB (ACL found from resolution)

        offset 0      ; type of record (left), size in words (right)
               1      ; address of FIB (IDP)
               2      ; link resolution name (could
               .      ;     be another link)
               ?      ; end determined by size of resolution name

FUI -- file unique id; to locate, use offset 3 in FIB (type not used)

        offset 0      ; type of record (left), size in words (right)
               1      ; address of FIB (IDP)
               2      ; unique id
               .
               ?      ; end determined by size of id

FUD -- user data area; to locate, use offset 23 in FIB

        offset 0      ; type of record (left), size in words (right)
               1      ; address of FIB (IDP)
               2      ; FUD forward link
               3      ; FUD backward link
               4      ; user data
               .
               ?      ; end determined by amount of data and link values
```

not used in FIB 4


DIB on first unit (block 3)
"funny fib"
(1)
find directory RIB
from "funny fib"
directory RIB
(2)
find
file
name
(3)
find
file
info
(4)
find
file
acl
hv0
hv1
hv2
hv3
hv4
hv5
hv6
ddb
bitm
res
ddb
ddb
(two word addresses) ddb
displacement found
from hash value
displacement found
from FIB IDP in
file name block
displacement found


directory data block (ddb)
DDE0
forward link
backward link
DDE1
DDE2
type=3 / size=40
FNB IDP
FAC IDP
FLB IDP
FUD IDP
RIB LOGical address
# index levels
type=2 / size=20
FIB rel blk # / el #
o / p
<0> / 00011111
<0> / <0>
DDE0
forward link
backward link
DDE1
type=1 / size=10
FIB rel blk # / el #
f / i
l / e
n / a
m / e
<0> / <0>
<0> / <0>
DDE2
forward link
backwark link
-- other file
name blocks
for hash
value two --

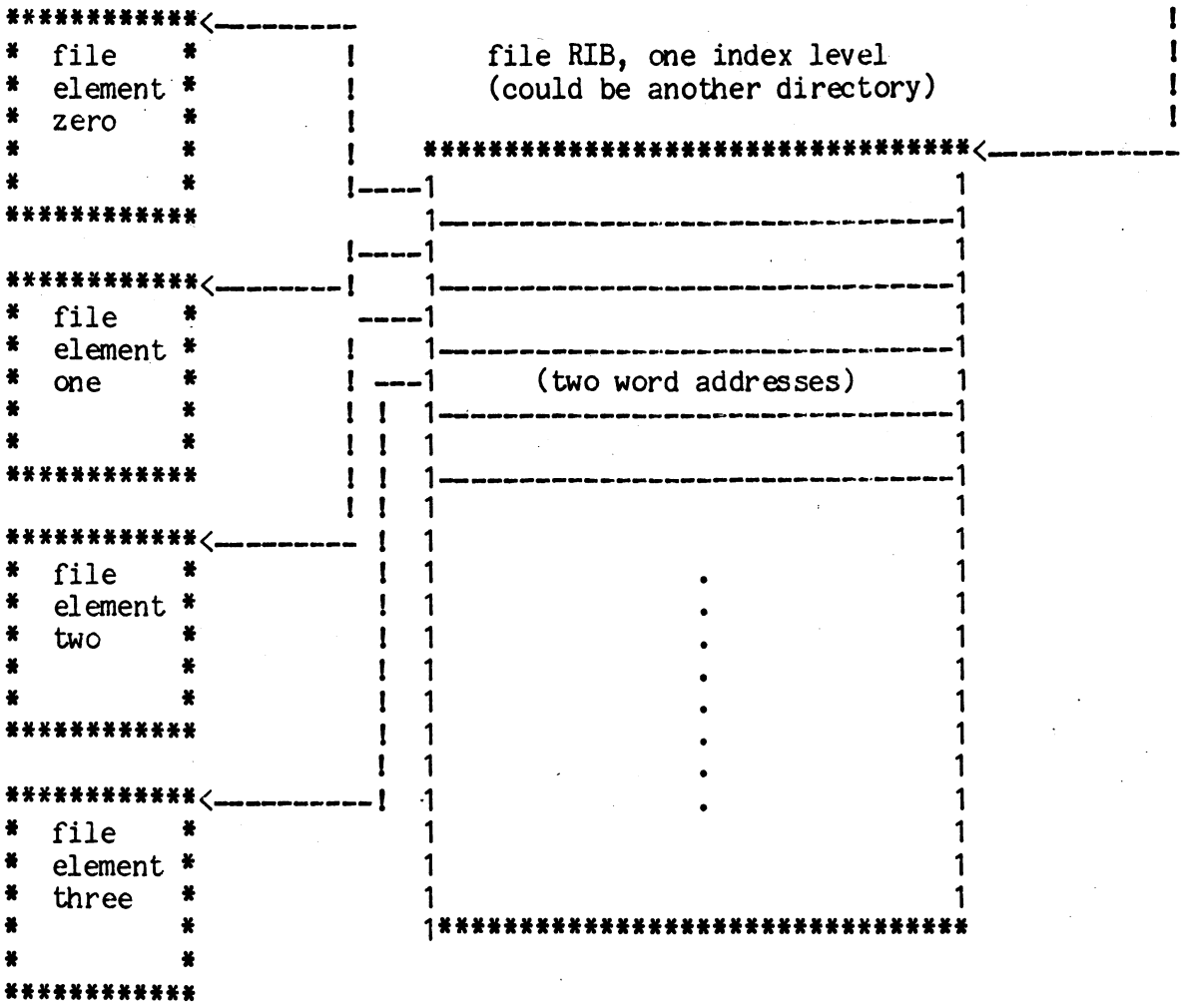


Figure 4. Directory file structure

G. More about the directory file RIB

One aspect of Figure 4 not yet revealed is the significance of the abbreviations in the RIB (ie, HVO, DDB, bitm, etc). The first seven addresses (or more properly. the first 'frame size' addresses as determined at directory create time) are unconditionally set aside to point to the directory data blocks (DDB's) containing the FNB's. When so many FNB's of the same hash value exist that they can no longer fit into one block, AOS/VS lets the file grow in the normal fashion by allocating a free block and putting its logical address in the RIB. The forward link in the filled block is thus established and saved. (Remember, the forward link is the relative block # of the next block.)

Other DDB's contain the other record types.

The entry marked 'res' is not used, but reserved for future use.

Finally. the entry marked 'bitm' is the address of a bit bit map for the directory file DDB's. Note: this bit map is used only for keeping tabs on the usage of the DDB's in the directory file itself and should not be confused with the bit map pointed to in PU1's DIB, which is used to control block usage on the entire LDU.

File System Data Bases in Memory

A. Memory structures involved in user disk I/O

The tables described below are shown in Fig 5. The discussion assumes that a ?RDB is under consideration.

1. CCB's -- channel control blocks

This table (16. words long) describes an individual disk request, such as a relative block number and the number of blocks to transfer. There is a unique CCB for every open channel, and they are kept in GSMEM space. User CCB's are saved on disk as part of a process' swappable context and therefore the space for them is allocated in 256 (10) word chunks. The implication of this is that 17 channels open concurrently is much worse than 16. The maximum amount of memory available to a process for CCB's is 1k (= 64. channels x 16. words). When the process is swapped back in, the CCB's are not brought along immediately, but are instead "faulted" into memory as they are required by the user's system calls. CCB's point to FCB's.

2. FCB's -- file control blocks

This table is 32. words long and essentially holds the FIB of an open file; Hence, it contains the information necessary to convert a file relative block number into a logical disk address (possibly through the reading of several index blocks). The relationship between the CCB and the FCB is another example of AOS/VS' sharing philosophy; when two or more users open channels to the same pathname, only one FCB is needed because its contents are common to all the channels. The CCB's, on the other hand, describe each user's particular requests of the file. Since size information is kept in the FCB, note that users will see the results of file appending done by other users. FCB's point to LCB's.

3. LCB's -- logical control blocks

Logical control blocks are 16. words long, and there is one in GSMEM for each inited logical disk in the system. LCB's contain the info needed to "tie together" a logical disk. Via the LCB, the system can find, among other things, the LDU's bit map, the root directory CCB address, and an ordered list of unit descriptor blocks (see below) that describe each physical unit. Knowing the order of the physical disks that comprise the logical disk is the first step of converting a logical disk address to a sector, surface, cylinder, and drive number.

4. UDB's -- unit descriptor blocks

UDB's are 32 words long. They are used to describe the physical characteristics of a unit in an LDU. The information kept there is essentially that of the DIB of the unit. To convert logical disk addresses into physical adresses local to a particular drive, it is only necessary to move down the LDU's UDB chain, subtracting the total number of blocks on each unit from the logical address desired. (Note that invisible space complicates the algorithm a bit.) When a unit is found whose total exceeds the remainder, the unit's sector, surface, and cylinder info can be used to calculate the controller commands needed to read the block.

5. DCT's -- device control blocks

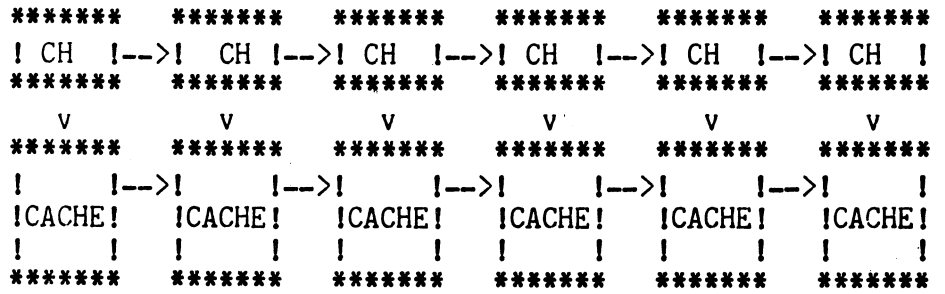
Each UDB is actually a member of two UDB chains. The first was described above in relation to the LDU; the second involves all units on a particular controller and begins in the controller's DCT. The reason two chains are needed is because the units of an LDU do not necessarily coincide exactly with all the units of just one controller. Hence, the system setup of a disk transfer uses the UDB chain starting in the LCB to find information, whereas the interrupt world prefers the chain originating in the DCT.

6. IOCB's -- I/O control blocks

IOCB's are used to hold state information when the processing paths used to drive disks need to pend. Recall that ?RDB is a direct call and therefore that it cannot pend because it has no CB. All that ?RDB does is validate the user's request, initialize an IOCB, and link it into a queue called "iorun". It is the routine pointed to by the IOCB that will start the real disk processing. It will get control the next time the scheduler is entered and finds that the disk manager control block is next to run and, if it is, searches down the queue looking for an IOCB that is ready for service. Hence, system setup of disk I/O is essentially treated as another type of control block. Note that SMON sees a queue that is limited to CB's and PTBL'sand TCB's. However, since ELQUE starts at the first CB on queue (Disk Manager), any entry into the scheduler at a point other than SMON will not involve IOCB processing.

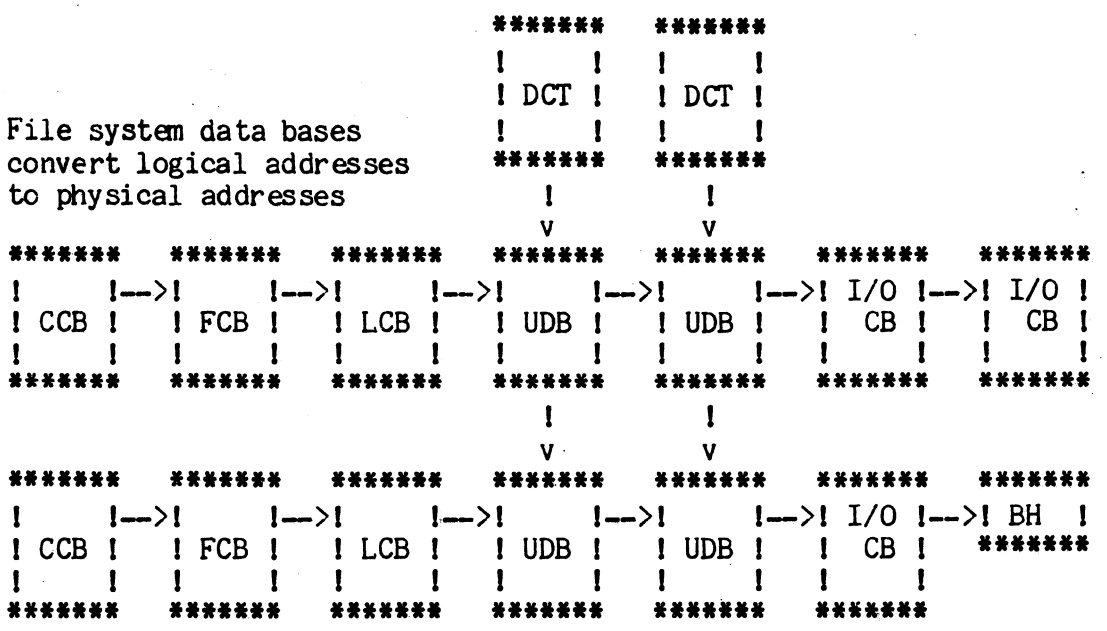
There is a pool of 5 IOCB's for each inited logical disk. If a user does a ?RDB and no IOCB's are available to start the request, the user's CCB is enqueued to another linked list call "CCWQ" where the ?RDB functions will be performed when an IOCB becomes free.

Cache Buffers hold the contents of recently used system buffer info

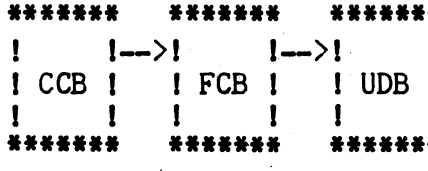


A disk controller might support many drives, not all in the same LDU

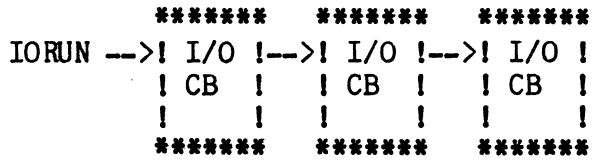
File system data bases convert logical addresses to physical addresses



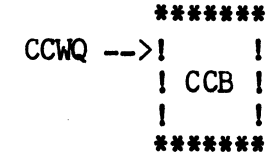
IOCB's describe transfers and point to processing paths



Physical unit opened by user



IORUN is the chain of IOCB's that want service



CCWQ is the chain of backed-up CCB's that ?RDB couldn't find IOCB's for

System buffers are used to read RIB's, directory records, disk bit maps, the IPS
files, Swap, Page, etc.

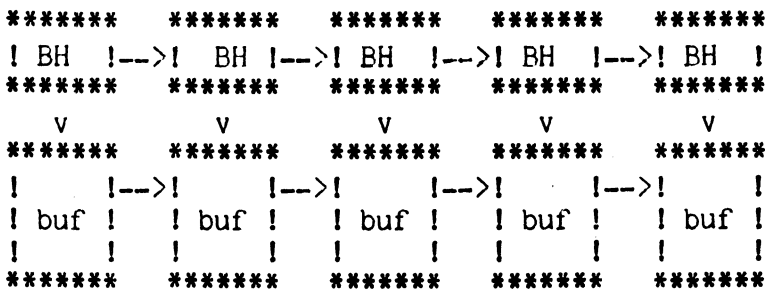


Fig 5. File system data bases.

B) System buffers for system disk requests

System buffers are 256. words long and are associated with a 16. word header. They are taken out of GSMEM space and there are a mimimum of eight of them, although the system will dynamically try to grow the available buffer pool as long as GSMEM is not strained as new disk requests are made. Otherwise, they are replaced on an LRU basis. (See the discussion on CACHE buffers below.)

There are two cases which concern system buffers. The first is a user doing a ?RDB in a file with one to three index levels; all RIB's are read into system buffers. The other involves the system reading its own special files, like active directories, SWAP, PAGE, IPS.XXX, LDU bit maps, etc.

In the first case, the IOCB discussed in section "E" above is not linked onto the UDB. It is marked as not ready, but a buffer header is enqueued instead. The processing entry point in the IOCB is updated to bring control back into the "E" routine when the interrupt world finally reads the index block into the buffer and wakes up the IOCB. If a second level must be read, the same procedure is repeated, and so on until the IOCB itself can be placed on the UDB and the desired block delivered to the user.

In the second case, system requests are handled in much the same manner as user requests. CCB's for system files are filled with relative block information, the FCB-LCB-UDB relationship is exploited to convert relative block numbers to absolute block numbers, and an IOCB is used as a state table queued on IORUN. If index levels must be traversed, buffers are used just as described above.

Because buffer headers and IOCB's can both be queued on a UDB, it should be obvious that many of their displacements have identical interpretations.

[Note: When the user does a ?RDB to a physical unit, there is no IOCB processing at all (because the device is not inited) and in this case, a buffer header is enqueued to the UDB and a status bit indicates that the I/O is intended for user space.]

C. The purpose and use of CACHE buffers

Whenever the system must read a block into a system buffer, it first checks to see if the block is already in a buffer in an unaltered state. If it is, the block needn't be read and a very large amount of seek time is avoided. Thus, it is clear that the more buffers that are available, the faster (in general) the system will respond.

The buffers are used on a least-recently-used basis when no room is available in GSMEM to grow the buffer pool. Under these circumstances, the system suffers when there are a maximum of twenty buffers and it is trying to read twenty-one different blocks (from a bit map, for instance) in a continual loop.

The purpose of CACHE buffers is to save the contents of unused blocks in gvmem so that. later on, they can be copied back into a buffer when the system requests them; a memory-to-memory transfer is much faster than a disk-to-memory transfer. However. not all blocks saved in CACHE will be called back before they are overwritten because of the least-recently-used algorithm. so it is not necessarily true that adding CACHE buffers to the system will speed it up. The addition of CACHE must be weighed against the CPU time introduced to support the feature as well as the loss of total memory available to users.

The memory overhead is easy to calculate. Each CACHE buffer requires 256. words of GSMEM as well as 16. words of header. Thus, a spec for 128 CACHE buffers puts a memory load of 32 + 2 = 34 pages onto the system. If the same system without CACHE under normal operating conditions shows swapped users (excluding those that are waiting for son termination), the introduction of CACHE probably will degrade the system's overall response time.

The CACHE algorithm must be considered, too. When it is determined that a buffer must be involved in I/O, the system first tries to find a buffer with the desired block. Then it looks down the CACHE chain. If the block is not found in CACHE, the least recently used buffer is blm'ed into the least recently used CACHE buffer (unless, of course, free data bases can be found) and the new block read. If the block is found in CACHE, a buffer must be blm'ed up in order to make room for CACHE to be blm'ed down.

It should be pointed out that if too few CACHE buffers are spec'ed, thrashing is invited; users may no longer be swapped out, but CPU time is being wasted in blm'ing blocks up into CACHE without the compensating effect of a good "hit" rate.

To properly choose a CACHE buffer spec, it is necessary to know the "break even" situation and judge whether or not the introduction of CACHE helps or hinders. One statistic -- the percentage hit rate -- is extremely easy to find using the ?LMAP system call. documented in the AOS/VS System programmer's manual. The tougher question is, "what, for my system, is a good hit rate?".

CHAPTER 7 -- EXEC
(AOS/VS Revision 5.00)

AOS/VS EXEC is a program that provides many of the features that our user community associates with the AOS/VS multi-user environment.

EXEC's Major Functions

EXEC's multi-user functional worlds include the following:

(1) Logon World - Logs users on and off of consoles

(2) Queue World - Manages user queues

(3) Batch World - Manages batch streams

(4) Cooperative World - Communicates with cooperative processes (i.e., processes that control printers, plotters, network file transfers, etc.)

(5) Mount World - Manages a tape handling facility that coordinates user tape requests, operator directives, and magnetic tape units.

(6) CONTROL @EXEC (CX) World - Fullfills requests from the operator.

(7) ?EXEC World - Fullfills requests from the user community.

(8) Miscellaneous - Includes a variety of support worlds (delay functions, message dispatcher, memory management, etc.).

Each of these functions will be described in much greater detail as the EXEC chapter progresses.

EXEC's General Purpose

More generally, EXEC's purposes can be described as follows:

(1) Centralizes operator/system manager control over time-sharing, multi-user functions.

(2) Provides the flexibility and "tuning" capabilities necessary to effectively manage the many and varied AOS/VS multi-user environments.

(3) Enforces system security.

(4) Provides monitoring and accounting information.

EXEC Task Structure

AOS/VS EXEC has 14. tasks (refer to the diagram):

* 9 tasks are in ring 7.

* 5 tasks in ring 6.

The tasks are named as follows:

AOS/VS EXEC Tasks

Task Name	Priority	Ring
Initialization Task	200.	7
CONTROL @EXEC Task	200.	7
?EXEC Task	200.	7
Termination Listener	200.	7
Dequeuer Task	200.	7
Cooperative Listener	200.	7
Mount Manager	200.	7
Delay Manager	200.	7
IPC Ignorer	200.	7
Console Driver	3.	6
PMGR Listener	2.	6
SVTA Listener	2.	6
Delay Manager	3.	6
Message Dispatcher	3.	6

EXEC Task Descriptions

Following are brief descriptions of these tasks. More information is provided in the detailed descriptions of EXEC's various functions.

(1) Initialization Task (Ring 7)

* The "initialization task" is the task that EXEC first comes up in.

* This task performs a variety of one-time initialization operations including bringing up all of the other tasks.

* When EXEC initialization is complete, this task suspends itself (?SUS) for the rest of EXEC's life.

[More information on EXEC initialization is provided later in this document.]

(2) CONTROL @EXEC Task (Ring 7)

* The "CONTROL @EXEC Task" (also called the Command Decoder) listens for all commands entered by the operator and performs the requested actions. This task provides the interface through which the operator may control all of EXEC (e.g., enable/disable consoles, control queues, manage tape requests, etc.).

* All "CONTROL @EXEC" commands go to this task (via the IPC ports associated with file :PER:EXEC).

* This task interacts with most of EXEC's data bases and other tasks.

(3) ?EXEC Task (Ring 7)

* The "?EXEC Task" (also called the Request Decoder") listens for all ?EXEC system calls and performs the requested actions.

* This task provides the EXEC interface through which system users can submit queue requests (print, batch, etc.), request tape operations (mount/dismount), and ask for status information (logon status, consolename, operator on/off duty, etc.).

* ?EXEC system call requests come to EXEC as IPC messages via ports associated with file :PER:EXEC_REQUESTS.

(4) Termination Listener (Ring 7)

* The "Termination Listener" listens for all process termination and broken connection messages coming from the operating system to EXEC.

* EXEC receives termination messages for all son processes (consoles, batch jobs, and cooperatives) and any other process to which EXEC is connected (VTA and FTA).

* Console and VTA terminations are reported to the console driver task.

* Batch job terminations are reported to the dequeuer task.

* Cooperative terminations are reported to the operator's console.

(5) Dequeuer Task (Ring 7)

* The "Dequeuer Task" matches queue entries that need servicing with idle batch streams and cooperative processes.

(6) Cooperative Listener (Ring 7)

* The "Cooperative Listener" listens for IPC messages coming to EXEC from cooperative processes (processes that control devices and system resources).

* Cooperative processes include XLPT (line printer), XPLT (plotter), FTA (file transfer), SNA (SNA/RJE emulator), etc.

* Cooperative IPC messages fall into 2 categories:
  - Notifications of a change of status (e.g., job is done).
  - Requests for EXEC to display a message on the operator's console (e.g., "PHYSICAL UNIT FAILURE").

(7) Mount Manager (Ring 7)

* The "Mount Manager" monitors EXEC's tape mount/dismount request data base, searching for entries that need servicing by the operator.

* When a tape request requires attention, the mount manager prompts the operator with a display at the operator's console.

(8) Delay Manager (Ring 7)

* The "Delay Manager" times all ring 7 EXEC operations that cannot be performed until a certain amount of time has elapsed.

* All delay requests come to this task from other tasks.

* When the specified time expires, the delay task processes the delayed request.

* This delay task is most commonly used to delay processing of queue entries (e.g., "QPRINT/AFTER=8:30:00 FILE").

* Note that each ring has its own delay task (i.e., there are two delay tasks, one for ring 7 and one for ring 6).

(9) IPC Ignorer (Ring 7)

* The "IPC Ignorer" receives IPC messages and simply discards them (without interpretation or action).

* All of EXEC's IPC requests to PMGR and SVTA generate responses. If EXEC doesn't care about the response, it specifies that the answer be returned to this task.

* EXEC must receive all incoming IPC messages or they will clog the IPC spool file queue (maintained by the operating system on EXEC's behalf) and cause problems.

(10) Console Driver (Ring 6)

* The "Console Driver" pulls requests for console processing off the console driver queue (internal queue) and performs the specified action.

(11) PMGR Listener (Ring 6)

* The "PMGR Listener" task listens for all console-related responses coming to EXEC from the PMGR.

* The PMGR's messages are in response to EXEC's requests to to assign/deassign a console, get/set the console's characteristics, and perform read/write operations to a console.

(12) SVTA Listener (Ring 6)

* The "SVTA Listener" task listens for all console-related responses coming to EXEC from SVTA.

* SVTA's messages are in response to EXEC's requests to to assign/deassign a console, get/set the console's characteristics, and perform read/write operations to a console.

* All messages to/from SVTA are identical to those to/from the PMGR except that they pertain to VCONS (virtual consoles across the network) instead of local consoles.

* EXEC uses the IPC interface to communicate with SVTA (via the ?ISEND/?IREC system calls).

(13) Delay Manager (Ring 6)

* This "Delay Manager" task times all ring 6 EXEC operations that cannot be performed until a certain amount of time has elapsed.

* All delay requests come to this task from other ring 6 tasks.

* When the specified time expires, the delay task notifies the task that made the original delay request.

* The ring 6 delay task is most commonly used to delay console logon attempts when too many invalide username/password pairs are entered (i.e., "Too many attempts, locking console for 10 seconds").

* Note that each ring has its own delay task (i.e., there are two delay tasks, one for ring 7 and one for ring 6).

(14) Message Dispatcher (Ring 6)

* The "Message Dispatcher" task sends out all console-related messages to the operator.

* Other ring 6 tasks place messages on an internal queue and the message dispatcher sends them to the appropriate pid/port.

EXEC Data Bases

EXEC maintains a variety of data bases necessary for its many functions. The following table lists the major data bases and where they reside.

EXEC Data Bases

Name	Residence
Batch Descriptors (BD)	Ring 7
Logon Descriptors (LD)	Ring 6
Mount Descriptors (MD)	Ring 7
Unit Descriptors (UD)	Ring 7
Virtual Coop Desecriptors (VCD)	File EXEC.COOPERATIVES
In-Core Cooperative Descriptors (CD)	Ring 7
Disk Queue (DQ)	Disk file :QUEUE:Q
In-Core Queue (CQ)	Ring 7
Queue Descriptors	DQ and CQ
Delay Descriptors	Ring 6 and 7
User Profiles	Disk files in :UPD
Operator's Console Descriptor	Ring 7
Pids/Sons Data Bases	Ring 7

[Note that there are a variety of EXEC data bases besides the major ones listed above.]

Following are brief descriptions of these data bases. More information on these are provided in the detailed descriptions of EXEC's various functional worlds.

(1) Batch Descriptors (BD)

* There are 4 batch descriptors, one for each batch stream.

* Batch descriptors are static (assembled into EXEC).

* Batch descriptors are linked together on the "batch stream chain" (BSCHN).

* Batch descriptors include status info., pid, info. on the queue entry being processed, etc.

* Ring 7

(2) Logon Descriptors (LD)

* Each logon descriptor describes one console enabled under EXEC (including local consoles, modems, virtual consoles, etc.).

* LDs are allocated from free memory when a console is enabled.

* The logon descriptors are entered in two tables: one indexed by console control port number, one by local LD numbers assigned by EXEC.

* LDs contain console name, status, pid, username, connect time, etc.

* Ring 6

(3) Mount Descriptors (MD)

* Each MD describes a user's tape mount request.

* MDs are allocated dynamically from free memory when a user mount request comes in. They are deleted when the operator fullfills the corresponding dismount request (generated by the user/agent or when the user terminates).

* MDs are linked together onto the "mount chain" (MDCHN).

* MDs contain username of requestor, volid list, pid, logical tape name, associated UD, etc.

* Ring 7

(4) Unit Descriptors (UD)

* Each UD describes one magnetic tape unit.

* When EXEC comes up, it allocates one UD from free memory for every tape unit genned into the system (i.e., one UD for each file of type "MTU" in directory :PER).

* After initialization, the number of UDs is static (none are added or deleted during EXEC's lifetime).

* UDs are linked together onto the "unit chain" (UDCHN).

* UDs contain unitname, status, associated MD, etc.

* Ring 7

(5) Virtual Cooperative Descriptors (VCD)

* Each cooperative (coop) descriptor describes one cooperative process.

* VCDs reside in disk file EXEC.COOPERATIVES because EXEC does not have enough room internally to store them all.

* Coop descriptors contain cooperative name, IPC ports, pid, status, info. on the queue entry being processed,

(6) In-Core Cooperative Descriptors (CD)

* Each CD is an abbreviated version of the VCD (the coop descriptor that resides on disk).

* The abbreviated CD is used to improve performance by reducing disk accesses when a particular coop is needed (i.e., the in-core coop data base is searched for the appropriate CD and then the corresponding VCD can be retrieved from disk with only one access).

(7) Disk Queue (DQ)

* The disk queue resides in file :QUEUE:Q, which is created by EXEC at initialization time.

* The disk queue contains the queue descriptors (one per queue up to 32.).

* The disk queue also contains all of the queue entries (entries submitted to the queues by users).

(8) In-Core Queue (CQ)

* The in-core queue is an abbreviated version of the disk queue (above) that resides within EXEC.

* The in-core queue is used to improve performance by reducing disk accesses when the queue world must be scanned (e.g., queueing and dequeing jobs).

* Ring 7

(9) Queue Descriptors

* Each queue descriptor describes one of EXEC's queues.

* There are three queues created by EXEC at initialization time that cannot be deleted: BATCH_INPUT, BATCH_OUTPUT, and BATCH_LIST.

* Other queues may be created and deleted dynamically while EXEC is running (up to a maximum of 32).

* The queue descriptors reside in the disk queue; abbreviated versions live in the in-core queue (both disk and in-core queue are described above).

* Queue descriptors contain queuename, queue type, associated coop(s), etc.

(10) Queue Entries

* Each queue entry describes one user queue submit request.

* Queue entries are modified dynamically as users submit requests, and as those entries are processed (maximum of 256 queue entries).

* The queue entries reside in the disk queue; abbreviated versions live in the in-core queue (see above).

(11) Delay Descriptors

* Delay descriptors describe an event that cannot be performed until a certain amount of time has elapsed.

* EXEC maintains two sets of delay descriptors: one for ring 7 and one for ring 6. Each set is completely seperate in contents and organization.

(12) User Profiles

* User profiles reside on disk in directory :UPD (User Profile Directory).

* There is one profile for each user who can logon under EXEC.

* Profiles are created by OP with the preditor utility.

* The profile is used by EXEC to determine who can use the system and how; it contains username/password pair, privileges, initial program and IPC, etc.

(13) Operator's Console Descriptor

* The OP console descriptor contains information about the operator's console (ports, etc.).

* It is used when ring 7 EXEC wishes to send messages to the operator's console. (Ring 6 has a different scheme for sending messages).

* The operator's console is assembled into EXEC.

* Ring 7

(14) Pids/Sons Data Bases

* The Pids/Sons data bases indicate what pids are sons of EXEC.

* "Sons" is a bit map with the bits of EXEC's sons set.

* "Pids" is a table indexed by pid. The table entries contain address of the son's descriptor (e.g., logon, batch, coop descriptor addresses).

* Ring 7

EXEC Initialization

When EXEC first comes up, it performs a variety of one-time initialization actions. This portion of the document lists and describes these actions.

The initialization discussion is divided into two parts:

* Ring 7 Initialization
* Ring 6 Initialization

Ring 7 Initialization

When you PROC EXEC.PR, EXEC first comes up in ring 7. The initial task that gets control is the 'initialization task.' The following list describes what this task does.

[Overlay files XIOV0.SR, XIOV1.SR, and XIOV2.SR contain the ring 7 initialization code.]

(1) Load the ring 6 portion of EXEC.

* The first thing ring 7 EXEC does is load the ring 6 portion of EXEC.

* Ring 6 EXEC resides in file EXECVS.PR and this must be in the same directory as the ring 7 EXEC.PR file (e.g., if EXEC.PR is in :UTIL, EXECVS.PR must also be in :UTIL).

* Ring 7 uses the ?RINGLD system call to load in ring 6.

* If an error occurs when loading ring 6 EXEC, EXEC terminates immediately.

(2) Disable all control character sequences.

* EXEC issues the ?KIOFF system call to disable control character sequences (e.g., ^C^B).

* Since EXEC does not have a process console, it should never get any control character sequences.

* This is a safeguard to make sure a user cannot terminate EXEC via control character sequences in the case where PMGR gets confused and incorrectly assigns EXEC a process console.

(3) Turn on superuser and superprocess.

* EXEC requires both superuser and superprocess to perform its various functions (e.g., access :UPD, :UDD, :QUEUE, etc.).

* If EXEC receives an error on the ?SUSER or ?SUPROC calls, it terminates.

(4) Initialize ring 6 EXEC.

* Now that ring 6 is loaded (see above), ring 7 can LCALL into ring 6 to initialize that ring.

* A later section of this chapter describes ring 6 initialization.

(5) Find out EXEC's username.

* EXEC stores its username for later use.

* Only users with the same username as EXEC can issue CONTROL @EXEC system calls.

(6) Make sure EXEC's initial directory is :PER.

* EXEC creates and access a variety of files in :PER and should have this directory as its initial one.

(7) Make sure that only one EXEC process is running.

* Only one EXEC can operate at a time. If another EXEC is up, terminate.

* To determine if another EXEC is already up, check for the existance of various files (e.g., :PER:EXEC, :PER:EXEC_REQUEST, etc.).

(8) Create IPC ports.

* Create IPC port :PER:EXEC for receiving "CONTROL @EXEC" commands.

* Create IPC port :PER:EXEC_REQUEST for receiving ?EXEC system call requests.

(9) Create file @LMT.

* EXEC creates generic labeled tape file :PER:LMT for later use in implicit tape mount requests.

* Set ACLs on file @LMT to "+,WR".

(10) Make sure user director :UDD exists.

* If :UDD does not exist. create a one as a control point directory (maximum size is -1).

* If creating :UDD, set ACLs to "+,E".

(11) Make sure the :QUEUE directory exists.

* If :QUEUE does not exist, EXEC creates one.

* EXEC sets the ACLs on :QUEUE to "+,AE" so that the CLI and AGENT can create files in here.

(12) Get system revision number.

* Save this value for later use in banners, etc.

(13) Determine directory of the EXEC.PR file.

* This directory name is saved so that EXEC knows where to find the file LOGON.MESSAGE, the cooperative programs (e.g., XLPT.PR), etc.

(14) Re-create file EXEC.COOPERATIVES and allocate one shared page.

* Delete EXEC.COOPERATIVES (if it exists) and recreate it.

* This file contains EXEC's virtual cooperative descriptors (VCD).

* EXEC declares one page of its address space as shared (via ?SSHPT) so that it can later bring in pages from EXEC.COOPERATIVES.

(15) Bring up the other ring 7 tasks.

* Issue ?TASK system calls for the other 8 tasks in ring 7.

(16) Initialize the queue world.

* Create the file :QUEUE:Q, if it doesn't already exist.

* If the Q file already existed, read the file and initialize the in-core queue data base to reflect the disk queue. That is, rebuild the queues and queue entries in-core as they are stored on disk.

* Set ACLs on file :QUEUE:Q to "+,R" (the CLI requires read access to provide the QDISPLAY function).

* Ensure that the three permanent batch queues exist (BATCH_INPUT, BATCH_OUTPUT, BATCH_LIST).

(17) Build the tape unit descriptors (UDs).

* Go through directory :PER looking for files of type MTU (magnetic tape units).

* For each one, create a unit descriptor (UD) and link that descriptor onto the unit chain (UDCHN).

(18) Initialization complete -- suspend operation.

* Now EXEC's initialization is complete (both rings 6 and 7).

* The initialization task is no longer needed and, thus, suspends itself via a ?SUS system call, never to be used again.

Ring 6 Initialization

As we saw above, EXEC first comes up in ring 7 which, in turn, loads ring 6 and then LCALLs in to initialize it. The following list describes ring 6's initialization actions. Note that all these actions are performed in ring 6 by the initialization task (the same task that performs the ring 7 initialization actions described above).

(1) Store initialization values passed from ring 7 to ring 6.

* Ring 6 needs the following values and addresses:

Ring 7 console ?PROC packet address
Ring 7 console ?PROC routine address
Ring 7 SVTA ?CON routine address
Ring 7 address that holds SVTA's pid

We supply more information on these in the section on ring 6 - 7 interactions.

(2) Save EXEC's father's pid and username.

* Used for (a) sending unsolicited EXEC messages to the operator and (b) deciding who can issue CONTROL @EXEC commands.

(3) Initialize the ring 6 delay chain.

(4) Initialize the logon world.

* Allocate the "logon lock" for the logon descriptor data base.

* Initialize the various banners and greeting line text messages.

(5) Initialize the free memory pool.

(6) Create the ring 6 tasks.

* EXEC is now ready to bring up the five ring 6 tasks.

(7) Done with ring 6 initialization.

* After ring 6 is initialized, control returns to ring 7 where ring 7 intialization continues (see above).

EXEC Queues

The purpose of EXEC's queue functionality is to allow many users access to limited system resources in an orderly, flexible manner.

Everyone should be familiar with the CLI's QDISPLAY which shows the current state of EXEC's queue world.

QDISPLAY Showing Queues

```
BATCH_INPUT      BATCH   OPEN
 38070 DA    POLY_60   :UDD:SORBATES:?028.CLI.001.JOB
*38071       U.232     :UDD:HEAVY:MACROS:?015.CLI.004.JOB
 38063 DA    ZIPPY     :UDD:ZIPPY:TACOS:?046.CLI.001.JOB
*38064 D     ZIPPY     :UDD:ZIPPY:DONUTS:?046.CLI.002.JOB

BATCH_OUTPUT     PRINT   OPEN

BATCH_LIST       PRINT   OPEN

LPT              PRINT   OPEN
*38182       HOUDINI   :UDD:HOUDINI:3.00.RN:RELEASE.3.00
 38184 N     JEFF      :UDD:UTILGRP:STRS:NEW:EXEC.C22860
 38185 H     JEFF      :UDD:UTILGRP:STRS:NEW:EXEC.C22861

LPT1             PRINT   OPEN

PLT               PLOT    OPEN

FTQ              FTA     CLOSED

FLAGS EXPLANATION:
H = HELD BY USER
D = /DELETE
N = /NOTIFY
A = UNEXPIRED /AFTER
* = ACTIVE
```

Features

The following list describes the major features of EXEC's queue world:

(1) Efficient sharing of system devices, batch streams, and other system resources.

* EXEC receives many requests from users and spools them to the limited number of cooperatives and batch streams in an orderly, efficient manner.

* Since all jobs come from EXEC, the batch streams and cooperative processes need only know about EXEC and not all the various users on the system.

(2) Flexibility for each user to specify how his/her job is to run.

* By having all queue requests go through EXEC, various features are available that would otherwise be difficult to implement. The following is a partial list (refer to the ?EXEC description in the AOS/VS Programmer's Manual for more information):

- Postpone job processing until a certain time (e.g., QPRINT/AFTER=).

- Prioritize jobs (e.g., QBATCH/QPRIORITY=).

- Do not make a job eligible for processing until explicitly notified (e.g., QPRINT/HOLD).

- Notify the submitter upon job completion (e.g., QBATCH/NOTIFY).

- Specify page limits and CPU time limits (e.g., QPRINT/PAGES=, QBATCH/CPU=).

- Specify whether to restart a job or not if EXEC or the system crashes during the job (e.g., QPRINT/NORESTART).

- Specify options such as number of copies, fold long lines, special printing forms, title lines at the top of each page, etc.

(3) Flexible, centralized system manager/operator control over the batch streams, cooperatives, queues, and queue entries.

* Operator can OPEN, CLOSE, CREATE, DELETE, FLUSH, or obtain the STATUS (SPOOLSTATUS) of any queue.

* Operator can HOLD, UNHOLD, CANCEL, or obtain the STATUS of any entry in a queue.

* Operator can START, STOP, PAUSE, or obtain the STATUS of batch streams and cooperatives at any time.

Queue Data Bases

[The queue data bases are defined in EXEC parameter file XQPARS.SR.]

The File :QUEUE:Q

(1) EXEC makes sure the file :QUEUE:Q exists when it comes up, creating it if it's not there.

(2) EXEC places all information about queues and queue entries in file :QUEUE:Q.

(3) EXEC stores queue information on disk for two reasons:

(a) If EXEC or the system crashes, the information in the queue will not be lost. When EXEC comes up, it will read the file and restore the queues and queue entries.

(b) The CLI can read the queue file directly when a user wants to see a "QDISPLAY" (see earlier figure); the CLI does NOT have to disturb EXEC who has plenty of other, more important things to do.

Format of the :QUEUE:Q File

Queue Descriptors

Each queue descriptors describes one EXEC queues (e.g., LPT. BATCH_INPUT, etc.). The following list describes how these queue descriptors are stored in file :QUEUE:Q and also the contents of these queue descriptors (See diagram).

(1) The first 4 blocks (1 page) of file Q contains the queue descriptors.

(2) Each queue descriptor is 32. words long. Thus, EXEC can support up to 32. queues (i.e., 32. * 32. words = 1024 words = 1 page).

(3) EXEC reads/writes the queue file in 1 page chunks. Thus, EXEC can read/write all the queue descriptors at one time.

(4) Each queue descriptor contains the following information:

* Status flags: Queue is open/closed
  Queue may not be started
  (batch queues cannot be STARTed at a cooperative process)
  Queue may not be deleted
  (batch queues are permanent)

* Queue type: batch, print, plot. fta, sna, hamlet, etc.

* Queue name: BATCH_INPUT, BATCH_OUTPUT, LPT, FTQ, etc.

(5) Each queue has a queue number (an internal number used to identify the queue within EXEC).

* The queue number is the queue offset into the Q file. For example, queue #3 is the third queue descriptor in the file :QUEUE:Q.

Queue Entries

Each queue entry describes one user queue request (i.e., one qsubmit). The following list describes how these queue entries are stored in file :QUEUE:Q and also the contents of these queue entries (See diagram).

(1) The queue entries reside in file :QUEUE:Q after the queue descriptors; that is, after the first 4 blocks.

(2) Each queue entry occupies 1 block (4 per page). EXEC supports up to 256 queue entries.

(3) Each queue entry contains the following information:

* Status flags: Current being processed?
  Looking at this entry? (lock)

* Queue number: Number of the queue that this entry was submitted to.

* Date/time of queue entry submission.

* Limit, if specified (pages, cpu time, etc.).

* User specified qpriority factor.

* Entry Flags:
  User hold
  Operator hold
  Queued by superuser
  Delete after processing
  Don't restart after crash
  This is a restart
  Cancelled by operator
  Cancelled by user
  Binary (print jobs)
  Requires operator on duty
  /AFTER= specified
  Titles option (print jobs)
  Operator flush (batch jobs)
  Fold long lines (print jobs)

* Sequence number.

* Date/time after which job may be processed (valid if after flag is set).

* Pathname of file being submitted to the queue:

  - For batch jobs, this file contains the commands to be executed in the batch stream.

  - For other jobs, this pathname specifies the file to be printed, plotted, transferred, etc.

* Username of process who submitted the queue job.

* Queue specific information (e.g., forms name, destination username, queue output pathname, listfile pathname, etc.).

[The ?EXEC system call description in the AOS/VS Programmer's Manual contains information on many of these values.]

User Data Area (UDA)

A User Data Area (UDA) is a 128-word data area that the system associates with a file but is "invisible" to the user. That is, the UDA will not be seen if the user reads the file in a "normal" manner.

You must use special system call's to create, read, and write to a file's UDA (i.e., ?CRUDA, ?WRUDA).

EXEC creates a UDA for file :QUEUE:Q and uses this UDA as follows:

(1) File :QUEUE:Q has a bit map in the UDA that indicates if a block in the file is valid (in use) or not. If a bit is set. the corresponding queue entry block has valid information in it. If the bit is NOT set. the corresponding queue entry block is not valid.

(2) This bit map helps speed up the CLI's QDISPLAY processing. The CLI first reads in the Q file's UDA bit map. Using this map, the CLI can determine which blocks of the Q file contain valid queue entries. Thus, the CLI can read only those blocks that are necessary for its QDISPLAY.

(3) Note that the first word of the UDA contains a "queue revision" number so that if we ever change the queue format. anyone who reads the file can tell.

In-Core Queue Data Bases

In addition to the disk queue data base (file :QUEUE:Q), EXEC maintains several in-core queue data bases (i.e., data bases that reside within EXEC's logical address space).

The In-Core Queue

(1) The in-core queue data base is an abbreviated version of the queue entry portion of the disk queue. The order of the entries in the in-core queue is identical to the disk queue; however. each entry is much shorter.

(2) By using the in-core queue, EXEC can select an entry without having to search the disk queue (a time consuming process due to disk accesses). After determining which queue entry it wants, EXEC can retrieve the corresponding disk entry with only 1 disk access.

(3) In-core queue descriptors are 11. words in length. They are identical to the disk queue entries EXCEPT for the following:

* In-core queue entries do NOT contain the strings (i.e., pathnames, filename, usernames, etc.).

* In-core queue entries contain a hash values for the username and forms name (if appropriate).

* The in-core queue entries also do not contain some miscellaneous information that resides in the last three "X" words of the user's ?EXEC submit packet (i.e.,words ?XXW1, XXW2, ?XXW3).

Queue Names and Queue Status Tables

EXEC maintains two in-core tables that provide information on the queues:

(a) queue names table

(b) queue status table

The following two sections describe these two tables.

Queue Names Table

EXEC maintains a table of queue names (QNAMS) so it can quickly find out if a queue exists and, if so, what its name and number is.

(1) The queue name table has 32 one-word entries. The table offsets correspond to the queue descriptor offsets in the queue data base. Thus, the third entry in the queue name table corresponds to the third queue descriptor in the :QUEUE:Q disk file.

(2) The table entries contain the following values:

* -1, if there is no queue.

* if not -1, the value is a byte-pointer to the name of the queue corresponding to the table offset (see diagram).

Queue Status Table

EXEC also maintains a table of queue status values so it can quickly determine the status and type of a queue. The format of the table is quite similar to the queue names table.

(1) The queue status table has 32 one-word entries. The table offsets correspond to the queue descriptor offsets in the queue data base. Thus, the third entry in the queue status table corresponds to the third queue descriptor in the :QUEUE:Q disk file.

(2) The table entries contain the following values:

* -1, if there is no queue.

* if not -1, the value indicates (a) the status of the queue, and (b) the queue type (see diagram). Note that these values are the same as those stored in the queue descriptors on disk (file :QUEUE:Q).

Managing EXEC's Queues

Now that we've examined the EXEC queue data bases, we can show how EXEC manages and controls the queues.

[Most of EXEC's queue code resides in modules EXQUE.SR, XCOV6.SR, and XDEQ.SR.]

Creating Queues

To create a queue, the operator issues a command in the following format:

CONTROL @EXEC CREATE <QUEUETYPE> <QUEUENAME>

For example, "CONTROL @EXEC CREATE PRINT LPT".

When EXEC receives a CREATE command, it does the following:

(1) Make sure the <QUEUETYPE> is legal. The legal queue types for the CREATE command are: PRINT, PLOT, HAMLET, SNA, FTA. The user must specify the queue type so EXEC know what kind of queue entries may be placed in the queue.

(2) Validate the queue name (same as filenames).

(3) See if the queue name already exists. Check the QNAMS table.

* If the queue name already exists, return an error.

(4) If the queue name is NOT already in the table, try to find an empty slot for the new queue name.

* If the table is full (i.e., already 32. queues), return an error.

* Else, return the table offset of the first free entry (this value is the queue number for the new queue).

(5) Fill in the appropriate data bases:

* Fill in the name in the queue name table (QNAMS).

* Fill in the status and type in the QSTAT table.

* Fill in the queue descriptor entry in the disk queue (:QUEUE:Q).

Opening/Closing Queues

(1) When a queue is created, it is initially "closed". This means that users may not submit entries to this queue.

(2) To open a queue to the user community, the operator issues the command "CONTROL @EXEC OPEN <QUEUENAME>".

(3) Similarly, the operator may close a queue at any time by issuing the command "CONTROL @EXEC CLOSE <QUEUENAME>".

(4) Opening and closing queues does not affect queue entries that are already in the queue.

(5) To open/close a queue, EXEC does the following:

* Find the queue number (via the QNAMS table).

* Update the open/close flag in the QSTAT table.

* Update the open/close flag in the queue descriptor in the disk file :QUEUE:Q.

Deleting Queues

Since the operator can create queues, s/he may also delete them by issuing the command:

CONTROL @EXEC DELETE <QUEUENAME>

In order to delete a queue, the following must apply:

(1) The queue must, of course, exist (EXEC checks the QNAMS table).

(2) The queue must be closed (check QSTAT table).

(3) The queue must be empty (no queue entries in it).

(4) The queue cannot be currently servicing a cooperative (more on this later).

(5) The queue must be "deletable". There is status flag that indicates whether it is legal to delete the queue or not. (Batch queues BATCH_INPUT, BATCH_OUTPUT, BATCH_LIST are permanent queues and cannot be deleted.)

If the queue fulfills all of these conditions, the queue can be deleted. EXEC does the following:

(1) Clear the queue name entry in the QNAMS table.

(2) Clear the queue status entry in the QSTAT table.

(3) Clear out the queue descriptor in the disk file :QUEUE:Q.

Purging Queues

In some cases, the operator may want to delete all entries in a particular queue. This action is called "purging" and the command that accomplishes this is:

```
CONTROL @EXEC PURGE <QUEUENAME>
```

In order to purge a queue, the following must be true:

(1) The queue must, of course, exist (EXEC checks the QNAMS table).

(2) The queue must be closed (check QSTAT table).

(3) The queue must not be currently servicing a cooperative (i.e., the queue can't be STARTed).

If the queue fulfills the above conditions, EXEC can purge all entries in this queue. To do this, EXEC does the following:

(1) First, EXEC saves the queue's number (determined from examining the QNAMS table).

(2) EXEC then scrolls through the in-core queue data base looking for entries that have the queue's number in them.

(3) For each entry that is in the queue, EXEC marks it "not in use" (i.e., marks the in-core queue entry, clears the corresonding UDA bit in :QUEUE:Q, updates the disk queue entry).

(4) When EXEC reaches the end of the in-core queue data base, all entries have been checked and appropriate ones deleted (marked as "not in use").

?EXEC Queue Calls

As mentioned earlier, EXEC maintains a task that listens for ?EXEC system calls. Basically, this task receives ?EXEC packets, interprets them, and dispatches to the appropriate routine to perform the desired actions. We will go into the details of this task later.

There are several ?EXEC system calls that pertain to the queue world:

(1) ?EXEC Queue Submit Call

Users issue ?EXEC calls to place submit queue entries to EXEC for processing.

(2) Hold/Unhold/Cancel Requests

After submitting a queue entry, users may "hold" that entry (i.e., direct EXEC not to select the job for processing). Similarly, users can "unhold" the entry (make a "held" job eligible for processing).

Also, a user may cancel a queue request if s/he decides that the job should not be processed.

Note that most AOS/VS users do not issue ?EXEC system calls directly. Rather, they issue CLI commands which, in turn, get translated into ?EXEC system calls (e.g., QPRINT, QBATCH, QHOLD, QUNHOLD, QCANCEL, etc.).

[The code that processes these ?EXEC queue requests resides in source modules XROV0.SR, XROV1.SR, and EXQUE.SR.]

Submitting Queue Requests

To submit entries to queues, users issue ?EXEC system calls specifying a queue submit code in the first word of the ?EXEC packet.

The ?EXEC queue submit packet includes the following information (note that some of the values are returned by EXEC):

?EXEC Queue Submit Packet

* Queue type (print. plot. fta. sna, hamlet, etc.).

* Byte-pointer to queue name (e.g., BATCH_INPUT, LPT).

* Date/time enqueued (returned by EXEC)

* Resource limit - The meaning of this value depends on the queue type (e.g., maximum number of pages, maximum CPU time).

* Qpriority for processing the job (0 - 255).

* Flags word:
  - Hold this entry
  - Delete file after processing
  - Don't restart on crash
  - Output in binary mode (print)
  - Requires OP on duty
  - Notify user when completed
  - /AFTER flag
  - Print titles line (print)
  - Fold long lines (print)

* Sequence number (returned by EXEC).

* Byte-pointer to jobname (batch) or forms name (print), if desired.

* Byte-pointer to pathname of file to be processed.

* /AFTER= date and time.

* Values specific to the particular queue type (e.g., qoutput pathname, qlist pathname, number of copies, file transfer destination pathname, etc.).

[Refer to the ?EXEC system call description in the AOS/VS Programmer's Reference Manual for more information.]

When EXEC receives the queue submit, the AGENT has already performed some validation (e.g., validates that byte-pointers and pathnames are legal, etc.).

EXEC performs a variety of other validations, some of which are listed below:

(1) Make sure queue exists.

(2) Make sure the queue name matches the queue type specified in the request (e.g., QPRINT/QUEUE=BATCH_INPUT is illegal).

(3) Make sure queue is OPEN.

(4) The job file must exist and the user must have access to it.

(5) Other validations specific to the queue type, the queue entry values, and the options specified by the user.

Placing The Queue Entry In The Queue

If the user's queue submit packet is valid, EXEC tries to place it in the queue data base.

(1) First, EXEC scans through the in-core queue looking for an empty (unused) queue entry. This is a sequential search from the beginning of the queue.

(2) When an empty slot is found, it is initialized with the appropriate information (some from the user's queue submit packet. some generated by EXEC).

(3) The corresponding disk queue entry is initialized. Since queue entries are 1 block long, this disk action is atomic (i.e., either the new info. is there or the old info is there; it is impossible that only part of the block will be written out).

(4) Lastly, EXEC sets the bit in the disk queue's UDA bit map that corresponds to the new queue entry block (the one just initialized). Since the CLI uses the bit map to determine which blocks contain valid queue entries, we always intialize the block first and set the bit later.

[As a side note, when deleting the queue entry from the queue, EXEC will clear the appropriate UDA bit FIRST, and then update the queue entry disk block. Again, once the UDA bit is cleared, the CLI will not access the corresponding block.]

Holding, Unholding, Cancelling Queue Requests

In addition to the ?EXEC call that places entries into the queue, there are three other calls related to the queue world:

* HOLD - Do not select this queue entry for processing.

* UNHOLD - Cancel a previous HOLD request (i.e., make a queue entry eligible for processing).

* CANCEL - Remove a queue entry from the queue.

The general format for these three system calls is as follows:

* ?EXEC function code (hold, unhold, or cancel)

* Sequence number or jobname

When EXEC receives a hold/unhold request, it does the following

Hold/Unhold Request

(1) Finds the corresponding queue entry (in-core first, then on disk).

(2) Sets/Clears the "hold" bit in the queue entry.

(3) When a job has the "hold" bit set, the job will not be selected for processing.

When EXEC receives a cancel request, it does the following:

Cancel Request

(1) Finds the corresponding queue entry (in-core first, then on disk).

(2) If the queue entry is active (being processed), EXEC aborts the job.

(3) If the queue entry is NOT active, EXEC simply sets a bit in the queue entry (in-core and disk).

(4) Later, during job selection, EXEC will see the cancel bit and perform the appropriate action. (Usually, EXEC or the cooperative will simply write "CANCELLED BY USER" wherever the job's output would normally go.)

As mentioned earlier. AOS/VS users do not usually issue ?EXEC calls directly. Instead, they issue CLI commands that are translated into ?EXEC calls (e.g., QHOLD, QUNHOLD, QCANCEL).

[Refer to the ?EXEC system call description in the AOS/VS Programmer's Manual for more information.]

Batch Processing

EXEC's batch functionality allows you to run a job without using your console. Thus, by using batch, you can have one or more programs running and still have access to your console for interactive processing.

The following list provides some general characteristics of EXEC's batch world:

(1) Programs that run in batch should not require a console for execution.

(2) One batch process can run in each "batch stream" (EXEC maintains four of them).

(3) Since batch jobs are ?PROCd as a direct son of EXEC (instead of the user's son), the user's console is not tied up or affected in any way during batch execution.

Batch Data Bases

(1) EXEC maintains 4 batch streams.

(2) There is a "batch descriptor" for each stream and they are linked together in a list (the batch chain).

(3) Currently. the 4 batch descriptors are static. They are assembled into EXEC and cannot be either created or deleted during EXEC's lifetime.

(4) The corresponding batch queues BATCH_INPUT, BATCH_OUTPUT, BATCH_LIST are also static. That is, they always exist.

(5) The format of a batch descriptor is as follows:

* Status flags:
  - Batch @OUTPUT opened
  - Termination in progress
  - Paused at end of job
  - Idle (nothing to do)
  - Verbose messages enabled
  - Silence mode enabled
  - Limiting enabled
* Process type and priority
* Username
* ?PROC output name
* ?PROC list name

* Stream name (e.g., "STREAM_2")

* Queue entry select packet: contains information about the job that is currently running in the batch stream.

* Highest/lowest qpriority that may run in this batch stream.

* Highest acceptable CPU time limit for this stream (used if limiting enabled).

Managing Batch Streams

Managing batch streams is somewhat less complicated than managing cooperatives.

(1) Batch queues are static. That is, the queues BATCH_INPUT, BATCH_OUTPUT, and BATCH_LIST are created by EXEC and exist for the life of EXEC.

(2) The batch stream <--> queue associations are simpler: there are always 4 batch streams and the jobs are always selected from the BATCH_INPUT queue. This cannot be modified.

This means that there is no need for a queue bit map in the batch descriptors.

(3) EXEC does not need to use IPCs to communicate with batch streams. Instead, EXEC does the following:

Batch Manipulation

Action	How It Happens
Start this job	EXEC ?PROCs a job in a batch stream.
Job is done	EXEC receives a process termination from AOS/VS.
Flush the current job	EXEC simply terminates the process running in the batch stream.
Get a stream's status	EXEC can look directly at the batch descriptor data base and determine the status.

(4) When EXEC ?PROCs a user's batch job, it reads the user's profile and uses the privileges and parameters specified

there.

The user must have "batch privilege" to run jobs in the EXEC batch streams.

Spooling Queue Entries

Up to now, we have described:

* The format of the various queue data bases.
* How op manages the queues.
* How users place entries in the queues.
* How coops are associated with queues.
* How EXEC and coops communicate.
* How batch streams operate.

To complete our coverage of EXEC's QUEUE/COOP/BATCH worlds, we must consider how EXEC selects queue entries for the various cooperatives and batch streams.

In EXEC, "spooling" is the act of selecting a particular entry from the queue for running on an idle cooperative or batch stream. This is also referred to as "dequeueing".

[The code that supports the functionality described in this section resides in file XDEQ.SR.]

The Dequeuer Task

EXEC maintains a task whose sole purpose is to find jobs for coops and batch streams to run.

The dequeuer task is normally "asleep". It does not search for jobs unless it is "poked" (woken up) by an inter-task message from another EXEC task.

The following actions will "poke" the dequeuer task:

Poking the Dequeuer

(1) CONTROL @EXEC START; CONTROL @EXEC CONTINUE

A new cooperative may have been started up or a coop/batch stream has been readied.

(2) BATCH TERMINATION

A batch stream has terminated and is now ready to accept another job.

(3) JOB DONE MESSAGE FROM COOP

A cooperative has finished a job and is now ready to accept another one.

(4) ?EXEC QUEUE SUBMISSION

A user request has been queued up. This request may be able to run at an idle coop/batch stream.

(5) QUEUE DELAY EXPIRATION

A user submitted a job with the /AFTER= option. The specified time has elapsed and the job may now be spooled to a coop or batch stream.

Dequeuer Action

When the dequeuer task gets poked, it does the following:

Dequeuer Action

(1) The dequeuer task searches all batch and coop descriptors looking for idle ones.

(2) When the dequeuer task finds an idle stream/coop, it searches the queue to find the "best" job that is eligible to run on that stream/coop.

(3) If the dequeuer task finds an eligible job for the stream coop, it starts that job up and marks the stream/coop busy (not idle) and the queue entry active.

As we've seen, to start coop and batch jobs, EXEC does the following:

Cooperatives - Send a "start this job" IPC message to the cooperative.

Batch streams - Issue a ?PROC system call to bring up the user's batch process.

(4) After checking all coops and batch streams, the dequeuer goes back to sleep and waits to be poked again.

Note that the dequeuer performs the same action no matter why it got poked. That is, whenever it's poked, the dequeuer searches ALL batch streams and cooperatives, not just the one that caused it to wake up.

Selecting a Queue Entry

When the dequeuer task finds an idle cooperative or batch stream, it calls a routine that searches the in-core queue looking for a job to run at the coop/stream.

The queue selection routine does the following:

Queue Selection Routine

(1) Initialize the "best queue entry" variable to 0 (i.e., no best queue entry yet).

(2) Look at a queue entry and see if it fulfills the following criteria:

* The queue entry cannot be waiting for a /AFTER= to expire.

* The queue entry type and the coop/batch type must match (e.g., only spool print queue entries can go to print coops).

* If coop/batch limiting is enabled, the queue entry's limit must be less than or equal to the coop's/stream's limit.

* The queue entry's qpriority musts fall within the range specified for the coop/stream.

OP uses the QPRIORITY command to specify a valid range for coops/streams. Users specify qpriorities for their queue submissions with the /QPRIORITY= switch.

* For print type coops, the form specified in the queue entry must match the form enabled at the printer.

(3) If a queue entry does NOT fulfill all of the above, the dequeuer gets the next queue entry and checks these criteria again (i.e., go to step #2).

(4) If the queue entry DOES fulfill these criteria, the dequeuer calls a "queue entry compare" routine that compares the current "best queue entry" (best so far) with the new one.

* If the new queue entry is better suited for the coop/stream, the compare routine places the new queue entry in the "best queue entry" variable.

* If the old queue entry (the one in the "best queue entry" variable) is deemed better than the new one, then the "best queue entry" variable is left as is.

[We provide more information on the queue entry compare routine below.]

(5) At this point, the selection routine checks to see if the new "best queue entry" is a cancelled job (i.e., the job has been cancelled by OP or user). If so, EXEC will run this job immediately (go to step #7 below).

(6) If the best job is NOT cancelled, the selection routine gets the next queue entry and continues with step #2 above.

(7) When the selection routine cannot find any more entries in the queue, it spools the queue entry stored in the "best queue entry" variable to the idle coop/stream. (That is, the queue entry is marked "active" and the coop/stream is marked "busy".)

(If the "best queue entry" variable is still 0 at the end of the selection routine, the coop/stream remains in the idle state.)

Comparing Queue Entries

As mentioned above, the selection routine finds an eligible queue entry and calls another routine that decides if the new queue entry is better suited for the coop/stream than the previous best.

In the following tables, "old entry" refers to the previous best queue entry; "new entry" refers to the one that is being compared to the previous best.

First, the compare routine checks the new queue entry for the following:

Examine New Queue Entry

Variable	Action
Cancelled by OP	New entry is better
Cancelled by user	New entry is better
Entry on hold	Old entry is better
Entry requires operator on duty and s/he is not	Old entry is better
New job is a restart (i.e., EXEC or AOS/VS crashed while job was active)	New entry is better

If the above comparisons did not determine whether the new entry was better than the old one, the two entries are compared head to head as follows:

Head to Head Comparison

Variable	Action
New entry's qpriority is higher than old entry's	New entry is better
New entry submitted earlier than old entry	New entry is better
New entry's sequence number is lower than old entry's	New entry is better

If the new entry is determined to be better, then the new entry is placed in the "best queue entry" variable.

If the new entry is NOT better, then the "best queue entry" variable will not be modified.

Mount World

EXEC's mount world functionality provides system users with a way to ask the operator to perform tape operations. At the same time, the operator can control access to all tape units in a flexible, organized manner.

Mount World Features

(1) Users do not have to know what unit their tape is on.

* User's can issue dump/load commands supplying a logical name instead of a unit name.

* When the operator mounts the user's tape, EXEC creates a link from the user's logical name (e.g., "TAPE") to an actual unit name (e.g., "@MTB0").

* When the user accesses the tape, s/he can use the logical name and does not need to know the actual tape name (e.g., "TAPE:0" resolves to "@MTB0:0").

(2) The operator has centralized control over tape requests and units.

* OP can accept or refuse any user tape requests.

* OP can assign tape units to requests in any order.

* OP can direct EXEC not to accept any more tape requests (via the "CONTROL @EXEC OPERATOR OFF" command).

(3) Request and unit book keeping.

* EXEC keeps track of what tapes are on what drives

* EXEC also keeps track of which user requests are on which tape units.

* In the case of labelled tapes, EXEC keeps track of what volids are on which drives and which volids the user intends to access.

* OP can get this information with the commands "UNITSTATUS" and "MOUNTSTATUS".

(4) EXEC logs tape use to the system log (i.e., file :SYSLOG).

* This is useful for monitoring and billing purposes.

* The magnetic tape unit log entry contains the following:

  - Username
  - Tape unit name
  - Current time
  - Amount of time unit was in use

(5) There is no internal limit to the number of user mount requests or tape units that EXEC can support.

Data Bases

EXEC's tape handling facility works with 2 data bases:

(1) Unit Descriptors (UD)

(2) Mount Descriptors (MD)

Unit Descriptors (UD)

(1) When EXEC comes up, it searches the :PER directory for all magnetic tape units (i.e., files of type "MTU").

(2) EXEC creates a "unit descriptor" for each one and links them together onto the "unit chain" (UDCHN).

(3) The unit chain is static; that is, after initialization, EXEC never deletes or adds unit descriptors to the chain. Since the number of tape units is genned into the system and is, thus, static, EXEC's scheme is perfectly valid.

(4) Each unit descriptor contains the following:

* Link to the next unit descriptor.

* Date/time unit was connected to user's request (used for billing).

* Link to associated mount descriptor.

* Link to other units associated with the same mount descriptors.

* Flags:

  - Drive is (pre)mounted
  - Drive is currently open
  - User has been billed for this drive

* Unit name (e.g., "@MTB0").

* Username of user whose tape is on this unit.

* Volid on this unit.

(5) The operator may issue the "CONTROL @EXEC UNITSTATUS" command to determine what drives exist on the system. Unitstatus will also return the status of the various units (i.e., whether it is being used, username, volid, etc.).

Mount Descriptors (MD)

(1) When a user wishes to access a tape, s/he issues a mount request:

* ?EXEC system call with a mount code in it.

* CLI "MOUNT" command (which resolves to a ?EXEC call).

[Refer to the ?EXEC system call description in the AOS/VS Programmer's Manual for information on the format of this call.]

(2) When EXEC receives a mount request from the user. it does the following:

(a) Checks to see if the operator is "ON DUTY" (OP can control this with the "CONTROL @EXEC OPERATOR" command).

If OP is "OFF DUTY", the user will receive an error.

(b) EXEC then checks to make sure that the requestor is a son of EXEC (or grandson, etc.). The user must be a son of EXEC to use the tape facility so that if s/he terms, EXEC will get a termination message and can delete the user's mount request.

If a user is not a son of EXEC, s/he will receive an error.

(c) If the operator is on duty and the user is a son of EXEC, EXEC will create a "mount descriptor" (MD) for the user's request.

(d) The mount descriptor is allocated dynamically from EXEC's free memory pool.

(e) Mount descriptors are linked together on the mount descriptor chain (MDCHN).

(3) Each mount descriptor contains the following:

* Link to the next mount descriptor.

* Flag word (mostly used to communicate with AGENT):

  - Volid not verified
  - IBM format
  - Tape density
  - Read only
  - OK to extend volid list
  - First/specific volid in list

* Flag word (for EXEC's internal use):

  - Mounted explicitly
  - Logical name supplied
  - Return status word
  - Return unitname
  - Return volid
  - Dismount text present
  - First volid
  - Requestor has logged off
  - Dismount me
  - Fileset (MD) is open

* Action bits (if any are set, the MD requires action by the operator):

  - Mount error
  - Mount next volume
  - Mount specific volume
  - Mount in progress
  - Mount request outstanding
  - Request to extend volid list

* Associated unit descriptor chain (points to the unit(s) currently associated with this MD).

* Associated user descriptor (points to a logon descriptor or batch descriptor).

* Logical tape name (as supplied in the user's request).

* Volid list - list of ordered volids required by the user (labelled tape only).

* Pointer to current volid - points into volid list (labelled tape).

* Requestor's text (as supplied by the user).

* Requestor's pid.

* Pid of EXEC's immediate son.

* Unique ID for this mount descriptor (MID).

* Username associated with this MD.

(4) The operator may issue a "CONTROL @EXEC MOUNSTATUS" command to determine if there are any mount requests and, if so, what their status is.

(5) In addition, EXEC will prompt the operator if there are any outstanding mount requests (i.e., mount requests that require action).

Associating Units With Mount Requests

(1) When a mount request comes in from the user, EXEC will prompt the operator to either:

* Refuse the request, in which case EXEC will return the error "REFUSED BY OPERATOR" and delete the mount descriptor; or

* Fulfill the request by assigning a unit to the user and mounting the user's tape on that unit.

(2) Again, the operator may issue the "UNITSTATUS" command to determine which drives are available.

(3) The operator assigns a drive to the mount request by issuing a command such as "CONTROL @EXEC MOUNTED <UNITNAME>".

(4) When OP issues a "MOUNTED" command, EXEC does the following:

(a) Makes sure that the unit is not already assigned.

(b) Sets the ACLs on the drive to "<USER>,OWARE" so that no one else can access the drive.

(c) Places the date/time in the unit descriptor for later billing purposes.

(d) Links the mount descriptor to the unit descriptor.

(e) Marks the unit descriptor as "mounted" (i.e., in use).

(f) If the request is for a labelled tape, EXEC will place the volid of the tape in the unit descriptor.

(5) Once the user's mount request is associated with a tape unit, the user may access the tape as desired. That is, the user can LOAD, DUMP, COPY, etc. to the tape that OP has mounted.

(6) When the user is finished with the tape, s/he issues a dismount request (either a ?EXEC dismount call or a CLI DISMOUNT command).

(7) When EXEC receives the dismount request, it does the following:

(a) Sets the ACLs on the tape drive to null so that no one can write on the tape.

(b) Places a tape usage entry in the system log (file :SYSLOG).

(c) Prompts the operator to dismount the tape.

(8) When OP removes the tape from the drive, s/he notifies EXEC by issuing the command "CONTROL @EXEC DISMOUNTED". At this point, EXEC does the following:

(a) The user's mount descriptor is removed from the mount descriptor chain and freed up (i.e., the memory is released back to EXEC's free memory pool).

(b) EXEC marks the unit descriptor as available (i.e., not in use).

Tape Mount Example

The following section traces through a typical mount session between the operator and the user. This example takes place on a secure system where users do not have access to the tape drives and must, therefore, ask OP to mount tapes for them.

(1) Suppose that user "ZIPPY" wishes to look at a tape. He would issue the CLI command:

```
) MOUNT TAPE I JUST GAVE YOU
```

In this request:

"TAPE" is the logical name that the user will use to refer to the tape.

"I JUST GAVE YOU" is a text string that EXEC will relay to the operator; it may contain anything the user wishes to tell OP.

(2) After ZIPPY issues the MOUNT command, his CLI process hangs until OP responds with a "MOUNTED" or "REFUSED" command

(a) The user's CLI process will interpret the MOUNT command and turn that into a ?EXEC system call (with the appropriate mount function code).

(b) The AGENT, in turn, changes the CLI'S ?EXEC packet into an IPC and sends it to EXEC'S "EXEC_REQUEST" port. The AGENT pends waiting for a response to the IPC (i.e., the AGENT issues an ?IS.R) and, thus, the user's MOUNT also pends. (We describe ?EXEC calls elsewhere.)

(3) When EXEC receives the MOUNT request, it creates a mount descriptor for ZIPPY and places it at the end of the mount descriptor chain (MDCHN).

(4) After EXEC creates the mount descriptor, it will prompt the operator with the following message:

```
** UNIT MOUNT **

MID=2, USER=ZIPPY, PID=26, EXEC SUB-TREE PID=26

REQUEST IS 'I JUST GAVE YOU'

UNIT(S) ARE: NONE

RESPOND:  CONTROL @EXEC MOUNTED @UNITNAME

OR:       CONTROL @EXEC REFUSED
```

(5) Assuming that the operator wants to fulfill ZIPPY's request, s/he may issue a "UNITSTATUS" command to see what tape units are available. The UNITSTATUS display might look like this:

```
@MTB0 NOT MOUNTED

@MTB1 NOT MOUNTED

@MTB2 NOT MOUNTED
```

In this case, the system has 3 tape drives and none of them are currently in use.

(6) OP decides to assign ZIPPY's request to tape unit @MTB0. S/he gets the appropriate tape (in this case, the one specified in ZIPPY's text) and mounts it on unit @MTB0.

After OP mounts the tape, s/he issues the command

```
*) CONTROL @EXEC MOUNTED @MTB0
```

(7) EXEC now links the unit descriptor for @MTB0 with the mount descriptor (MID 2) and sends an answer back to the user (so that ZIPPY'S CLI wakes up again). (See diagram).

(8) Just to make sure everything worked OK, OP issues MOUNTSTATUS and UNITSTATUS commands:

```
*) CONTROL @EXEC UNITSTATUS

@MTB0  MID=2, USER=ZIPPY, PID=26

@MTB1  NOT MOUNTED

@MTB2  NOT MOUNTED

*) CONTROL @EXEC MOUNTSTATUS

      ** UNIT MOUNT **

MID=2, USER=ZIPPY, PID=26, EXEC SUB-TREE PID=26

REQUEST IS 'I JUST GAVE YOU'

UNIT(S) ARE: @MTB0

*) ACL/V @MTB0

@MTB0   ZIPPY,OWARE
```

(9) Meanwhile, ZIPPY's MOUNT request has come home and he can again issue CLI commands.

First, ZIPPY Checks to see if EXEC correctly created the tape link from the logical name "TAPE" (supplied in ZIPPY's MOUNT request) to the tape unitname (supplied by OP).

```
) F/AS/SO TAPE

DIRECTORY :UDD:ZIPPY

TAPE     LNK @MTB0
```

ZIPPY is convinced that EXEC has done its job so he goes ahead and accesses the tape. He can use the tape link name for convenience.

```
) LOAD/V TAPE:0
12-MAY-83  14:00:03
```

```
        MAINPROG.PL1
        MODULE_1.PL1
        MODULE_2.PL1
        MODULE_3.PL1
        GENERAL_SUBS.PL1

    ) LOAD/V TAPE:1
    12-MAY-83  14:00:14
        PROGRAM.PR
        MAIN_PROG.MEMO
```

Note that when ZIPPY references "TAPE:0" and "TAPE:1", he is really accessing tape files "@MTB0:0" and "@MTB0:1".

(10) ZIPPY has gotten the information he needs and issues a CLI DISMOUNT command to notify OP that he is done:

```
) DISMOUNT TAPE
```

This command sends an IPC to EXEC indicating that ZIPPY is done with the tape unit.

The CLI DISMOUNT command does not pend like the MOUNT command so that the user's CLI regains control immediately.

(11) When EXEC receives the DISMOUNT request it:

(a) Deletes the link file "TAPE" from ZIPPY's directory.

(b) Sets the ACLs on drive @MTB0 to null (so no one can write to it).

(c) Places a tape unit entry in syslog indicating how long ZIPPY had control of the @MTB0 unit.

(d) Writes the following DISMOUNT message on the OP console:

```
                ** WAITING TO BE DISMOUNTED **

        MID=2, USER=ZIPPY, PID=26, EXEC SUB-TREE PID=26

        UNIT(S) ARE: @MTB0

        RESPOND:  CONTROL @EXEC DISMOUNTED
```

The operator must issue a "DISMOUNTED" command; OP cannot refuse to dismount the tape.

(12) When the operator receives this request, s/he issues the command "CONTROL @EXEC DISMOUNTED". This causes EXEC to mark the tape unit as unused and releases ZIPPY's mount descriptor to the free memory pool.

Non-Default Mount Descriptors

In the previous section, we assumed that there was only one mount request outstanding. However, at times there will be many outstanding requests at one time.

In the above "CONTROL @EXEC" commands, the operator did not indicate which mount descriptor the commands were for. For example, OP simply entered "CONTROL @EXEC MOUNTED @MTB0" and EXEC figured out that it was for ZIPPY's mount request.

(1) If OP does not specify a mount descriptor in his/her commands, EXEC assumes that the command refers to the first mount descriptor on the mount chain that requires action (the default MD).

EXEC links the mount descriptors onto the chain in the order in which they are received.

(2) If OP wishes to perform an operation on a mount request other than the default, s/he must specify a "mount id" (MID) in the command. For example, the following commands refer to mount request 3 (which need not be the default MD):

```
CONTROL @EXEC MOUNTED/MID=3 @MTB0

CONTROL @EXEC DISMOUNTED 3

CONTROL @EXEC MOUNTSTATUS 3
```

EXEC's MOUNTSTATUS display returns the MID for each mount descriptor (sed MOUNTSTATUS commands in previous example).

(3) When EXEC receives an operator command for a non-default mount descriptor, EXEC simply performs the appropriate operation(s) using the specified MD instead of the default.

Labelled Tape

"Labelled Tape" allows a single file to span several tapes. This is desireable when trying to dump lots of files at once and they will not fit on one single tape (e.g., backing up all of :UDD at once).

(1) To use labelled tape, the user (or OP) must first create a "labelled tape set"; that is, a set of tapes that are grouped together in a specific order.

(2) Each tape in a "labelled tape set" has some information stored at the beginning (in the "label" area).

Each tape label contains a unique name that differentiate it from the rest of the tapes in the set. Often, these names will indicate where in the ordered set the tape is (e.g., three tapes may be named "VOL1", "VOL2", "VOL3"). Each labelled tape is called a "volume" and each name is referred to as a "volume id" (or "volid" for short).

(3) The user must place these names on the various tape volumes with the LABEL utility. For example, the command "X LABEL @MTB0 VOL1" places the label "VOL1" onto the tape that is mounted on @MTB0. (Refer to the documentation for more information on the LABEL utility.)

(4) Though EXEC knows which volumes are on which tape drives, it is the AGENT (not EXEC) that actually looks at the labels (and validates them). Thus, the AGENT knows the format of the labels; EXEC does not.

Referencing Labelled Tapes

(1) When the user wishes to reference a labelled tape set, s/he issues a special version of the CLI MOUNT command (or ?EXEC system call) and specifies the names of all volumes in the labelled tape set in the correct order.

```
MOUNT/VOLID=VOL1/VOLID=VOL2/VOLID=VOL3 TAPE BACKUPS FOR 4/1/83
```

"VOL1", "VOL2", "VOL3" are the volumes in the labelled tape set.

"TAPE" is a logical unit name.

"BACKUPS FOR 4/1/83" is a comment that will be forwarded to the operator with the mount request.

(2) When EXEC receives this mount request, it displays the following prompt to the operator:

```
** EXPLICIT LABELLED MOUNT **

MID=7, USER=ZIPPY, PID=12. EXEC SUB-TREE PID=8

REQUEST IS 'BACKUPS FOR 4/1/83'

UNIT(S) ARE: NONE

CURRENT VOLUME: VOL1, ALL VOLUME(S): VOL1, VOL2, VOL3

RESPOND: CONTROL @EXEC MOUNTED @UNITNAME

OR:      CONTROL @EXEC REFUSED
```

(3) OP will mount the first volid (VOL1) and respond with the command "CONTROL @EXEC MOUNTED @MTB0".

(4) EXEC will link the unit descriptor for @MTB0 with the user's mount descriptor and send a response back to the user.

(5) The user may now reference the labelled tape set with the appropriate commands. For example:

```
) DUMP/V/REC TAPE:0 :UDD:#
```

(6) When the user accesses the tape, the AGENT will perform all necessary label validations (e.g., make sure OP mounted the correct volume, make sure the tape has not expired, etc.).

(7) Thus far, the labelled tape session is the same as the unlabelled tape session (from EXEC's standpoint).

(8) If the user's DUMP reaches the end of VOL1 without finishing the DUMP operation, the following occurs:

(a) The AGENT will realize that the labelled tape file should be continued on the next volume of the set.

(b) The agent will generate a "NEXT VOLUME" ?EXEC request (IPC) to EXEC indicating that the operator should mount the next volume of the set.

(c) The "NEXT VOLUME" request occurs without the user knowing. As far as s/he is concerned, the DUMP operation is proceeding along without interruption.

(9) When EXEC receives the "NEXT VOLUME" request, it finds the appropriate mount descriptor and prompts OP to mount the next volume in the labelled tape set.

```
** EXPLICIT LABELLED MOUNT ** NEXT VOLUME **

MID=7, USER=JEFF, PID=8, EXEC SUB-TREE PID=8

REQUEST IS 'BACKUPS FOR 4/1/83'

UNIT(S) ARE: @MTB0

CURRENT VOLUME: VOL2, ALL VOLUME(S): VOL1, VOL2, VOL3

RESPOND: CONTROL @EXEC MOUNTED

OR:      CONTROL @EXEC REFUSED
```

Note that EXEC indicates which volume OP should mount (e.g., "CURRENT VOLUME:").

(10) OP then mounts the next volid in the set and responds with the "CONTROL @EXEC MOUNTED" command. This sends a return IPC to the AGENT.

(11) Again, the AGENT performs some validation on the tape:

(a) Makes sure the volid is correct.

(b) Makes sure that the volid is from the correct labelled tape set.

(c) Makes sure the new tape has not expired.

(12) When the AGENT is convinced that the new volid is the correct one, the DUMP proceeds onto this tape from the point where it left off earlier.

(13) This sequence of events will proceed until one of the following occurs:

(a) The user's DUMP completes successfully.

(b) The user reaches the last volume in the set and the DUMP has not completed (in this case, the user receives an error).

Multiple Units for One Mount Request

It is interesting to note that the operator need not mount the second (or third, etc.) volid on the same tape unit that the first one was on.

(1) For example, suppose the operator mounted VOL1 on @MTB0. When the operator receives the next volume request, s/he may mount that volume on @MTB1 and issue the command "CONTROL @EXEC MOUNTED @MTB1".

(2) In this case, EXEC will link the unit @MTB1 to the mount descriptor. @MTB0 is still linked in and the 2 units form a "mount descriptor unit chain" (see diagram).

Premounting Units

The "premount" feature is a natural extension of the multiple unit functionality. "Premounting" a tape means that the operator mounts a tape on a unit for a user before the request for that tape actually comes in to EXEC. That is, the operator anticipates the need for a tape and mounts it before it is requested.

(1) In the previous labelled tape example, the user specified that there are 3 volumes in the labelled tape set. After the operator mounts VOL1 on @MTB0, s/he knows that the user will probably want to access VOL2 next.

(2) Instead of waiting for the "NEXT VOLUME" request, OP can place VOL2 on unit @MTB1 and issue the command:

```
CONTROL @EXEC PREMOUNT @MTB1 VOL2 ZIPPY
```

In the PREMOUNT command, the operator must supply the following:

(a) The unitname the tape is on.

(b) The volid of the tape that OP mounted.

(c) The username that will access the tape.

(3) When EXEC receives the PREMOUNT command, it does the following

(a) Links the specified tape unit descriptor into the mount descriptor unit chain (as described above) and marks the unit as "mounted" and "not open" (i.e., it is allocated but is not actually in use at this time).

(b) EXEC sets the ACLs to "<USER>,OWARE" so that no one else can access the drive.

(4) When the "NEXT VOLUME" request comes in for that user, EXEC will search the units associated with that mount descriptor to determine if the volume it needs is already mounted.

(5) In this case, EXEC finds the next volume already mounted (on @MTB1). Thus, EXEC will send the appropriate answering IPC back to the user indicating that the next volume is mounted. This occurs without bothering the operator (no OP intervention is necessary).

(6) EXEC's MOUNTSTATUS and UNITSTATUS commands always show:

(a) Which unit(s) are associated with which mount descriptors.

(b) Which unit(s) are mounted, premounted, and not in use.

Still More on Labelled Tape

In the above discussions, we have focussed on the normal, straight forward use of EXEC's labelled tape functions. There are, in addition, a variety of additional features that EXEC provides:

(1) Specific Volumes

EXEC allows the user to jump directly into the Nth volume of a labelled tape set without scanning through all of the ones before that.

For example, if you know the file you want to access begins on volume VOL3, you can ask OP to mount VOL3 directly. without first going through VOL1 and VOL2 first.

See the LOAD and DUMP /SPECIFIC switch documentation for more information.

(2) Extending Volid Lists

If the user specifies the /EXTEND switch on his/her MOUNT request, the operator may extend the user's labelled tape set if the tape operation reaches the end of the specified set but has not completed. That is, if a user's operation comes to the end of the volid set, EXEC allows the operator to append tapes to the end of the set (rather than return an error to the user).

See the MOUNT/EXTEND documentation for more information.

(3) Error Handling

When an error occurs, the AGENT and EXEC must determine whether it is the user's problem or the operator's problem and notify the responsible individual.

If the problem is the operator's (e.g., wrong volid mounted, unit off line, etc.), EXEC will tell OP what is wrong and give him/her a chance to correct it. The user will not be disturbed and when the problem is corrected by the operator, the user's operation will continue.

If the error is the user's fault, s/he will receive an error on the current operation.

(4) Other Mount Options

EXEC provides support for the following:

* Tape densities

* Read only tapes

* IBM Format

(5) Implicit Mount Requests

In the above examples, the user explicitly asked for his/her labelled tapes to be mounted. That is, the user explicitly issued the MOUNT command (or the corresponding ?EXEC call). We refer to these mount requests as "explicit" ones.

In addition to "explicit mount requests", the user may generate "implicit" mount requests. In an implicit mount request, the user does NOT issue a MOUNT command. Rather, s/he references a labelled tape in the LOAD, DUMP, COPY, etc. command directly. For example, suppose a user issued the following command (WITHOUT previously issuing a MOUNT command):

```
DUMP/V  @LMT:VOL1:0  :UDD:#
```

"@LMT" is a special file created by EXEC that stands for "Labelled Magnetic Tape".

"VOL1" is the volid of the labelled tape the user wishes to access.

In this case, the AGENT will look at the destination of the dump (@LMT:VOL1:0) and determine that the user wants to dump to a labelled tape. The volid of the labelled tape is VOL1.

When the agent detects an implicit labelled tape reference, it generates a MOUNT request to EXEC on the user's behalf (the user does not know that this occurs). The AGENT informs EXEC that the request is implicit and EXEC, in turn, tells the operator that the request is implicit (in the mount request display).

After the operator mounts the specified volume, the AGENT proceeds with the operation that the user wants to perform (in this case, a DUMP operation). When the operation is complete, the AGENT will generate a DISMOUNT request to EXEC so that the tape(s) will be dismounted and the mount request deleted.

Though the implicit mount functionality is convenient and easy to use, it is less powerful and less flexible that the explicit mount functionality. The disadvantages stem from the fact that the user cannot specify all of the information and options in his/her implicit request that s/he can supply on the explicit MOUNT command (e.g., tape density, text message, read only, IBM format, etc.).

In particular, the user cannot supply the complete volid list, only the first one. This means that EXEC and the operator do not know the next volumes required by the user. Thus, when a next volume requests comes in, EXEC cannot tell the operator the name of the volume that is required. Rather, the user and operator must pass this information in some other way (i.e., independent of EXEC).

We recommend that people use the explicit mount functionality when they have a choice. Generally, implicit mount references are used in programs that issue reads and writes but have no knowledge of labelled tapes. In these cases, the user may specify his input/output file as "@LMT:<VOLID>:<TAPEFILE>" and the program can be made to access labelled tapes without knowing it.

Mount Prompter Task

The "mount prompter task" (or simply the "mount task") is the task that displays the mount requests on the operator's console. The mount task operates as follows:

(1) In its idle state, the mount task is pended on an intertask message. That is, it waits to be awakened by another EXEC task.

(2) The following operations wake up the mount task:

(a) A incoming mount/dismount request from a user (the ?EXEC task will wake up the mount task in this case).

(b) An operator command fulfilling a mount request (the CONTROL @EXEC task will wake up the mount task in this case).

(3) When poked, the mount task searches the mount descriptor chain looking for a request that needs to be serviced by the operator.

(a) If any of the "action bits" in a mount descriptor are set, the request needs action by the operator.

(b) If there are no mount requests or if none of them require operator action, the mount task goes back to sleep.

(4) If the mount task finds a request that needs operator action, it displays that request on the operator's console (i.e., the mount task prompts the operator to fulfill the request).

(a) The mount task only displays 1 request at a time.

(b) The mount task displays the first mount request on the MD chain that requires action. As described earlier, this is the "default mount request".

(5) The mount task's display to the operator's console includes the following information:

* Type of tape request (e.g., labelled/unlabelled, explicit/implicit, first/next/specific volume, mount error, etc.).

* Mount ID (MID).

* Username of requestor.

* Pid of requestor.

* Pid of user's process immediately subordinate to EXEC.

* Request text supplied by user.

* Unit(s) associated with the mount request, if any.

* OP's possible responses to the request.

Except for the possible responses, the mount task's display is identical to the MOUNTSTATUS display. (See the sample displays in the earlier examples.)

(6) After displaying the request to the operator, the mount task delays for a specified amount of time and then wakes up to display the mount request again.

* The delay time starts at 1 minute and doubles with each display up to a maximum of 16 minutes. At this point, the mount task will prompt the operator to answer the request every 16 minutes.

(7) When the operator answers the request (either fulfills it or refuses it), the mount task goes back to step #3 above and searches the chain for another mount request to display.

* Again, if none can be found, the mount task goes to sleep until it is awakened by another EXEC task.

(8) As we described earlier, the operator need not specify a mount ID (MID) in his/her CONTROL @EXEC commands when referencing the default mount request.

* Again, the default mount request is the one that the mount task is currently displaying.

(9) Also as mentioned earlier, if the operator wishes to view the status of a mount request other than the default, s/he may issue the CONTROL @EXEC MOUNTSTATUS command.

(10) Similarly, the operator may fulfill a request other than the default mount request by supplying a MID value in the CONTROL @EXEC command.

* In this case, the mount task will continue to prompt the operator to act on the default mount request since it still requires action.

EXEC Memory Management

This section describes how EXEC manages free memory within its logical address space. Each ring has its own memory management system; the following sections describe each of these.

Ring 7 Memory Management

The following sections describe the major characteristics, internal design, and allocate/release operations of EXEC's ring 7 free memory pool.

[Ring 7 memory code resides in the EXEC resource management module EXRSM.SR.]

Ring 7 Memory Management Characteristics

EXEC's ring 7 memory management has the following characteristics:

(1) EXEC's ring 7 free memory pool starts immediately after the unit descriptors and extends to the end of the last memory page allocated to EXEC.

(The unit descriptors, allocated during initialization, reside at location ?NMAX.)

(2) EXEC issues a ?MEMI system call whenever it needs to expand the memory pool (1 page at a time). Since ring 7 EXEC is 16-bit, its maximum size is 32K words.

(3) EXEC never shrinks. That is, EXEC never releases free memory pages back to the system.

(4) The free memory pool is organized such that any length area can be acquired exactly (e.g., no rounding up to the nearest power of 2, etc.).

(5) When releasing memory back to EXEC's pool, the length of the area need not be supplied -- only the address.

(6) Free space is garbage collected at acquire time, but only insofar as it is required to find the first free area large enough to fulfill the request.

Ring 7 Memory Organization

EXEC's ring 7 memory pool is organized as a series of areas, each beginning with its length. If an area is free, the length

value is positive; if the area is in use, the length is negative. (See diagram).

Allocating Ring 7 Memory

The length word is "invisible" to the acquirer. When N words are requested, EXEC allocates N+1 words. The length of the area (e.g., N+1) is negated and stored in the first word; that is, -(N+1) is stored. The requestor is given the address of the second word and, thus, receives an area of size N.

If a free area is not large enough to satisfy the request, the next area is checked. If this area is also free, EXEC combines the two adjacent free areas and checks to see if the new area created is large enough. This free memory combination continues until the request is satisfied or until EXEC determines that there is not enough free memory available.

If there is not enough memory available, EXEC issues a ?MEMI call to increase its logical address space by another page. This new page is added to the free memory pool and EXEC again tries to fulfill the request.

If EXEC is already 32K words in size, it will receive an error on the ?MEMI system call. When this happens, EXEC sends a warning message to the operator and then pends the allocate operation waiting for a release to occur. When a memory release occurs, the allocate operation is unpended and EXEC again tries to find enough free memory to satisfy the request.

Releasing Ring 7 Memory

At release time, the releaser specifies the memory area's address. The release routine accesses the word before the given address (the invisible word), which contains the length of the memory area.

EXEC performs a gross validity check to make sure the word is negative.

EXEC then makes the length value in the invisible word positive to show that the area is free.

Ring 6 Memory Management

The following sections describes the major characteristics, internal design, and allocate/release operations of EXEC's ring 6 free memory pool.

Ring 6 Memory Characteristics and Organization

The following list describes the major characteristics of ring 6 memory management:

(1) Ring 6 maintains free block chains of sizes 8, 16, 32, 64, 128, 256, 512, and 1024 words each (See diagram).

(2) Initially, EXEC starts out with a page of memory on each of these chains.

(3) When a page of memory is allocated to EXEC (via ?MEMI) that whole page is broken up and placed on a single chain. For example, the whole page will be broken up into 16-word chunks and all will be placed on the 16-word chain. (Note that this is NOT a buddy system.)

Allocating Ring 6 Memory

When one of EXEC's ring 6 tasks needs memory, it calls the allocate memory routine specifying the size it needs. The allocation routine will determine the smallest size block (power of 2) that can fulfill the request (e.g., if you request 12 words, you will receive a 16-word block).

EXEC then goes to the appropriate chain, gets the first available block on the chain, and returns the address to the requestor.

If there are no free blocks on the needed chain, EXEC requests another page of memory from the system (?MEMI), breaks the whole page up into blocks of the needed size, and places all of these blocks onto the same chain. For example, if a user needs a 32-word block and there aren't any, EXEC gets another page, breaks it up into 32-word chunks, and places all of these chunks on the 32-word free chain.

After breaking up the new page onto the appropriate chain, EXEC fulfills the memory request with the first block on the chain.

If EXEC receives an error on the ?MEMI system call, it terminates with an internal consistency error. Since ring 6 EXEC is 32-bit, if it exceeds its logical address space, EXEC has grown way too big and something is wrong.

Releasing Ring 6 Memory

When a ring 6 EXEC task wants to release some memory back to the free pool, it calls the FREE_MEMORY routine specifying (a) the memory address and (b) the size. The size specified by the releaser is the same size specified at allocation time and, thus, is not necessarily the actual size of the block.

When EXEC receives the memory release request, it rounds the size up to the next power of 2 as it did during the allocation process (e.g., 8, 16, 32, etc.). EXEC then places this memory area onto the corresponding free memory chain.

COOPERATIVES

A "COOPERATIVE" (or "coop") is a process that relies upon EXEC's queueing and spooling facility to initiate job processing. Currently, all cooperatives are either ?PROC'd or ?CON'd to by EXEC. EXEC currently knows about five cooperative processes. Each of these processes has an associated device or a system resource that it utilizes. These are the ccoperatives and the devices or system resources:

1) XLPT.PR controls line printers.

2) XPLT.PR controls digital plotters.

3) FTA.PR transfers a file over a network to a remote machine.

4) HAMLET.PR is the IBM a serial transfer device.

5) SNA.PR allows DG machines to be used as a host in an IBM network.

Single-streamed or Multi-streamed Cooperatives

A cooperative process is either one of the following, single-streamed or multi-streamed. XLPT and XPLT are single-streamed coops, they accept jobs from only one stream at a time. The networking cooperatives,
FTA, HAMLET and SNA/RJE are all multi-stream coops. FTA and HAMLET accept jobs from any number of streams in the range 1 to 7 and SNA/RJE accepts jobs from any number of streams in the range 1 to 6. The exact number of streams is set at NETGEN time for that cooperative.

Tasks associated with the Coop world functions

1) CONTROL @EXEC (or CX) task
   This task pends on an ?IREC on local port CONPRT waiting for OP to issue a CONTROL @EXEC command.

2) Coop listener task
   This task pends on an ?IREC on local port COPPRT waiting for IPCs from EXEC's coops.

3) ?EXEC request task
   This task also pends on an ?IREC, on local port REQPRT, waiting for a user to issue a ?EXEC request (either via a ?EXEC call or via a CLI Q-command which ultimately resolves to a ?EXEC call).

4) Dequeuer task
   This task waits on a ?REC on mailbox DEQMBX waiting to be poked so it can search the queues for a job to spool to an idle coop or batch stream.

5) Term task
This tasks pends on an ?IREC on system port ?SPTM waiting for termination messages for sons of EXEC.

EXEC and Cooperative Communications

EXEC and the cooperatives communicate via IPC messages. To send a message to a coop, EXEC issues a ?ISEND; to receive a message from a coop EXEC issues an ?IREC. EXEC's CX task initiates the first of these IPCs when the coop is first brought up or is first connected to. The next round of IPCs comes from the coop when it notifies EXEC's coop listener task that it is ready to process jobs. All other IPCs which pass between EXEC and the coop are directives which request the coop to perform the required function to fulfill the request. All of the IPCs which pass between EXEC and its cooperatives will be discussed in more depth later.

Databases

A. Major databases

1) In-core coop descriptor (CCD)
A minimal amount of information describing a cooperative.

2) Disk or 'virtual' coop descriptor (VCD)
All the necessary information required to support a cooperative.

B. Related databases, bit maps and tables

1) Coop bit map
A bit map of all of EXEC's coops where the bit set represents the pid of that coop.

2) Sons bit map
A bit map of all of EXEC's sons and coops where the bits correspond to the son's or coop's pid.

3) Pidds table
A table of descriptors where the pid offset into the table points to the associated descriptor (in-core CD for coops).

4) Core queue
A minimal amount of information describing a job currently in a queue.

5) Disk queue
All the information available about a queued job (residing in the file :QUEUE:Q).

The Coop descriptor

The in-core Coop descriptor

The in-core coop descriptor, an abbreviated version of the disk coop descriptor,
lives within EXEC's address space and is allocated statically. The most frequently accessed words of the descriptor are duplicated in both databases, thus reducing the number of requests to disk to obtain the disk version of the database.

The disk or virtual coop descriptor

The disk (or virtual) coop descriptor is created at EXECs initialization time in the same directory in which EXEC was ?PROC'd, with the filename EXEC.COOPERATIVES. EXEC.COOPERATIVES is a user datafile with dynamic record format and file element size of 4, a must since we ?SPAGE one page of the file at a time into EXEC's address space. EXEC reserves the last page in its address space for its shared partition via a ?SSHPT.
See attachment 'A' for a complete description of the CCD and VCD databases.

Queues, Coops and Devices

Introduction to queues, coops and devices

When EXEC is proc'd all the queues which existed when EXEC last terminated will still exist when EXEC comes back up. These queues, however, are stopped (except for batch_input) and must be started before any jobs submitted to those queues can become active. Once the queues are started at a coop, the coops must be continued before job processing can occur. At this point, all jobs submitted to a queue can be processed by a cooperative process.

Queue Type

The relationship which exists between queues and coops is partially determined by the queue type. The queue type is set by the 'CONTROL @EXEC CREATE <queue type> <queuename>' command. Once a queue has been created with this command, it will always exist with its original queue type until the queue is deleted via a 'CONTROL @EXEC DELETE <queuename>' command.

Queues, Coops and Devices

In addition to the queue type, queues and devices are associated via a 'CONTROL @EXEC START <queuename> @<devicename>' command. The queue type determines which coop gets associated with the specified queue and device.

Please note that 'CONTROL @EXEC' will be replaced with the macro name 'CX' in all following command notation.

A. Associating queues with devices

To associate a queue with a particular device, the username 'OP' must issue a command of the form:

CX START <queuename> @<devicename>

where <queuename> is an existing queue, and <devicename> is the name of a device genned into the system, an IPC port to which EXEC will communicate (for the network coops), or simply a file existing in :PER. For example, the command:

'CX START LPT @LPB' indicates that all jobs submitted to queue LPT will be printed on device @LPB. Note that the device need not be '@LPB', it may be '@CON7' or '@FOO'.

B. Associating queues and coops

Once a particular queue is started at a device, EXEC must decide which coop to associate with them. This association is predefined by the function that the coop performs.

Coop	Function	Queue type
XLPT	prints a file to a predefined destination	print
XPLT	plots a file on a plotter	plot
FTA	transfers a file across a network	fta
HAMLET	acts as a serial file transfer device	hamlet
SNA/RJE	allows a DG machine to be used in an IBM network	sna

Initialization

Coop and Coop descriptor initialization

The CX START command not only associates the specified queue with the specified device, but it also prompts EXEC's CX task into initializing a CCD and VCD for that cooperative if one does not already exist. Also, the command eventually results in the CX task ?PROCing or ?CONnecting to the cooperative process that corresponds to the specified queue type.

A. CCD and VCD Initialization

1. Scan the CCDs looking for the first one where the PID location is zero. This is the first free CCD. Note that if the entire CCD block is scanned and no free CCDs are available, the error 'Maximum Cooperatives Exceeded' (60 or 48.) is returned and EXEC continues processing.

2. When a free CCD is found, the corresponding VCD on disk is brought into core. The CCDs and VCDs correspond sequentially to each other. Thus, if the first free CCD is the fifth in the CCD table, then the fifth VCD on disk needs to be paged in to memory. To retrieve the corresponding VCD from disk, the routine .GETCD is called. The .GETCD routine performs the following function:

   a) converts the CCD address to an address relative to the beginning of the CCD table,

   b) divides the address by the size of a single CCD (34) resulting in the entry number of the CCD in the CCD table (also the entry number of the VCD we want from disk).

   c) there are 4 VCDs/page so the entry number must be divided by 4 to give the page number within EXEC.COOPERATIVES where the requested VCD resides.

   d) If this page is already in memory (we keep the current page number that is paged in), then there is no need to ?SPAGE it in. Instead, all but the last 2 bits of of the entry number must be cleared. This converts the entry number to a number in relation to 4 CCDs/page.

   e) This number is then multiplied by 400 (400 words/VCD) to get the address of the VCD on the page,

   f) and the address is added to the address of VCDBUF (address of the first word of EXECs shared page partition).

   g) If the requested page is not in memory, then that page number is saved as the current page now in memory,

   h) and the page number is then multiplied by 4 (4 blocks/page).

i) The result, the first block number of the page requested is stored in ?PRNL of the ?SPAGE packet, ?PSTI is set to 4, and ?PCAD is set to VCDBUF.

j) The ?SPAGE is then issued to page in the requested page.

k) Steps d - f are then followed to convert the entry number to an address relative to VCDBUF.

3) When the VCD address is returned from .GETCD, the entire database is zeroed (not to be confused with the entire page which was paged in, only the requested VCD is zeroed).

4) The 'START' command code reserves space on the stack for the ?PROC packet. All the information currently known about the coop, the queue associated with it, and the device, is initialized in the CCD, VCD and ?PROC packet. When the CX task ?PROCs the coop, the initial IPC to the coop is passed in the ?PROC packet. The local port COPPRT is stored in ?IOPN of the IPC message header. This notifies the coop of the port to which messages should be passed. The ?PROC packet, IPC message, VCD and CCD must all be initialized. See attachment 'B' for a description of the initialized ?PROC packet, CCD and VCD.

5) FTA and RJE are somewhat different than previously described in 4 above. The CX task ?CONnects to FTA and RJE to obtain a customer-server relationship. The task then issues a ?ISEND to send the initial IPC message to those coops. The VCDs for these coops are initialized in almost the same way as the other coops VCDs. The one difference is FTA and RJEs IPC message header contains a pointer to an initial message separate from the VCD, which isn't the case for the other coops. Attachment 'B' reflects this difference.

QNAMS table, :QUEUE:Q and the queue number

When a 'START' command is issued, the queuename is searched for in a QNAMS table. QNAMS is a table of byte pointers to the queuenames which have been previously created. The offset into this table for the queuename corresponding to a particular coop, is its queue number. The queue number is also the offset of that queues queue descriptor into the file :QUEUE:Q.

Queue and Coop Relationship

In the CCD and VCD are two words which hold a queue bit map. For each queue that is started at a device, the cooperative associated with that device
has a bit set in its queue bit map (CDQ and CDQ2 in the VCD and CDQBM and CDQBM2 in the CCD) which corresponds to that queues number. Setting a bit in the bit map results in the association of the queue with that queue number with the coop which was ?PROC'd or ?CON'd to as a result of the 'START' command. The queue bit map initialization is done at CCD and VCD intialization time described above.

Multiple queues and coops

Several queues can be started at the same device, resulting in several bits being set in the coop descriptor queue bit map. Currently, 32. queues (total) are allowed in EXEC, 3 of which are the BATCH related queues. This is the reason for the 2 word bit map. It should also be noted that a single queue can service several devices or coops resulting in the same bit being set in the queue bit maps of all the coop descriptors associated with that queue.

Coops/Sons/PIDDS Tables

The COOPs bit map, SONs bit map and the PIDDS table are used by all of EXECs
tasks. It is at 'START' time that these tables get the appropriate values set for the newly proc'd coop. The pid of the coop is returned from the ?PROC or ?GPORT, in the case of FTA and SNA/RJE, and the bit corresponding to that pid is set in the COOPs' and SONs' bit maps. For the PIDDS table, the value currently in the corresponding offset into the table is retrieved and stored in the CDLNK word of the CCD and the CCD address for the newly proc'd coop is placed in the pid offset into the PIDDS table (this is done to prevent possible confusion within EXEC when a terming son and a newly proc'd son are assigned the same pid and EXEC hasn't cleared the terming son's databases yet).

Continuing the cooperative

Once the cooperative has been proc'd or connected to, it must be continued. EXEC initially brings the coops up as paused (i.e., the paused bit in word STS of the VCD and CDSTS of the CCD is set and thus prevents any jobs from being spooled to that cooperative), requiring OP to issue a 'CX CONTINUE @<device>' to clear the paused bit so the coop will be marked as idle and thus ready to receive jobs for processing.

The initial response message

After the coop receives the initial IPC message from EXEC, it knows that EXEC is ready to accept messages from it. The coop must send a 'READY'
message to EXEC indicating that it is ready to begin processing jobs. If the coop is multi-streamed, it must send one ready message per

stream. Each ready message contains the stream number to which the message applies.

Job Processing

Job Processing

When the coop has sent its initial 'READY' message to EXEC, it is ready for the dequeuer task to spool a job(s) to it for processing. The dequeuer task searches the disk queue entries in :QUEUE:Q for a job to spool to the idle coop. When the best job is found, it is selected, marked as active, the coop is marked as busy, and the dequeuer task issues a 'RUN THIS JOB' IPC to the idle coop. The message that is sent to the coop contains all the information about the job that is necessary for the coop to process the job. It should be noted that the message is slightly different for the multi-streamed coops. See attachment 'C' for the exact format of the multi-streamed coops 'RUN THIS JOB' IPC.

READY and DONE Messages

When the coop finishes processing the job sent by EXEC, it responds with a READY message. A READY message from the coop indicates that the coop has finished the current job and is ready to accept another. When the coop listener task receives a READY message, it pokes the dequeuer again to initiate the search for another job to spool to the coop. If one is found, another RUN THIS JOB message is sent to the coop. This cycle of READY and RUN THIS JOB messages may continue between EXEC and the coop indefinitely. If at some point, the cooperative wants to tell EXEC that it has finished processing the current job but does NOT want EXEC to send another one, it sends a DONE message. When EXEC receives a DONE message, it does not send any message to the coop at all.
The coop must re-initiate communications with EXEC via a READY message.

Save UID Message

When the cooperative is FTA, and the recoverability functionality was specified in the FTA request, FTA sends EXEC an IPC specifying a two word unique ID. This ID is used to assist FTA in recovering should the system, EXEC, FTA or the network (all local or remote) terminate during a FTA request.

Received UID Message

When EXEC receives a SAVE UID IPC from FTA, it saves the two UID words in the disk queue for that job and sends an IPC message back to FTA notifying FTA that the UID was received.

Status Messages

At any point, EXEC can ask the cooperative for its status. This is triggered by a CX STATUS @<devicename> command. When a status command is executed, EXEC sends a STATUS request message to the specified coop. When the coop receives a STATUS request message, it determines the status of the job(s) it is running and returns that information to EXEC as a STATUS response message. The STATUS response indicates if the coop is idle (done and waiting for a job), not ready (done but not waiting for a job), or active. If active, thee coop will return information on the job being processed (e.g., sequence number, etc.). If the coop is multi-streamed, information on each stream must be supplied.

Pause Messages

Any time the coop is active (i.e., processing a job), EXEC can ask the coop to pause within the current job by sending a PAUSE message to the coop. Currently only XLPT has the ability to pause within a job. This directive is a result of a 'CX ALIGN' command. When the coop receives a PAUSE request, it determines when the best time is to stop processing the current job, and then does so. When the coop has stopped processing the current job, it sends a PAUSED message back to EXEC. EXEC then relays this information to the OP console where the operator may then take the desired action. When the operator wants the coop to continue processing the job, s/he issues a 'CX ALIGN/CONTINUE' command. EXEC then sends a RESUME message to the coop. At this point, the coop continues processing the paused job. The RESUME message can contain additional information of value to the coop (e.g., the ALIGN/CONTINUE command accepts an argument which directs XLPT to resume printing 'x' pages back from where it paused or on page 'x'). Note that the pause/

resume functionality described above is different from the 'CX PAUSE' command. The 'CX PAUSE' command directs EXEC to pause the coop BETWEEN jobs; this is implemented totally inside of EXEC by setting the paused bit in the coops CCD and VCD. The pause/resume functionality directs the coop to pause WITHIN a job; this is implemented inside the cooperative process.

Binary Message

The operator may at some point want to tell XLPT to go into binary mode. To do so, the 'CX BINARY <filename or OFF>' command is issued. The CX task sends a binary message to XLPT with the specified filename in the VCD and the binary bit set, or no file if none was specified, but the binary bit cleared.

Recover Message

When the system, EXEC, FTA or the network goes down and FTA is processing a request with recovery specified, EXEC checks the job's UID field in the disk queue and if one exists, sends FTA a recover IPC. This IPC prompts FTA to continue processing the job from its last checkpoint instead of restarting the job from the beginning.

Restart Message

Sometimes, it is desirable to start a job over at the beginning. To do so, the operator issues a CONTROL @EXEC RESTART command. The CX task then sends a RESTART message to the coop. Upon receiving this message, the coop goes back to the beginning of the job and starts it over again. If the coop is multi-streamed, the RESTART command will indicate which stream the RESTART is for.

Restart After Message

If at some point during a FTA transfer, FTA detects a problem on the remote machine, it may send EXEC a restart after IPC. This IPC message specifies a relative amount of time after which EXEC should restart the job. EXEC handles this request in the same way it handles a job submitted with the '/AFTER=' switch.

FLUSH and CANCEL Messages

To stop a coop from further processing an active job, the operator issues a CX FLUSH command or the user who submitted the job can issue a CLI QCANCEL command (or the corresponding ?EXEC cancel request). These commands prompt EXEC to send FLUSH a message or a cancelled by user message to the coop. This message directs the coop to stop processing the current job immediately. When the coop has flushed/cancelled the job, it sends a READY (or DONE) message to EXEC.

Error Message

At various times during its operation, the cooperative may want to display a message on the operator's console. Instead of sending a message directly, the coop can send an ERROR IPC message to EXEC and EXEC will relay that message to the operator's console. The ERROR message from the coop to EXEC may contain a text string and/or it may contain an AOS/VS system error code. If the message has a text string, EXEC will simply display that message on the operator's console. If the coop's message contains a system error code, EXEC will display the corresponding text message (obtained from the ?ERMSG system call).

Final Messages

When the system manager wants to stop the cooperative in an orderly, graceful manner, s/he issues a CX STOP command. EXEC will then send a STOP message to the cooperative. When the coop receives a STOP message, it finishes all job(s) currently being processed and sends DONE messages to EXEC for each one (i.e., done, but not ready for another job). After the cooperative is completely idle, it either (1) terminates itself in an orderly manner (usually, by issuing a ?RETURN system call); or disconnects from EXEC (?DCON system call).

Cooperative Terminations

Cooperative Terminations

As mentioned above, cooperatives can be terminated by the CX STOP command. The stop command with a devicename argument prompts the CX task to clear the coops queue bit map word in the CCD and VCD. Thus, the next time the dequeuer is poked and searches the CCDs for an idle coop, the queue bit map is checked and when found to be zero, a term IPC is sent to the coop and the coop terminates itself. The stop command with a queuename argument can be used to also term a coop. In this case, just the bit for the specified queue is cleared from the coops queue bit map. If, however. this was the only bit set in the queue bit map, then the dequeuer task will issue a term IPC to the coop the next time it checks the coop for possible next job processing. The CX TERM command can also be used to term a coop. This command simply issues a ?TERM on the cooperative.

Term task

EXEC's term task pends on a ?IREC waiting for its sons and coops termination
messages from the system. When a termination message is received, the following occurs:

1) The coops bit map is checked for a matching pid. If the termination is from a cooperative, the pid offset into the PIDDs table is checked for the correct CCD address. If the address is not the same, then the PIDLNK chain in that descriptor is followed until the CCD address is found.

2) When the CCD address is found, it is zeroed. If it was also in the PIDLNK chain, then the pids bit in the SONs bit map is left set, otherwise, it is cleared.

3) If there is a currently active descriptor in PIDLNK, it is checked for a CCD, and if so, then the pid bit in the COOPs bit map is left set, otherwise it is cleared.

4) The pid location and queue bit map are cleared from the CCD

5) The coop is checked for FTA or SNA/RJE and if so then the term task issues a ?DCON to disconnect from the process.

6) A message is then sent to the OP console to notify OP of the cooperatives termination.

The cooperative is at this point, cleared from EXEC's databases and is no longer a son of, or associated with EXEC.

CONTROL @EXEC Commands

CONTROL @EXEC commands

Several of the CX commands have some effect on cooperative processes. In general, these commands fall into one of three categories:

1) Those commands which direct a cooperative to take a specified action
(i.e., an IPC ultimately results from issuing the command and the cooperative must take some action):

ALIGN	BINARY	FLUSH	RESTART
START	STATUS	STOP	

2) Those commands which require support from the cooperative
(i.e., no IPC is sent, but new values are set in the coops VCD and CCD
resulting in changes in the cooperatives operation):

CPL	CONTINUE	DEFAULTFORMS	ELONGATE
EVEN	FORMS	HEADERS	LIMIT
LPP	PAUSE	SPOOLSTATUS	TRAILERS
UNLIMIT			

3) Those commands which are coop related but do not require support
(i.e., they apply to a cooperative process but have no effect on it):

BRIEF	PRIORITY	QPRIORITY	SILENCE
TERMINATE	UNSILENCE	VERBOSE	XBIAS

Attachment 'A' - VCD and CCD Databases

```
; THE VIRTUAL OR DISK COOP DESCRIPTOR DATABASE
; THE FIRST ?IPLTH WORDS (7) ARE FOR AN IPC HEADER.

000007 .DUSR    APR=?IPLTH              ;ASSOCIATED PROCESS I.D.
000010 .DUSR    TPKT=APR+1              ;TERM PACKET BUFFER
000011 .DUSR    BTS=TPKT+1              ;BATCH/LOGON STATUS BITS
000220 .DUSR    BTSDIS=(BTS*16.)+0      ;DISABLE SOON AS POSSIBLE
000221 .DUSR    BTSDTO=(BTS*16.)+1      ;TIMEOUTS ENABLED
000222 .DUSR    BTSITC=(BTS*16.)+2      ;I/O TERMINATED BY CLOSE
000223 .DUSR    BTSDCT=(BTS*16.)+3      ;DISCONNECTED
000224 .DUSR    BTSOPN=(BTS*16.)+4      ;BATCH: @OUTPUT OPENED
                                        ;LD:    SEND ENABLE MSG TO OP
000225 .DUSR    BTSCPW=(BTS*16.)+5      ;PASSWORD CHANGE REQUESTED
000226 .DUSR    BTSNIS=(BTS*16.)+6      ;POSTPONE WTLD ?ISEND FOR
                                        ;ANOTHER TASK
000227 .DUSR    BTSENA=(BTS*16.)+7      ;OPEN IS FIRST AFTER ENABLE
000230 .DUSR    BTSLCL=(BTS*16.)+8.     ;LAST CLOSE BEFORE DISABLE
000231 .DUSR    BTSTER=(BTS*16.)+9.     ;TERMINATION IN PROGRESS
000232 .DUSR    BTSFO=(BTS*16.)+10.     ;FORCED OUTPUT IN PROGRESS
000233 .DUSR    BTSVD=(BTS*16.)+11.     ;VTA DIED FOR THIS LD
                                        ;UNUSED BIT
                                        ;(WAS POST-?PROC IN PROGRESS)
000235 .DUSR BTSBAT= (BTS*16.)+13.      ;0=LD 1=BATCH DESCRIPTOR
000236 .DUSR BTSTAK= (BTS*16.)+14.      ;DO NOT RELEASE STACK
000237 .DUSR BTSLP=  (BTS*16.)+15.      ;LOG ON/OFF IN PROGRESS

000012 .DUSR    PIDLNK=BTS+1            ;LINK FOR PIDDS TABLE
000013 .DUSR    USR=PIDLNK+1            ;B.P. TO USER NAME
000014 .DUSR    POF=USR+1               ;B.P. TO ?PROC OUTPUT NAME
000015 .DUSR    PLF=POF+1               ;B.P. TO ?PROC LIST NAME
000016 .DUSR    DNM=PLF+1               ;B.P. TO ?PROC DEVICE NAME
000016 .DUSR    STN=DNM                 ;B.P. TO STREAM NAME
000017 .DUSR    UQPR=DNM+1              ;USER MAX QUEUE PRIORITY
000020 .DUSR    UPBTS=UQPR+1            ;USER PRIVELEGE BITS
000021 .DUSR    DCN=UPBTS+1             ;CONNECT TIME - DATE
000022 .DUSR    TCN=DCN+1               ;CONNEXT TIME - HOUR
000023 .DUSR    NXT=TCN+1               ;LINK TO NEXT DESCRIPTOR
000024 .DUSR    BIASF=NXT+1             ;BIAS FACTOR
000025 .DUSR    PRIORITY=BIASF+1        ;PROCESS TYPE & PRIORITY
000026 .DUSR    STS=PRIORITY+1          ;BATCH/COOP STATUS BITS
000000 .DUSR    STPAU=0                 ;PAUSE(D) AT END OF JOB
000001 .DUSR    STIDL=1                 ;IDLE (NOTHING TO DO)
000002 .DUSR    STVRB=2                 ;VERBOSE MESSAGES
000003 .DUSR    STTRM=3                 ;COOP TOLD TO TERMINATE
000004 .DUSR    STSIL=4                 ;SILENCED
000005 .DUSR    STLIM=5                 ;LIMITING
```

```
000006 .DUSR    STLP2=6             ;LP2 FLAG
000007 .DUSR    STELO=7             ;ELONGATE FLAG
000010 .DUSR    STUPP=8.            ;UPPERCASE ONLY ENABLED
000011 .DUSR    STUNE=9.            ;UNEVEN MODE ENABLED
000012 .DUSR    STBIN=10.           ;BINARY MODE ENABLED
000013 .DUSR    STALN=11.           ;WAITING TO BE ALIGNED
000014 .DUSR    STDOE=12.           ;BATCH: ERROR ON DEFAULT
                                    ;OUTPUT FILE
000015 .DUSR    STNL=13.            ;CONVERT <NL> TO <CR><NL>
        ;           14.             ; - RESERVED
        ;           15.             ; - RESERVED

        ;FOLLOWING OFFSETS ARE THE SELECT PACKET

000027 .DUSR    QSPK=STS+1          ;BASE OF PACKET
000027 .DUSR    QTYP=QSPK+SELTYP    ;TYPE OF ENTRY (BATCH)
000030 .DUSR    QDAT=QSPK+SELDAT    ;DATE ENTRY ENQUEUED
000031 .DUSR    QTIM=QSPK+SELTIM    ;TIME ENTRY ENQUEUED
000032 .DUSR    QLMT=QSPK+SELLMT    ;MAX CPU SECONDS, PAGES, ETC.
000033 .DUSR    QPRI=QSPK+SELPRI    ;PRIORITY
000034 .DUSR    QFGS=QSPK+SELFGS    ;ENTRY FLAGS
000035 .DUSR    QSEQ=QSPK+SELSEQ    ;SEQUENCE NUMBER
000036 .DUSR    QXW0=QSPK+SELW0     ;THE FOUR "X" WORDS
000037 .DUSR    QXW1=QSPK+SELW1
000040 .DUSR    QXW2=QSPK+SELW2
000041 .DUSR    QXW3=QSPK+SELW3
000042 .DUSR    QPRH=QSPK+SELPH     ;HIGHEST PRIORITY TO SELECT
000043 .DUSR    QPRL=QSPK+SELPL     ;LOWEST  PRIORITY TO SELECT
000044 .DUSR    QPID=QSPK+SELPID    ;ENQUEUER'S PID
000045 .DUSR    QUBP=QSPK+SELUBP    ;B.P. USERNAME (COPY OF USR)
000046 .DUSR    QFBP=QSPK+SELFBP    ;B.P. FORMS OR JOBNAME
000047 .DUSR    QPBP=QSPK+SELPBP    ;B.P. PATHNAME
000050 .DUSR    QOBP=QSPK+SELOBP    ;B.P. @OUTPUT (COPY OF POF)
000051 .DUSR    QLBP=QSPK+SELLBP    ;B.P. @LIST
000052 .DUSR    QDBP=QSPK+SELDBP    ;B.P. DESTINATION
000053 .DUSR    QHDL=QSPK+SELHDL    ;ENTRY'S QUEUE HANDLE
000054 .DUSR    QLMAX=QSPK+SELMX    ;MAXIMUM LIMIT TO ACCEPT
000055 .DUSR    CDFHH=QSPK+SEFHH    ;HASH OF FORMS
000056 .DUSR    CDDHH=QSPK+SEDHH    ;HASH OF DEFAULTFORMS

000057 .DUSR    CDQ=CDDHH+1         ;BIT MAP OF ACCEPTABLE QUEUES
000060 .DUSR    CDQ2=CDQ+1          ;(DOUBLE WORD)
001357 .DUSR    CDBOFS=(CDQ*16.)-1  ;BIT OFFSET FOR QUEUE #1

000061 .DUSR    CDTYP=CDQ2+1        ;QUEUE TYPE COOP STARTED ON
000062 .DUSR    CDHDR=CDTYP+1       ;# OF HEADERS
000063 .DUSR    CDTLR=CDHDR+1       ;# OF TRAILERS
000064 .DUSR    CDCPL=CDTLR+1       ;# OF COLUMNS/LINE
000065 .DUSR    CDLPP=CDCPL+1       ;# OF LINES PER PAGE
000066 .DUSR    CDXPH=CDLPP+1       ;GLOBAL PORT # FOR COOPS TO
000067 .DUSR    CDXPL=CDXPH+1       ;TALK TO EXEC.
```

```
000070 .DUSR    CDHS0=CDXPL+1          ;STREAM 0
000071 .DUSR    CDHS1=CDHS0+1          ;STREAM 1
000072 .DUSR    CDHS2=CDHS0+2          ;  :
000073 .DUSR    CDHS3=CDHS0+3          ;  :
000074 .DUSR    CDHS4=CDHS0+4          ;  :
000075 .DUSR    CDHS5=CDHS0+5          ;STREAM 5
000076 .DUSR    CDHS6=CDHS0+6          ;  :
000077 .DUSR    CDHS7=CDHS0+7          ;STREAM 7
000001 .DUSR    CDHSL=1                ;LOWEST STREAM #
000006 .DUSR    CDSSH=6                ;HIGHEST STREAM # FOR SNA/RJE
000007 .DUSR    CDHSH=7                ;HIGHEST STREAM # FOR HAMLET
                                       ;AND FTA
       ; AREAS FOR NAMES WITHIN COOP DESCRIPTOR
000100 .DUSR    CDQUE=CDHS7+1          ;QUEUE NAME FROM WHICH IT CAME
000120 .DUSR    CDDEV=CDQUE+QQLTH      ;DEVICE NAME OF SPOOLED DEVICE
000140 .DUSR    CDPTH=CDDEV+QQLTH      ;CURRENT FILE'S PATHNAME
000240 .DUSR    CDFMS=CDPTH+QPLTH      ;FORMS REQUESTED FOR IT
000240 .DUSR    CDFIL=CDFMS            ;FTA DEST PATHNAME/RJE QOUTPUT
                                       ;PATHNAME
000260 .DUSR    CDDFM=CDFMS+QFLTH      ;DEFAULT FORMS
000300 .DUSR    CDDES=CDDFM+QFLTH      ;DESTINATION
000340 .DUSR    CDUSR=CDFIL+QPLTH      ;USER WHO SUBMITTED REQUEST
000350 .DUSR    CDEND=CDUSR+QULTH      ;LAST WORD OF CD
000351 .DUSR    CDLTH=CDEND+1          ;LENGTH OF CD

 ; THE FOLLOWING ARE DUPLICATED FOR EASIER REFERENCE
000016 .DUSR    CDDBP=DNM              ;B.P. COOP (DEVICE) NAME
000014 .DUSR    CDQBP=POF              ;B.P. QUEUE NAME
```

```
                ; THE IN-CORE COOP DESCRIPTOR


        000000 .DUSR    CDAPR=0                 ;PID OF COOP
        000001 .DUSR    CDDPH=CDAPR+1           ;DESTINATION HI PORT
        000002 .DUSR    CDDPL=CDDPH+1           ;DESTINATION LO PORT
        000003 .DUSR    CDQHDL=CDDPL+1          ;QUEUE HANDLE
        000004 .DUSR    CDQTYP=CDQHDL+1         ;QUEUE TYPE
        000005 .DUSR    CDSTS=CDQTYP+1          ;STATUS WORD
        000006 .DUSR    CDQBM=CDSTS+1           ;FIRST WORD OF QUEUE BIT MAP
        000007 .DUSR    CDQBM2=CDQBM+1          ;SECOND WORD OF QUEUE BIT MAP
        000010 .DUSR    CDBPD=CDQBM2+1          ;BP TO DEVICE NAME
        000011 .DUSR    CDTMP=CDBPD+1           ;UNUSED LOCATION SO THAT THE
                             ;LINK WORD IS IN THE SAME LOCATION AS THE
                             ;OTHER DESCRIPTORS SO 'PIDLNK' CAN BE USED.
      000012 .DUSR      CDLNK=CDTMP+1           ;LINK TO DESCR OF PROCESS
                                                ;WITH SAME PID
        000013 .DUSR    CDDNM=CDLNK+1           ;AREA FOR DEVICE NAME
        000034 .DUSR    CCDSIZ=CDDNM+QQLTH+1    ;END OF IN-CORE DESCRIPTOR

;OTHER CCD RELATED SYMBOL DEFINITIONS

            ;DEFINE THE BIT OFFSET TO THE FIRST BIT IN THE QUEUE BIT MAP
        000137 .DUSR    CDQBOF=(CDQBM*16.)-1    ;BIT OFFSET TO FIRST BIT

            ;DEFINE THE MIN AND MAX NUMBER OF COOPS
        000000 .DUSR    MINCPS=0                ;MINIMUM NUMBER OF COOPS
        000060 .DUSR    MAXCPS=60               ;MAXIMUM NUMBER OF COOPS

            ;THE FOLLOWING DEFINE THE SHARED PAGE PARAMETERS AND THE
            ;SHARED PAGE FILE FOR EXEC.COOPERATIVES
        000001 .DUSR    NOPAGS=1.               ;ONLY ONE PAGE
        002500 .DUSR    CCDLTH=(MAXCPS-MINCPS)*CCDSIZ
        000037 .DUSR    CPAGE=32.-NOPAGS        ;STARTING PAGE NUMBER
        076000 .DUSR    VCDBUF=CPAGE*1024.      ;ADDRESS OF FIRST PAGE
```

Attachment 'B' - Initialized ?PROC packet, VCD, CCD and ?ISEND initial message

?PROC packet offsets for single-streamed Coops and HAMLET which are initialized before issuing the ?PROC.

```
?PFLG = ?PFPX
?PSNM = BP to coop name
?PIPC = VCD addr
?PNM = CDDBP+1
?PMEM = -1
?PPRI = 3
?PDIR = -1
?PCAL = -1
?PUNM = -1
?PPRV = -1
?PWMI = -1
?PMEH = -1
?PWSS = -1
?SMCH = 0
?SMCL = 0
```

VCD offsets which are initialized either before or after the ?PROC or ?CON of the coop.

```
?IDPH \  high and low global
?IDPL /  ports for FTA and SNA
?IOPN = COPPRT
?ILTH = CDLTH or H.IMLEN
?IPTR = VCD addr or HAMIO addr
APR = pid of coop
USR = BP to username area
CDQBP/POF = BP to queuename area
CDDBP/DNM = BP to devicename area
STS = status word
QPRI = 0
QPRL = 377
QUBP = BP to username area
QFBP = BP to forms name area
QPBP = BP to pathname area
QOBP = BP to defaultforms name area
QDBP = BP to destination pathname area
QLMAX = -1
CDFHH = -1
CDDHH = -1
CDQ  \  bit corresponding to the
CDQ2 /  queue # is set
CDQTYP = queue type from QSTATS table
CDHDR = 1   \
CDTLR = 0    \  printers only
CDCPL = 120  /
```

CDLPP = 102 /
CDDEV = @devicename

CCD offsets which are initialized either after the ?PROC or ?CON of the coop.

CDAPR = pid of coop
CDDPH = high destination port of coop
CDDPL= low destination port of coop
CDQTYP = queue type
CDSTS = status word
CDQBM = \ bit for corresponding
CDQBM2 =/ pid is set
CDBPD = BP to devicename area
CDLNK = link to next descriptor with same pid
CDDNM = @devicename

FTA and SNA/RJEs offsets which are initialized for the initial ?ISEND message to the coop.

H.IMHI = \ EXECs high global port #
H.IMHL = / EXECs low global port #
H.IMDV = devicename

Attachment 'C' - RUN THIS JOB IPC for multi-streamed coops.

```
        ;EXEC --> HAMLET "PROCESS THIS FILE" MESSAGE

000000 .DUSR    H.SN=     0           ;STREAM NUMBER
000000 .DUSR    H.XO=     H.SN        ;XWO = STREAM #
000001 .DUSR    H.X1=     H.XO+1           ;XW1
000002 .DUSR    H.X2=     H.X1+1           ;XW2
000003 .DUSR    H.X3=     H.X2+1           ;XW3
000004 .DUSR    H.FGS=    H.X3+1           ;STATUS FLAGS
000005 .DUSR    H.USR=    H.FGS+1          ;USER NAME
000015 .DUSR    H.PTH=    H.USR+QULTH      ;PATHNAME
000115 .DUSR    H.END=    H.PTH+QPLTH      ;END OF THIS MESSAGE
000116 .DUSR    H.LEN=    H.END+1          ;LENGTH OF THIS MESSAGE

        ; EXEC --> FTA "PROCESS THIS FILE" MESSAGE

000000 .DUSR    F.STR=    0           ;STREAM NUMBER
000001 .DUSR    F.XWO=    F.STR+1          ;XWO
000002 .DUSR    F.XW1=    F.XWO+1          ;XW1
000003 .DUSR    F.XW2=    F.XW1+1          ;XW2
000004 .DUSR    F.XW3=    F.XW2+1          ;XW3
000005 .DUSR    F.SEQ=    F.XW3+1          ;SEQUENCE NUMBER FOR JOB
000006 .DUSR    F.FGS=    F.SEQ+1          ;FLAGS WORD
000007 .DUSR    F.LMT=    F.FGS+1          ;LIMIT WORD (UNUSED)
000010 .DUSR    F.UIDH=   F.LMT+1          ;HIGH 16 BITS OF UID
000011 .DUSR    F.UIDL=   F.UIDH+1   ;LOW 16 BITS OF UID
000012 .DUSR    F.RESO=   F.UIDL+1   ; - RESERVED
000013 .DUSR    F.RES1= F.RESO+1     ; - RESERVED
000014 .DUSR    F.RES2= F.RES1+1     ; - RESERVED
000015 .DUSR    F.RES3= F.RES2+1     ; - RESERVED
000016 .DUSR    F.RES4= F.RES3+1     ; - RESERVED
000017 .DUSR    F.RES5= F.RES4+1     ; - RESERVED
000020 .DUSR    F.RES6= F.RES5+1     ; - RESERVED
000021 .DUSR    F.RES7= F.RES6+1     ; - RESERVED
000022 .DUSR    F.RES8= F.RES7+1     ; - RESERVED
000023 .DUSR    F.RES9= F.RES8+1     ; - RESERVED
000024 .DUSR    F.RES10= F.RES9+1    ; - RESERVED
000025 .DUSR    F.RES11= F.RES10+1   ; - RESERVED
000026 .DUSR    F.RES12= F.RES11+1   ; - RESERVED
000027 .DUSR    F.RES13= F.RES12+1   ; - RESERVED
000030 .DUSR    F.RES14= F.RES13+1   ; - RESERVED
000031 .DUSR    F.RES15= F.RES14+1   ; - RESERVED
000032 .DUSR    F.RES16= F.RES15+1   ; - RESERVED
000033 .DUSR    F.RES17= F.RES16+1   ; - RESERVED
000034 .DUSR    F.RES18= F.RES17+1   ; - RESERVED
000035 .DUSR    F.RES19= F.RES18+1   ; - RESERVED
000036 .DUSR    F.RES20= F.RES19+1   ; - RESERVED
000037 .DUSR    F.RES21= F.RES20+1   ; - RESERVED
000040 .DUSR    F.RES22= F.RES21+1   ; - RESERVED
000041 .DUSR    F.PATH=F.RES22+1     ;PATHNAME OF SOURCE FILE
000141 .DUSR    F.DEST=F.PATH+QPLTH  ;PATHNAME OF DESTINATION FILE
000241 .DUSR    F.USER=   F.DEST+QPLTH    ;USER NAME
000251 .DUSR    F.END=    F.USER+QULTH    ;LAST WORD OF MESSAGE
```

```
000252 .DUSR    F.LEN=    F.END+1         ;LENGTH OF MESSAGE


       ;EXEC --> SNA/RJE "PROCESS THIS FILE" MESSAGE

000000 .DUSR    R.STR=    0               ;STREAM NUMBER (RUN ON
                                          ;       THIS STREAM)
000001 .DUSR    R.XW0=    R.STR+1             ;XW0 - STREAM NUMBER
                                              ;     SPECIFIED BY USER
000002 .DUSR    R.XW1=    R.XW0+1             ;XW1 - (UNUSED)
000003 .DUSR    R.XW2=    R.XW1+1             ;XW2 "SPECIAL" RJE WORDS
                                              ;           PASSED AS IS
000004 .DUSR    R.XW3=    R.XW2+1             ;XW3 FROM USER'S PACKET
                                              ;                 TO RJE
000005 .DUSR    R.SEQ=    R.XW3+1             ;SEQUENCE NUMBER FOR JOB
000006 .DUSR    R.FGS=    R.SEQ+1             ;FLAGS WORD
000007 .DUSR    R.LMT=    R.FGS+1             ;LIMIT WORD (UNUSED)
000010 .DUSR    R.RES0=   R.LMT+1             ; - RESERVED
000011 .DUSR    R.RES1= R.RES0+1              ; - RESERVED
000012 .DUSR    R.RES2= R.RES1+1              ; - RESERVED
000013 .DUSR    R.RES3= R.RES2+1              ; - RESERVED
000014 .DUSR    R.RES4= R.RES3+1    ; - RESERVED
000015 .DUSR    R.RES5= R.RES4+1    ; - RESERVED
000016 .DUSR    R.RES6= R.RES5+1    ; - RESERVED
000017 .DUSR    R.RES7= R.RES6+1    ; - RESERVED
000020 .DUSR    R.RES8= R.RES7+1    ; - RESERVED
000021 .DUSR    R.RES9= R.RES8+1    ; - RESERVED
000022 .DUSR    R.RES10= R.RES9+1   ; - RESERVED
000023 .DUSR    R.RES11= R.RES10+1  ; - RESERVED
000024 .DUSR    R.RES12= R.RES11+1  ; - RESERVED
000025 .DUSR    R.RES13= R.RES12+1  ; - RESERVED
000026 .DUSR    R.RES14= R.RES13+1  ; - RESERVED
000027 .DUSR    R.RES15= R.RES14+1  ; - RESERVED
000030 .DUSR    R.RES16= R.RES15+1  ; - RESERVED
000031 .DUSR    R.RES17= R.RES16+1  ; - RESERVED
000032 .DUSR    R.RES18= R.RES17+1  ; - RESERVED
000033 .DUSR    R.RES19= R.RES18+1  ; - RESERVED
000034 .DUSR    R.RES20= R.RES19+1  ; - RESERVED
000035 .DUSR    R.RES21= R.RES20+1  ; - RESERVED
000036 .DUSR    R.RES22= R.RES21+1  ; - RESERVED
000037 .DUSR    R.RES23= R.RES22+1  ; - RESERVED
000040 .DUSR    R.RES24= R.RES23+1  ; - RESERVED
000041 .DUSR    R.PATH=   R.RES24+1 ;PATHNAME OF SOURCE FILE
000141 .DUSR    R.QOUT= R.PATH+QPLTH    ;PATHNAME OF QOUTPUT FILE
000241 .DUSR    R.USER=   R.QOUT+QPLTH  ;USER NAME
000251 .DUSR    R.END=    R.USER+QULTH  ;LAST WORD OF MESSAGE
000252 .DUSR    R.LEN=    R.END+1       ;LENGTH OF MESSAGE
```

Introduction to the CONTROL @EXEC Commands

Introduction to CONTROL @EXEC Commands

The CONTROL @EXEC (or 'CX') commands are commands issued by a user with the same username as EXEC. These commands request an action to be performed or initiated by EXEC.

:PER:EXEC Initialization

During EXEC's initialization, the file :PER:EXEC (or @EXEC) is created. File @EXEC is an IPC type file with local port CONPRT. This is the file to which all IPCs for the CX commands are sent. Note that :PER:EXEC and @EXEC are the same file.

CONTROL @EXEC Command Task

The CONTROL @EXEC command task (or 'CX' task) is the task in EXEC which handles processing CX commands. The CX task pends on an ?IREC on local port CONPRT, or control port @EXEC waiting for a user to issue a CX command. When the ?IREC is fulfilled, the CX task verifies the command and ?RCALLs the appropriate overlay and performs the requested action.

CONTROL @EXEC Commands

There are currently approximately 50 CX commands. These commands can be grouped into one of the following categories: the logon world, the queue/coop/batch world, the mount world or general use commands. These commands are listed in attachment 'A' by the categories listed above.

The Command IPC

When a user issues a CX command, the CLI issues a ?ENQUE call to @EXEC specifying the command which was entered. The ?ENQUE call results in an IPC being sent to @EXEC with the command as the message. EXEC's CX task pends on local port CONPRT waiting for a command to be issued. When the IPC is received, the CX task begins processing the command.

The 'CX.CLI' macro

On most AOS/VS systems, a macro CX.CLI exists. This macro contains the CLI command line 'CONTROL @EXEC %1-%'. This macro allows the string 'CONTROL @EXEC' to be replaced by 'CX' with the command string passed to the macro as the argument string. In all following notation, 'CONTROL @EXEC' will be replaced by the macro 'CX'.

Processing CONTROL @EXEC Commands

Command Processing

When an IPC is received on port CONPRT, the command entered by the user is passed in the IPC and is stored in buffer CONCMD. The command is stored as entered by the user. links are not resolved by the CLI or EXEC.
The CX task performs the following functions:

1) Gets the pid of the user who sent the IPC (issues a ?GPORT on the sender's ports) and issues a ?GUNM on that pid to get the username. The username is then compared to EXEC's username.

2) If the username is not the same as the username of EXEC, the error 'VALID ONLY FROM OPERATOR' is returned to the user.

3) The command entered by the user. which may be minimally unique, is looked up in the command table. If the string is not found in the table, or if the string is not unique, the CX task returns an error to the user.

4) If the command string is found in the command table, the CX task then parses the rest of the command string for switches and arguments (.LKUP returns a byte pointer to the delimiter following the command. In EXEC's case a null, comma, or a slash are three acceptable delimiters).

5) Parsing is done by checking the remaining command string byte by byte for a null (=>end of string), a comma (=> argument), or a slash (=> a switch).
   a) If a null is encountered, the number of arguments is pushed on the stack, the address of the byte pointer to the last switch is pushed on the stack, and the switch count is pushed on the stack.

   b) If an argument is encountered, the comma is converted to a null in the command string, the byte pointer to the argument is pushed onto the stack and the parsing continues.

   c) If a switch is encountered, the slash is converted to a null in the command string, the byte pointer to the switch string is pushed onto the stack and the parsing of the string continues.

   d) When parsing is complete. the argument count is verified for being in the correct range for that command. If it isn't, the appropriate error is returned (too few or too many arguments).

e) The switch count is then verified for being in the correct range for that command. If it isn't, the appropriate error is returned (too few or too many switches).

f) The overlay entry point for the command code is obtained from the command table and is pushed on the stack.

g) The switch count is placed in AC2, AC0 is cleared and the ?RCALL is issued to bring in the appropriate overlay.

h) Each specific command routine then validates the specific switches and arguments to verify that they are in fact valid for that command.

i) If any switch or argument is determined to be invalid, the appropriate error code is placed in AC0 and execution returns to the main CX command loop.

j) As long as all the switches and arguments are valid, processing continues, the requested action is performed and control returns to the main command loop.

k) Upon returning from the overlay, the CX task checks AC0 for an error code (i.e., AC0=0 => no error). If an error code was passed back, the corresponding message is written to the operators console, followed by the command, the switches, and arguments.

l) The CX task then checks the PROMPT value for 0. If it does not equal 0, then the time prompt is written to the op console.

m) The CX task is now finished processing CX command and issues an ?IREC to wait for another.

Introduction to the ?EXEC World

?EXEC Request

A ?EXEC request provides users with a method of requesting EXEC to perform various actions for them.

:PER:EXEC_REQUEST Initialization

During EXEC's initialization, the file :PER:EXEC_REQUEST is created. File @EXEC_REQUEST is an IPC type file with local port REQPRT. This is the file to which all IPCs for the ?EXEC requests are sent.

?EXEC Request Task

The ?EXEC request task is the task in EXEC which handles processing of ?EXEC requests. The ?EXEC request task pends on an ?IREC on local port REQPRT waiting for a user to issue a ?EXEC request. When the ?IREC is fulfilled, the request task does some validation and then ?RCALLs to the appropriate command overlay to perform the requested action.

?EXEC Functions

The ?EXEC Functions

The actions performed by EXEC when an ?EXEC request is made, fall into one of the following categories:

1) Submitting jobs to user queues
   Users issue the CLI commands, QPRINT, QBATCH, QSUBMIT, QFTA etc., or a ?EXEC call to place entries into EXEC's queues.

2) HOLD/UNHOLD/CANCEL queue entries
   Users may hold, unhold or cancel any job which they previously enqueued to a particular queue. This action is performed via a CLI QHOLD, QUNHOLD, QCANCEL or the proper ?EXEC call.

3) MOUNT/DISMOUNT tape requests
   A user may issue a tape mount or dismount request to have the operator put a tape up on a tape drive. The operator must be "on duty" (i.e., 'CX OPERATOR ON') for mount and dismount requests to be honored. The user must also be in EXEC's subtree to issue these commands. This is because EXEC must remove the mount descriptor database in the event that the user terminates before the request is complete. Mount and dismount requests can be made by issuing the CLI MOUNT and DISMOUNT commands or by issuing the appropriate ?EXEC request.

4) Status Information
   A user may want to obtain status information from EXEC by issuing a ?EXEC status call or by issuing one of the following CLI psuedo- macros: [!LOGON], [!CONSOLE] or [!OPERATOR]. The following information can be obtained by issuing a status request:
   a) operator "on duty" status
   b) whether the user is a son of EXEC, and if so:
      i) whether the user is logged on in batch or at a console
      ii) returns the pid of the most immediate son
      iii) returns the console name or stream name where the most immediate son was proc'd

The ?EXEC System Call Format

The ?EXEC system call

The ?EXEC system call requires the user to specify a packet containing all the information necessary for EXEC to perform the user's request.

?EXEC function codes

The first word of the ?EXEC request packet contains the function code value that corresponds to the user's request. There are currently 31 function codes, six of which are reserved for internal use. The first 23 codes correspond to the functions previously mentioned in chapter 2. See attachment 'B' for a complete list of the ?EXEC function codes.

The remaining ?EXEC packet

All ?EXEC packets have a function code in the first word of the packet. The remaining information in the packet varies depending on the particular function being requested. In general, the remaining information consists of
the pointers and strings which EXEC requires to fulfill the request. See attachment 'C' for the different ?EXEC system call packets.

Processing ?EXEC Requests

Processing a ?EXEC request

When a user issues a ?EXEC system call, the call is passed to the AGENT. The AGENT checks to see if the call is a ?EXEC call, and if so, validates the user's ?EXEC packet verifying that the entries are reasonable.

The ?EXEC packet verification

When the AGENT validates the user's ?EXEC packet, it checks for the following in its validation:

1) The function code must be a valid code number (1-31)

2) Depending on the function code, the AGENT then validates that the data is the correct type for that code (i.e., the data in the packet is not validated, just the type). For example,
   a) valid numbers
   b) valid pathnames
   c) valid queuenames
   d) valid username or filename
   e) valid devicename

3) If any information fails to validate, the AGENT passes an error directly to the user without ever sending an IPC to EXEC.

This amount of AGENT validation reduces the amount of system overhead that would occur if EXEC performed this level of validation.

The AGENT IPC Message to EXEC

The AGENT builds an IPC message in its own address space from the ?EXEC packet that the user specified. The IPC message contains all the strings and data necessary for EXEC to fulfill the request.

The AGENT's ?IS.R to EXEC

The AGENT issues a ?IS.R to EXEC's local port, REQPRT. The IPC message contains the translated ?EXEC packet. The AGENT waits for a response from EXEC.

Processing The request

EXEC's ?EXEC request task receives the IPC and begins processing:

1) Verifies that the IPC came from the AGENT and ignores the IPC if not (EXEC can't return an error to an unknown sender).

2) Validates the function code (1-31)

3) Pushes the overlay routine address on the stack

4) ?RCALLs to the appropriate overlay routine to perform the requested action.

5) In the overlay routine, the request task performs further validation of the ?EXEC packet returning an error to its main routine if any values are found to be incorrect or if the requested action could not be performed.

6) When the requested action is complete or when an error is returned, the request task sets up an IPC to send back to the AGENT.

7) The IPC to the AGENT contains all the information which EXEC returns to the user for his/her requested action, or an error if one occurred. This IPC satisfies the AGENT's ?IS.R.

8) The AGENT interprets EXEC's IPC message, places the information which is to be returned to the user in the user's ?EXEC packet and returns control to the user.

9) The request task then loops back to pend on the ?IREC waiting for another user ?EXEC request.

Attachment 'A' - CONTROL @EXEC COMMANDS

Logon World Commands:

CONSOLESTATUS DISABLE ENABLE TERMINATE

Queue/Coop/Batch World Commands:

ALIGN	BINARY	BRIEF	CANCEL
CLOSE	CONTINUE	CPL	CREATE
DEFAULTFORMS	DELETE	ELONGATE	EVEN
FLUSH	FORMS	HEADERS	HOLD
LIMIT	LPP	OPEN	PAUSE
PRIORITY	PURGE	QPRIORITY	RESTART
SILENCE	SPOOLSTATUS	STACK	START
STATUS	STOP	TRAILERS	UNHOLD
UNLIMIT	UNSILENCE	VERBOSE	XBIAS

Mount World Commands:

DISMOUNTED	MOUNTED	MOUNTSTATUS	PREMOUNT
REFUSED	UNITSTATUS		

General Use Commands:

LOGGING MESSAGE OPERATOR PROMPTS

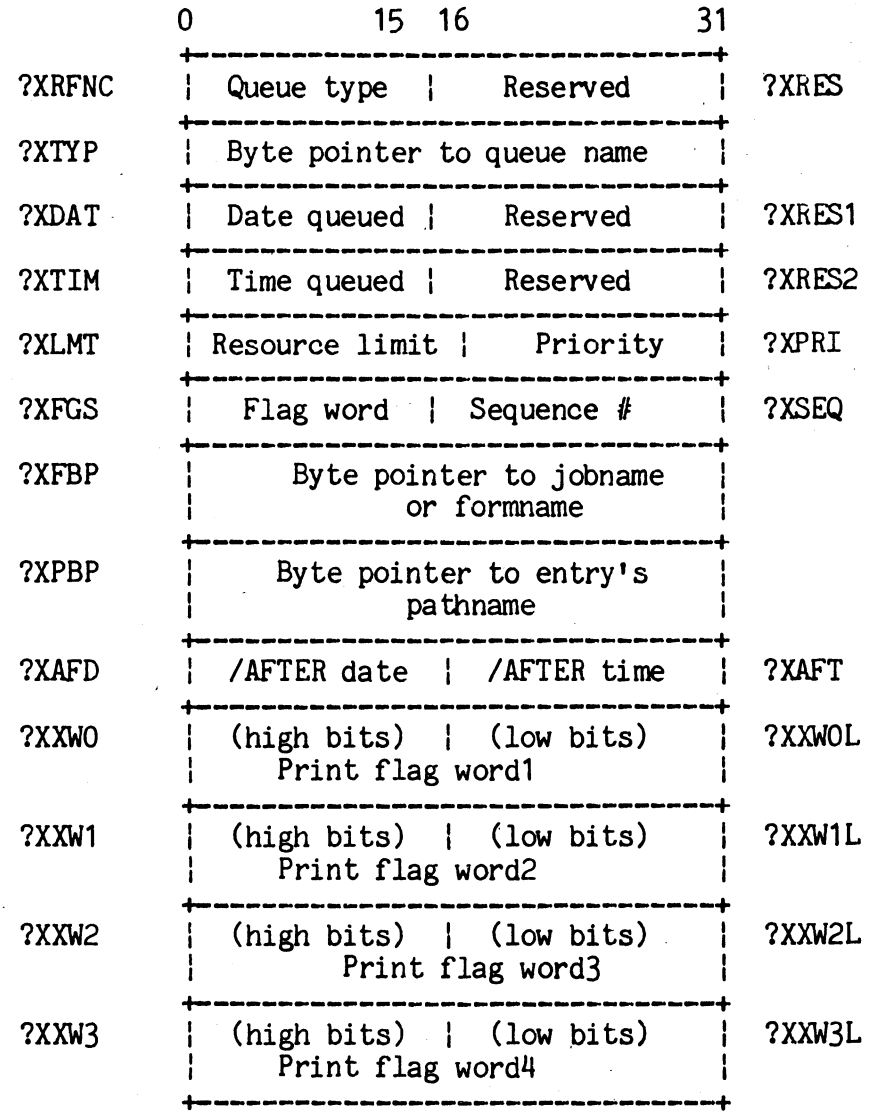
For a description of each commands function see the
"How To Run and Generate AOS/VS on your Eclipse MV/FAMILY Computer"
Manual # 093-000243-02.

Attachment 'B' - The ?EXEC Function Codes

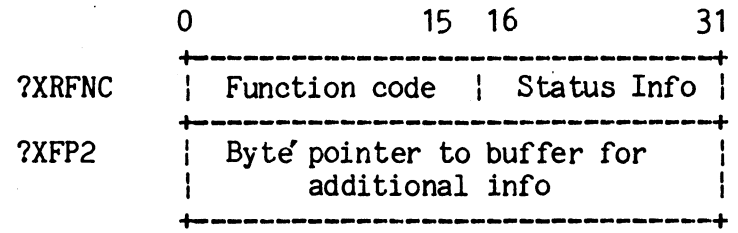
Code Value	Symbol	Function
1	?XFMUN	Mount a tape on a drive
2	?XMLT	Mount a labeled tape on a drive
3	?XFDUN	Dismount a drive or a labeled tape on a drive
4	?XFOTH	Submit a batch job for another user
5	?XFSUB	Submit a batch job
6	?XFLPT	Submit a print job
7	?XFPTP	Submit a paper tape punch job
10	?XF10R	Reserved
11	?XFPLT	Submit a plot job
12	?XFHAM	Submit a HAMLET file
13	?XFSNA	Submit a SNA/RJE file
14	?XFFTA	Submit a FTA file
15	?XFXUN	Extended mount a tape on a drive
16	?XFXML	Extended mount a labeled tape on a drive
17	?XFHOL	Hold a queue entry
20	?XFUNH	Unhold a queue entry
21	?XFCAN	Cancel a queue entry
22	?XFSTS	Get status information from EXEC
23	?XFQST	Get the queue type from the queue name
24	?XFLO	Labeled tape open
25	?XFLC	Labeled tape close
26	?XFME	Mount error
27	?XFNV	Mount next volume
30	?XF30R	Reserved
31	?XFSV	Mount specific volume

Attachment 'C' - The ?EXEC system call packet

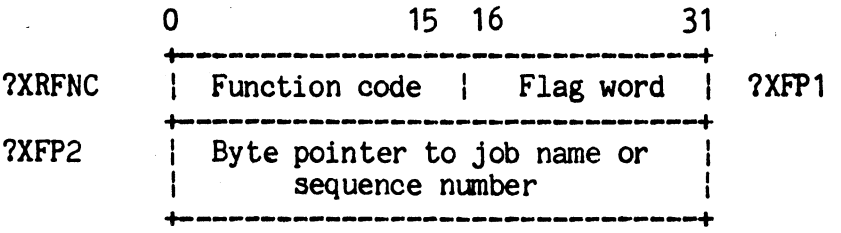
The following describes the ?EXEC system call packet for queue requests.



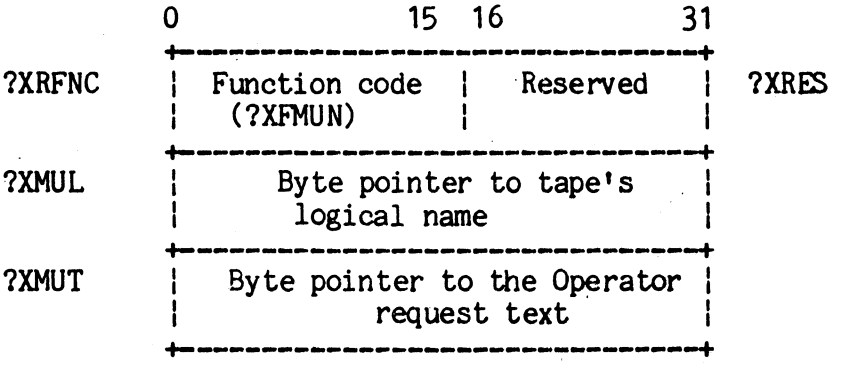
The following describes the ?EXEC system call packet for status requests.



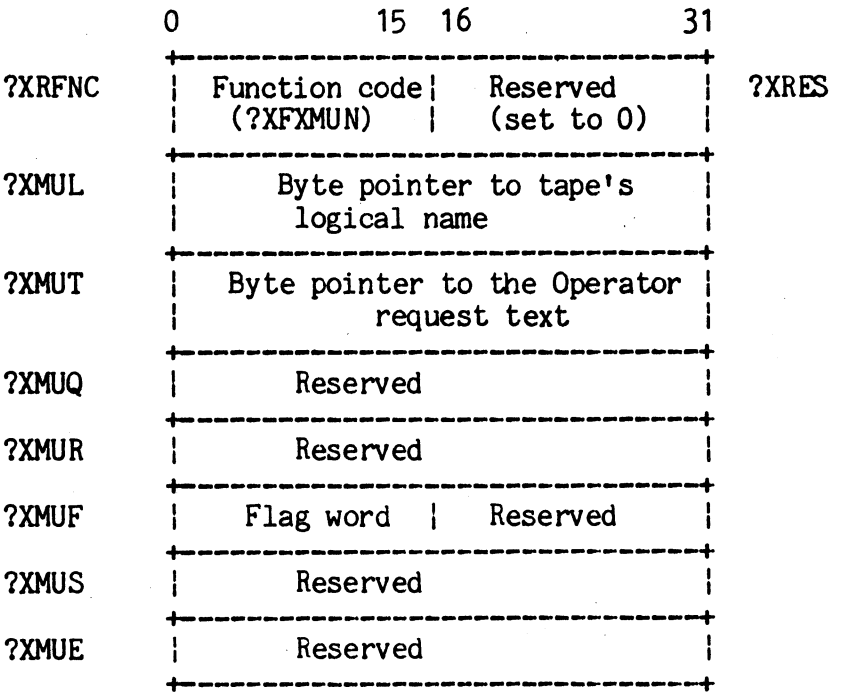
The following describes the ?EXEC system call packet for hold/unhold/cancel requests.



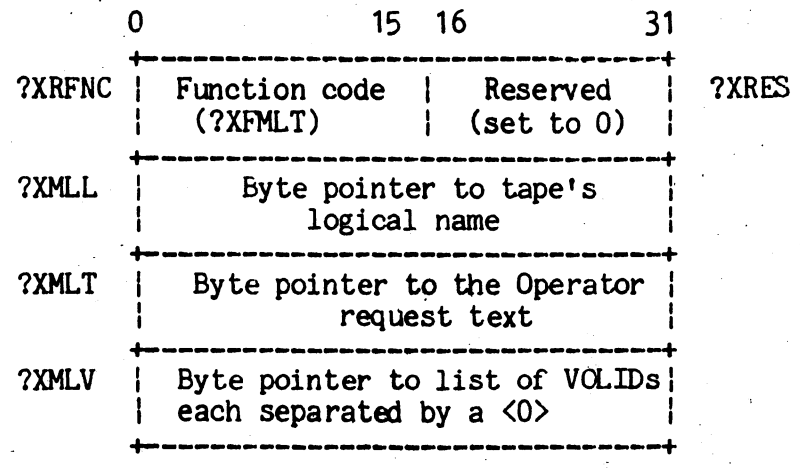
The following describes the ?EXEC system call packet for unlabeled mount requests.



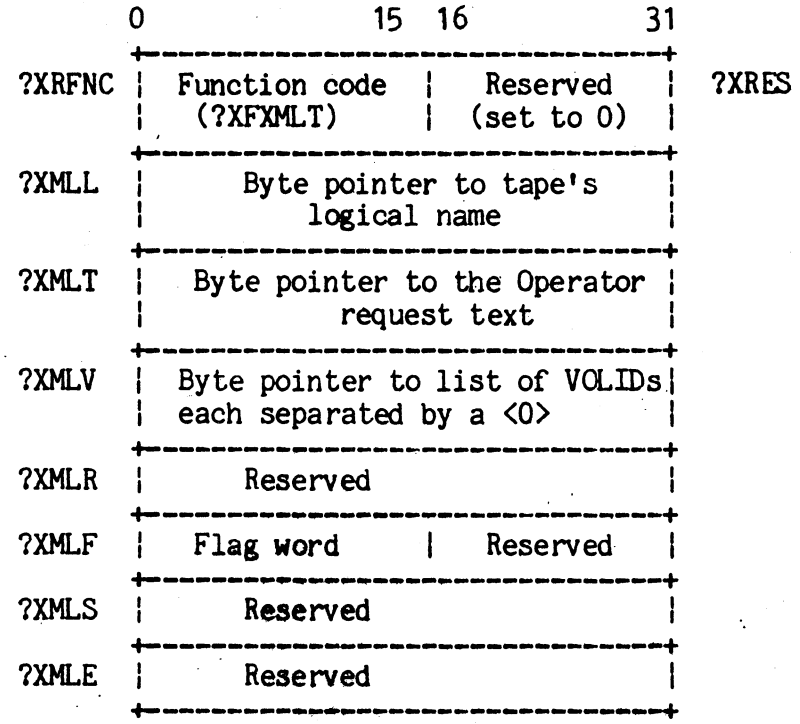
The following describes the extended ?EXEC system call packet for unlabeled mount requests.



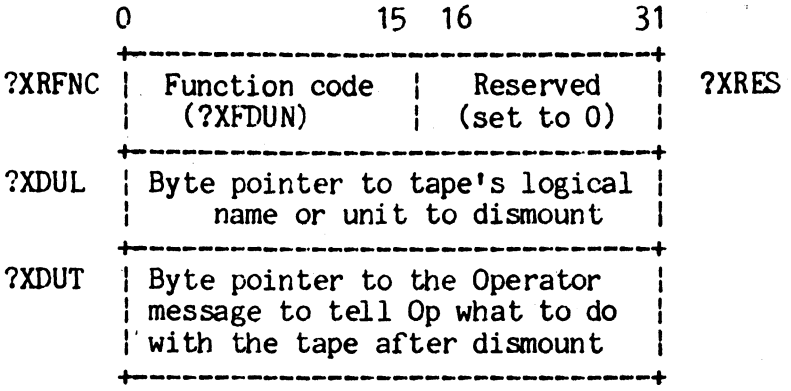
The following describes the ?EXEC system call packet for labeled mount requests.



The following describes the extended ?EXEC system call packet for labeled mount requests.



The following describes the ?EXEC system call packet for a dismount tape request.



Logon World

The Logon World's purpose is to allow users access to the AOS/VS operating system in a secure, orderly, and flexible manner. To accomplish this, the Logon World is given the tasks of: assuring systems security; controlling, monitoring, and logging console activity; and verifying user's access privileges.

Features

System Security

(1) Each user must have a PROFILE to log on. These profiles exist as files in :UPD and have names corresponding to the user's name. (Profiles are created/modified using the Preditor utility).

(2) User's must know a valid USERNAME/PASSWORD pair to gain entry to the system.

(3) User's must have the correct privilege to logon. (e.g. USE CONSOLE, MODEM, USE VIRTUAL CONSOLE, etc).

(4) Actions taken at a console when unsuccessful logon attempts occur can be specified by the operator. Using switches on the CX ENABLE command line the operator can specify that a console is to be disabled in the event of an unsuccessful logon attempt(/STOP) or re-enabled after a ten second pause(/CONTINUE or by default).

(5) The number of unsuccessful logon attempts necessary to trigger the actions described above can be modified using the CX ENABLE command and the /TRIES=n switch (10 >= n >= 1).

(6) The operator can change the enable parameters for a previously enabled console using the /FORCE switch and the CX ENABLE command. With this feature the operator can run a macro at night that puts all consoles at a high level of security without requiring users to logoff.

(7) Consoles disabled due to invalid logon attempts are reported at the op console and recorded in EXEC's log provided that EXEC logging is enabled.

Personalized Profiles

(1) Each user has a personalized profile that provides the ability to specify exactly what privileges and system resources each user may access (e.g. SUPERUSER, SUPERPROCESS, ACCESS DEVICES, MAXIMUM :UDD SIZE, MODEM, CONSOLE, VCON, etc; complete profile content and format will be discussed later in this chapter).

(2) Individuals with SUPERUSER, usually the operator, can use the PREDITOR utility to create, modify, and delete user profiles. (EXEC and PREDITOR alone may access user profiles directly).

Centralized Console Control

By using PREDITOR to define user profiles and various CONTROL @EXEC commands, the operator may do the following:

(1) ENABLE/DISABLE consoles for logon (CX ENABLE/DISABLE commands).

(2) Set the logon parameters for any one or all consoles (CX ENABLE command and switches).

(3) Decide how and if users may logon (PREDITOR utility).

(4) Decide which users may access which devices (PREDITOR utility).

(5) Determine the state of any console (CX CONSOLESTATUS command).

(6) Terminate any user job running at an EXEC console (CX TERMINATE command).

Logging of User Activity to :SYSLOG

EXEC logs several types of entries to the system log :SYSLOG. These entries can be useful for accounting and billing purposes, system usage monitoring, and security violation tracking.

Message Types

(1) Console Connect Time - This entry is logged whenever the user job running at an EXEC console is terminated. The log entry includes the username, console, and elapsed time since the initial logon occurred.

(2) Privileged User Logon - This entry is logged whenever a privileged user logs on. A privileged user is anyone who has one of the following privileges: SUPERUSER; SUPERPROCESS; or ACCESS DEVICES. The SYSLOG entry includes the username and the date and time of the logon.

*** The next two codes are not directly logged by the Logon World but have been added here for completeness. ***

(3) Pages Printed - The pages printed entry is logged whenever a user logs off. The log entry contains the username, device, and the number of pages printed at that device.

(4) Unit Usage - EXEC enters a unit usage entry whenever some type of media is used under EXEC's mount facility. The entry indicates the username, device name, and elapsed time of usage.

Unlimited Number of Consoles Under EXEC

EXEC will support an unlimited number of consoles. However, limits on this number are set at link time in accordance with AOS/VS limits.

```
Current Limits:
        local consoles      = 256
        virtual consoles    = 80
        total(max_consoles) = 336
```

Data Bases

Many EXEC data bases are specific to the Logon World and therefore reside in ring six. Most are listed and described here and have been separated into two groups: Support Data Bases and Primary Data Bases. Use these descriptions as a reference when reading the sections to follow. They describe how these data bases interact with the various Logon World specific tasks providing the features listed previously.

Support Data Bases

- Username Hash Table(UHT)
- Pid Array(PA)
- Profiles
- Profile Descriptors(PDs)

Primary Data Bases

- Logon Descriptors(LDs)
- Logon Local Port Table(LLPT)
- Logon Hash Table(LHT)
- VCON Logon Hash Table(VLHT)
- Input Descriptors(IDs)
- Console Driver Input Queue(CDIQ)

Support Data Bases

Username Hash Table

The Username Hash Table has a total of 47 entries. Each entry, when in use, is a two-word pointer to a Username Descriptor. The Username Descriptors will not be fully described here. It will suffice to say that they contain enough information to identify individual users, indicate the total number of jobs currently running on the system with that username, and contain a special link field whose purpose will be described later.

When EXEC wishes to determine whether a username is common to more than one job, the username is hashed as follows;

```
do count = 1 to length(user_name);
    hashed_value = hashed_value +
                   rank(substr(user_name,1))+1;
```

This hashed value is passed through a MOD function together with the HBOUND of the Username Hash Table. The result is a slot which will point to the Username Descriptor should there be another job currently on the system with that username.

Certainly there can be more than 47 unique usernames associated with processes under EXEC at a given time. It was decided however, that it would not be desirable to allocate a static data base with slots for nearly every username possible as the size would be ridiculously large. Instead, we assume that collisions will occur when hashing usernames into the table. When these collisions occur, the link field in the Username Descriptor whose address is already pointed to by the table entry is filled in with the address of the Username Descriptor whose hashing caused the collision.

This avoids having to allocate additional static space for the Username Hash Table. The overhead involved in maintaining this structure is not great as a given entry is only accessed twice during an EXEC son processes lifecycle.

Pid Array

The Pid Array initially has 64 entries which are allocated at initialization time. Each entry, when in use, is a two-word pointer to a Pid Descriptor. A Pid Descriptor contains the address of the Logon Descriptor associated with the pid, the pid type (console or other), and a special link field (described later).

Entries are filled when a user is ?PROCed. The resultant pid is used as a direct index into the table.

The value of the pid after EXEC's ?PROC can certainly be greater than 64. Therefore the table is dynamic; it can grow to a maximum dimension of 512 pids, EXEC's current limit (Note that the number 512 anticipates a growth from AOS/VS' limit of 256 pids). The way it grows is rather unique. EXEC will grow the table in chunks. It compares the current number of table entries plus the size of the growth chunk (currently 32 entries), with the pid itself and takes the greater of the two values. Then EXEC allocates two times the new value words(pointers). In effect, EXEC allocates a new table of zero entries. Then the old table is copied into the first section of the new table and the old copy is freed. This avoids maintaining a static data base with 512 two-word entries. One obvious problem is that since EXEC's son's pids are usually far from sequential, and since these pids are used to access the Pid Array, there is a high likelihood that empty slots will exist in the table. These slots are cleared and the Pid Descriptors released when the pid terms.

To add to the confusion, it is possible that the Pid Descriptor for a pid that is termed may not be freed before another EXEC son with that pid is released. In this case, the link field in the Pid Descriptor is used. The newest Pid Descriptor is always at the head of the chain with older descriptors linked to it. Therefore, when EXEC goes to release the Pid Descriptor associated with a given pid it must check to see if there are multiple descriptors with a common pid and release the oldest descriptor.

Profiles and Profile Descriptors

User Profiles exist as files in :UPD. These entries are created by the Preditor utility, one for each user on the system. Specific user information is stored here such as: privileges; username; password; max priority; etc.

EXEC is the only other process that should access the User Profiles. EXEC reads in the Profile information directly from :UPD and stores the information it requires in a Profile Descriptor. This descriptor is allocated during the Logon process and is deleted once the user is successfully ?PROCed. EXEC needs the profile information to:

(1) Validate the USERNAME/PASSWORD pair.

(2) Confirm that user has appropriate logon privileges (e.g. MODEM, CONSOLE, VCON).

(3) Acquire the users intial program for ?PROCing purposes.

(4) Set up process memory sizes.

(5) Get the users initial process priority.

(6) Set up maximum number of sons for user's process.

(7) Set working set min and max.

(8) Establish Max :UDD:USER size.

(9) Save user privilege bits and batch priority in Logon Descriptor.

In addition, EXEC must write information to the users profile. His "last previous logon" time must be updated and his new password if one was given. Here EXEC makes the changes in the profile buffer and then writes the information back to the :UPD file.

Primary Data Bases

Logon Descriptors

Logon descriptors are allocated, one for each console, at enable time. They may exist in unlimited numbers (limits set at link time, equal to the maximum number of consoles). These descriptors can be very large since they contain the consolename which may be as long as the maximum pathname length. But since the consolename is usually quite short (e.g. CON19), EXEC calculates the actual length of the consolename and allocates just enough space to accomodate it and the rest of the descriptor.

We can get away with this because the descriptors are never reused - the console name can never change.

We allocate extra room for a null to appease our data sensitive operating system.

The actual formula used to determine the necessary number of words is:

```
WORD LENGTH FULL_DESCRIPTOR -
WORD LENGTH OF VARYING PART +
       CONVERT TO WORDS (BYTE LENGTH OF NAME + ONE EXTRA BYTE +
                         ONE TO HANDLE ODD BYTES) + ONE WORD FOR
                         LENGTH OF STRING.
```

The Logon Descriptor for a given console is released when the console is closed and deassigned at the end of the disable sequence.

Logon Descriptor Format

```
%replace states_to_remember by 8;

declare 1 logon_descriptor based,
          5 node_type                  binary(15) fixed,
          5 control_port               bit(32) aligned,
          5 read_port                  bit(32) aligned,
          5 write_port                 bit(32) aligned,
          5 console_number             binary(15) fixed,
          5 current_state              binary(15) fixed,
          5 pid                        binary(15) fixed,
          5 soft_priv_bits,
            10  unused                 bit(10),
            10  change_pw              bit(1), /* true if 0 */
            10  remote_resource        bit(1), /* true if 0 */
            10  virt_cons_priv         bit(1), /* true if 0 */
            10  modem_priv             bit(1), /* true if 1 */
            10  batch_priv             bit(1), /* true if 1 */
            10  console_priv           bit(1), /* true if 1 */
          5 batch_priority             binary(15) fixed,
          5 i_o_buff                   pointer,
          5 profile_channel            binary(15) fixed,
          5 profile_buffer_addr        pointer,
          5 logon_try_count            fixed binary(15),
          5 logon_try_max              fixed binary(15),
          5 connect_time               binary(31) fixed,
          5 modem_ercds_counter        binary(15) fixed,
          5 virtual_console            bit(1),
          5 log_on_off_in_progress     bit(1),
          5 use_form_feed_banner       bit(1),
          5 timeouts_enabled           bit(1),
          5 timeout_occurred           bit(1),
          5 read_profile               bit(1),
          5 svta_died                  bit(1),
          5 input_when_svta_dead       bit(1),
          5 requests_pending       bit(1),
          5 disable                    bit(1),
          5 open_complete              bit(1),
          5 close_complete             bit(1),
          5 read_complete              bit(1),
          5 write_complete             bit(1),
          5 set_timeout_complete       bit(1),
          5 enable_timeouts_complete bit(1),
          5 earlier error_flag         bit(1),
          5 big_i_o_buffer             bit(1),
          5 continue_logon             bit(1),
          5 svta died_error_code       binary(15) fixed,
          5 device_chars,
               10 st bit(1),
               10 sff bit(1),
               10 epi bit(1),
```

```
          10 unused bit(1),
          10 spo    bit(1),
          10 raf    bit(1),
          10 rat    bit(1),
          10 rac    bit(1),
          10 nas    bit(1),
          10 ott    bit(1),
          10 eol    bit(1),
          10 uco    bit(1),
          10 lt     bit(1),
          10 ff     bit(1),
          10 eb0 bit(1),
          10 eb1 bit(1),
          10 ulc bit(1),
          10 pm  bit(1),
          10 nrm bit(1),
          10 modem   bit(1),
          10 device_type bit(4),
          10 timeouts    BIT(1),
          10 tsp BIT(1),
          10 pbn BIT(1),
          10 esc BIT(1),
          10 wrp BIT(1),
          10 fkt BIT(1),
          10 reserved bit(2),
          10 lines_per page BIT(8),
          10 line_length   BIT(8),
      5 state_history,
        6 last_state       binary(15) fixed,
        6 previous_states(states_to_remember),
          7 state          binary(15) fixed,
          7 input_type     binary(15) fixed,
      5 enable_request       pointer,
      5 pended_request_queue pointer,
      5 user name      character (?MAX_USERNAME_LENGTH) varying,
      5 device_name    character (?MAX_PATHNAME_LENGTH) varying;
```

Logon Local Port Table

The Logon Local Port Table has one entry for each possible Logon Descriptor (currently 336). Each entry, when in use, is a two-word pointer giving the address of a Logon Descriptor. Entries are filled in at enable time. As CONx is enabled, it is assigned its unique internal console number "n" and its Logon descriptor is allocated. The address of the LD is then stored in the 'n'th entry in the Logon Local Port Table. Later, when EXEC needs to find the Logon Descriptor for a given console, and it has access to that console's unique ID, it can find that LD using the ID as an index into the Logon Local Port Table.

This ID is available and as such, the table can only be used to find the appropriate Logon Descriptor for a console, when an EXEC initiated IPC comes home. The entire sequence is as follows:

(1) EXEC initiates IPC on behalf of a console with the unique console number and an action code in combination as the local port number (port number = 3 * console number (+1 if read port) (+2 if write port)).

(2) IPC returns and the local port number is decoded by the appropriate IPC listener.

(3) Unique internal console number is then used to access the appropriate Logon Local Port Table entry.

(4) This entry gives the address of the appropriate Logon Descriptor.

Logon Hash Table

The Logon Hash Table currently has 427 entries. 427 being the first prime number greater than 400. 400 was used instead of 256 because it was not clear how many consoles we would have to support in the near future (VCONs are a special case as discussed later).

Each entry, when in use, contains the control port and the address of the Logon Descriptor for a given console. These entries, as are the Logon Local Port Table entries, are filled in at enable time. When a console is enabled EXEC ?ILKUPs its control port number. This number is then hashed giving a slot in the Logon Hash Table. If this entry is not in use, the control port and the address of the Logon Descriptor associated with it are laid down. If it was already in use, the next sequential available slot is used.

This table is used to find a local console's Logon Descriptor when EXEC receives an IPC that it did not initiate such as the IPC generated by the CX DISABLE @CON6 command. Here EXEC has only the console name. The steps that are then taken to find CON6's Logon Descriptor are listed below.

(1) EXEC receives an IPC that it did not initiate (e.g. CX DISABLE @CON6).

(2) EXEC does an ?ILKUP on the console name and obtains its control port.

(3) The control port is hashed using the following algorithm
SLOT = MOD (CONTROL_PORT HBOUND(LOGON_HASH_TABLE)+ 1)

(4) Collisions may occur so it is necessary to compare the control port at the indicated slot with the original. If they don't match, the search continues sequentially from that slot.

(5) Once the correct control port entry is found, the address of the Logon Descriptor can be obtained from the second field in that entry.

Vcon Logon Hash Table

As of AOS/VS rev 4.00, the VCON Logon Hash Table has 83 entries. The number eighty three was chosen because it is the first prime number greater than the maximum number of VCONs supported by EXEC (80).

Each individual entry, when in use, contains the control port and the address of the Logon Descriptor for a given console. These entries are filled and accessed in the same way as entries in the Logon Hash Table. The reason for the separation of the two data bases is that it is sometimes necessary to access all the Logon Descriptors that are associated with VCONs. With the separate VCON Logon Hash Table, EXEC doesn't have to search for VCONs in a table with as many as 256 non VCON entries.

It becomes necessary to access the LDs for VCONs collectively at two different times.

(1) When SVTA dies it is necessary to terminate the process running under EXEC at each VCON and to notify the operator, through the OP console, that the virtual console has been disabled. We do not release the LDs associated with the virtual console at this time. Instead the LD is kept around so that the operator can do a CONSOLESTATUS to determine the state of the VCONs that were just disabled. Also, EXEC needs the VCONs' LDs so that it can successfully process the terminations for the processes that were running under EXEC at the VCONs. Without the LDs EXEC would not be able to handle the term and an internal consistency error would be taken.

(2) When the operator attempts to re-enable a VCON, EXEC then frees up all the old LDs. Once again, it is necessary to go through the VCON Logon Hash Table to find each of the LDs to be freed.

Input Descriptors

Input Descriptors can exist in unlimited numbers and can be of a variety of sizes and formats. Each of the formats does have a common header which gives the information necessary to determine the type and size of the Input Descriptors. These descriptors are allocated and filled in by several different routines in ring 6 and are pushed (actually linked) onto the Console Driver Input Queue as input for the Console Driver Task.

Input Descriptors can be of any of the following types.

- Console Input Descriptors:
  - Assign Descriptors
  - Open Descriptors
  - Get Characteristics Descriptors
  - Set Timeout Descriptors
  - Write Descriptors
  - Set Characteristics Descriptors
  - Read Descriptors
  - Close Descriptors
  - Release Descriptors
  - Enable Descriptors
  - Disable Descriptors
  - Terminate Descriptors
  - Delay Descriptors
  - Timeout Error Descriptors
  - Disconnect Error Descriptors
  - Network Error Descriptors
  - General Error Descriptors
- SVTA Died Descriptors
- Term Descriptors

As was previously mentioned, each of these types of Input Descriptors can be put on the input queue for the Console Driver Task. This task must then check the input type and pass control to the appropriate action routine. The algorithm used to decode and dispatch in the Console Driver Task shall be discussed in the next section. For now, it is sufficient to note that each Input Descriptor on the Console Driver Input Queue represents an action that needs to be performed within the Logon World. The type of the input descriptor and the current state of the console effected, indicates that action. These descriptors are freed in the routines which perform the requested actions.

Console Driver Input Queue

The Console Driver Input Queue(CDIQ), is a dynamic queue that is made up of linked Input Descriptors. As Input Descriptors are allocated and pushed on the queue and subsequently released, the CDIQ will grow and shrink accordingly. There is no limit to the number of IDs that can be on the queue at one time. When entries are added and/or deleted the atomic queue instructions are used eliminating the need for queue locking.

Each ID that is in the queue is linked to ID before and after it. Note that the link fields in the ID point to the link field in adjacent ID and not to its head.

Inputs

As mentioned previously, EXEC's Logon World is completely driven by external requests and responses. That is, the Logon World only performs actions when triggered to do so by an external input. In this section we will see that these inputs come from five sources: The Operator; the PMGR; SVTA; the Delay Task; and AOS/VS. The next section will go into greater detail and describe what actions these inputs trigger as well as how the Console Driver Task manages multiple inputs.

(1) When the Operator issues CX ENABLE, DISABLE, TERMINATE, and CONSOLESTATUS commands, the CLI partially decodes the command line and fires off an IPC to @EXEC which eventually winds up in the Logon World.

(2) The operating system, AOS/VS, is another source of input to the Logon World. The termination listener task in ring 7 receives terminations from the system whenever the process running at an EXEC owned console terminates, a batch job terminates, or SVTA dies.

(3) The PMGR sends inputs to the Logon World in the form of responses to EXEC requests. The PMGR_LISTENER_TASK, puts up a global ?PREC which receives all responses coming to ring 6. EXEC initiates the sequence by requesting a write, read, control, etc. The response comes back in the form of a yes the action was successful, or no it was not successful.

(4) SVTA sends inputs to the Logon World in same manner as PMGR except these are responses to EXEC ?IREC requests for virtual console operations over the net.

(5) The Delay Task responds to EXEC requests also. EXEC requests a notification from the Delay Task when ten seconds have passed so that it knows when to wake up a console that was paused as a result of an unsuccessful logon attempt. The Delay Task responds when the ten seconds have expired.

The example section to follow will show that inputs from one source can perpetuate inputs from the other sources.

Design

State Machine

The implementation design called the "Finite State Machine" is well suited for input-driven action schemes. EXEC uses this design technique to manage console actions. EXEC's State Machine is driven by a 2-dimensional table. The first dimension is the consoles current state and the second is the input type.

Each console is always in a certain state. These states reflect where in the logon/logoff sequence the console is. The console's current state as well as a parameterized number of previous states are stored in its logon descriptor. These previous states are used for debugging the Logon World.

There are a total of 35 states along the first dimension of the table. They are:

```
/*
*    States.
**/

%replace wait_for_enable_state          by 1;
%replace wait_for_assign_state          by 2;
%replace wait_for_open_state            by 3;
%replace wait_for_get_char_state        by 4;
%replace wait_for_stoc_state          by 5;
%replace wait_for_write_banner_state    by 6;
%replace wait_for_timeout_enable_state  by 7;
%replace wait_for_read_new_line_state   by 8;
%replace wait_for_write_greeting_state  by 9;
%replace wait_for_write_userprmt_state  by 10;
%replace wait_for_read_username_state   by 11;
%replace wait_for_write_pwprmt_state    by 12;
%replace wait_for_read_password_state   by 13;
%replace wait_for_write_timeout_state   by 14;
%replace wait_for_write_invalid_state   by 15;
%replace wait_for_write_5attempts_state by 16;
%replace wait_for_write_newpwprmt_state by 17;
%replace wait_for_read_newpw_state      by 18;
%replace wait_for_write_newpwmess_state by 19;
%replace wait_for_write_loghead_state   by 20;
%replace wait_for_write_logmess_state   by 21;
%replace wait_for_write_logtrlr_state   by 22;
%replace wait_for_disable_to_state      by 23;
%replace wait_for_termination_state     by 24;
%replace wait_for_logoff_banner_state   by 25;
%replace wait_for_disable_banner_state  by 26;
%replace wait_for_close_state           by 27;
%replace wait_for_deassign_state        by 28;
%replace disable_in_progress_state      by 29;
```

```
%replace wait_for_disable_timeout_state by 30;
%replace svta died_state                by 31;
%replace bad_state                      by 32;
%replace get_from_action_state          by 33;
%replace get from_table_state           by 34;
%replace free logon_descriptor_state    by 35;
```

There are a total of 18 different inputs making up the second dimension of the table. A list of the most recent input types for a given console are also stored in its Logon Descriptor. By matching these recorded inputs with the states that they put the console in, it is possible to walk back through a consoles actions in search of unexpected sequences. The 18 states are:

```
/*
*    Console Input Types
**/

%replace open_input                 by 1;
%replace assign_input               by 2;
%replace get_characteristics_input  by 3;
%replace set_timeout_input          by 4;
%replace write_input                by 5;
%replace set_characteristics_input  by 6;
%replace read_input                 by 7;
%replace close_input                by 8;
%replace deassign_input             by 9;
%replace enable_input               by 10;
%replace disable_input              by 11;
%replace terminated_input           by 12;
%replace delay_input                by 13;
%replace svta terminated_input      by 14;
%replace timeout_error_input        by 15;
%replace disconnect_error_input     by 16;
%replace network_error_input        by 17;
%replace general_error_input        by 18;

%replace num_console_inputs         by 18;
```

When EXEC receives an input for a console, it indexes into the State Table using the console's current state and the input type and finds the appropriate entry. Each entry in the State Table contains the action to perform on the console and a number indicating the next state to put the console

For example:

(1) CON7 currently has the logon banner on the screen (***press newline***)

(2) EXEC has issued a READ request to the PMGR to read any newline entered at CON7.

(3) CON7 is currently in the "WAIT FOR NEWLINE" state and this value is stored in CON7's Logon Descriptor.

(4) When a user hits newline at CON7, the READ request comes home from the PMGR.

(5) EXEC interprets the message as a READ input for CON7.

(6) At this point EXEC indexes into the State Machine's table using the current state(WAIT_FOR_READ_NEWLINE) and the input (READ).

(7) In the corresponding table entry, EXEC finds the action to perform on the console(WRITE_GREETING_LINE) and the next state to put the console in (WAIT_FOR WRITE_GREETING_LINE).

(8) EXEC puts up a request to the PMGR to perform the write action and updates the current state field in CON7's LD. This eventually results in the greeting line being sent to CON7 ("AOS/VS 4.00 / EXEC 4.00 .....).

There are many advantages to this State Machine implementation.

(1) Easy to implement and test action routines, (actions are isolated from each other).

(2) Easy to modify the Logon World's behavior. Action routines can be easily modified as can entries in the State Table.

(3) Timeout errors handled without difficulty. Errors are not handled as special cases but simply as inputs to the State Machine. If a timeout error occurs when EXEC is waiting for a user to enter his/her password, EXEC simply treats this as a "TIMEOUT INPUT" and indexes into the table as it would normally.

(4) Better validation of inputs/states. By definition, every element of the table must have an entry. All invalid or unexpected combinations of states and inputs have table entries that direct EXEC to perform an error action. For example, if CON5 is in the "WAIT_FOR_READ_NEWLINE" state, it is clearly incorrect for EXEC to receive a "WRITE" input from the PMGR. In this case, EXEC indexes into the table as it normally would. However, the action specified directs EXEC to take an "INTERNAL CONSISTENCY ERROR". This indicates that some portion of EXEC is seriously ill and processing cannot continue. Here again, the State Machine design forces the programmer to consider an action for every combination of states and inputs.

(5) Better debugging information. EXEC saves a number of states and inputs for each console in its LD. This saved info provides a state history for the console and is quite useful in the debugging process. For example, in the previous paragraph we saw an error caused by the receipt of an unexpected input. When EXEC took the INTERNAL CONSISTENCY ERROR, a ?MDUMP of rings 6 and 7 is triggered. We can then look through the dump and easily find the console that caused the error by examining the consoles state history. The cause of the problem can then be narrowed down to certain areas in EXEC, the PMGR, etc.

Console Driver Task

As previously mentioned, EXEC's Logon World receives input from a variety of sources. For effective ordering/handling of these inputs, EXEC maintains a the Console Driver Input Queue. Several other tasks listen for these inputs and determine if they are requests or responses for the Console Driver Task. If they are, the inputs are formatted into Input Descriptors and linked to other IDs forming the CDIQ.

The console driver task pulls requests off the CDIQ and performs the desired action(s) on the appropriate console(s). The following outline describes the general actions performed by the console driver task.

LOOP: (1) Pop an entry off the input queue (INPUT_QUEUE).

(2) Determine the type of input (3 major types). The type is stored in the input descriptor.

Console Input
Termination Input
Svta Died Input

(3) Dispatch to the correct code for each input:

(a) Console Input - Break down further into the type of console input. Again, the type of console input is also stored in the input descriptor.

(i) Enable Input

- Determine if the enable command is for all consoles or just 1
- For every console that is to be enabled:

- Allocate/initialize a logon descriptor if one doesn't exist
- Call STATE_MACHINE to perform the appropriate action for the console.

(ii) Disable Input

- Determine if the disable is for all consoles or just 1

- Call STATE_MACHINE to perform the appropriate action for each console

(iii) Control Input (input other than enable or disable)

- Find the logon descriptor associated with the input
- Call STATE_MACHINE to perform the correct action

(b) Termination Input

- Determine which console the termed pid is running on
- call STATE_MACHINE for that console with the termination input

(c) Svta Died Input (the net died)

- Send a message to OP (Disabling VCONS due to SVTA termination)
- Loop through all the consoles and for every VCON, call STATE_MACHINE to perform the appropriate action
- Disconnect from SVTA
  - Free up the message dispatcher task if it is pended on SVTA

(4) Free up the input descriptor, if appropriate.

(5) Go to the top of the loop and process another input.

When STATE_MACHINE is called as part of the above algorithm it does the following:

(1) Saves the logon descriptor we are working on

(2) Validates logon descriptor and input descriptor pointers

(3) Save the current state and the input received (Each logon descriptor has within it the last N states it was in and the last N inputs it received -> used for debugging and analysis. The value for N is set at link time).

(4) Perform the appropriate action on the console. The logon descriptor has a 'current state' value within it. STATE_MACHINE indexes into a table using the current state value and the input value to determine what action to perform. The appropriate action routine is executed for the console.

(5) Now that the STATE_MACHINE has performed the action, it must update the console's current state. The current state can come from the state table or the action routine itself.

In addition to identifying the action to be performed, the state table also includes the next-state value for the console. Normally, the state machine can simply take this value and use it as the current state in the logon descriptor.

Sometimes, however, the table cannot determine the next state for the console. In these cases, the table entry contains the value 'get_from_action_state' (instead of an actual state). When this happens, the STATE_MACHINE will use the next-state returned by the action routine.

(For example, when validating the user's password, the table cannot determine what the next-state for the console will be because it does not know if the password is correct or not. In this case, the validate password action routine must return the console's next-state).

(6) If the SVTA-died bit is set after the action is performed, then place the console in the 'svta_died_state'.

(7) If the new current state for the console is 'free_logon_descriptor', then do just that! Free up all of the console's memory and release the logon descriptor.

Design Advantages

There are many advantages that come with the Console Drive Input Queue and Task design.

(1) All inputs from the various sources are collected in one place. Therefore the Console Driver Task need only listen for inputs from one source instead of many.

(2) EXEC buffers its input messages internally instead of depending on the systems IPC spool file mechanism.

(3) The Console Driver Task never has to pend directly on external inputs. The listener tasks do the pending while the CD task only pends waiting for internal inputs.

(4) The structure of the input queue provides a simple way for EXEC to multiplex among many consoles. The Console Driver Task services the inputs in the order that they are placed on the queue. Since the various inputs are for different consoles, the CD task can be logging on/off many consoles at once.

As mentioned above, the console driver task is not pended directly on external events. When EXEC wants to send a message to a console, an action routine initiates an ?I/P-SEND but does not put up an ?I/P-REC. Instead, EXEC relies on the listener task to receive the responses/requests that are aimed at the Logon World. The CD Task is then free to process additional inputs.

There are advantages of this unpended design.

(1) The Console Driver Task need not hang even if a console does. If for some reason a communication to/from the PMGR/SVTA is lost or pended, only that console will be hung. The Console Driver will continue processing all other consoles on the system.

(2) Consoles requests are completely independent of time. EXEC does not care how long a request takes to complete.

Pended Request Queue

In certain situations, the Console Driver Task cannot perform an action that is requested until another action completes. For example, if the operator issues an ENABLE and a DISABLE command for the same console, EXEC cannot perform the DISABLE until the ENABLE completes.

In these situations, EXEC queues up the request on a 'pended request queue'. The pended request queue hangs off of the logon descriptor and is specific to one console. Certain of the specific action routines have knowledge of this queue but the global console driver task mechanism does not. Thus, if a console is in the 'WAIT_FOR_ASSIGN' state (state 3) and a disable comes in for that console (input type 11), the state table indicates that EXEC should perform the 'PEND_REQUEST_ACTION' (action 41). The disable request will be placed on a queue associated with that logon descriptor. The console's current state is not changed when a request is pended.

Appropriate action routines will check the pended request queue to see if further actions are necessary. In the above case, when an assign (input 2) comes home for the console, the Stat_Machine will execute the 'OPEN_ACTION' (action 3). After performing the action for the assign input, the open_action routine will check the pended request queue and perform any action(s) necessary to fulfill those pended requests.

Examples

Enabling Consoles

The following is a list of the actions EXEC must perform in order to ENABLE a console (in this case CON7) in response to the command CX ENABLE @CON7.

(1) The control @EXEC listener task in ring 7 receives an IPC @EXEC containing the string "ENABLE @CON7".

(2) Ring 7 does some validation of the command string and passes control through the Stub Handler to the ENABLE command module in ring 6 (the ring 7 activity will be fully described in the class on the CX World).

(3) In the ENABLE module an input descriptor of the "console input type" is allocated and initialized. The various fields of the ID contain all necessary information about the command or pointers to it.

(4) The new ID is pushed onto the bottom of the CDIQ.

(5) The Console Driver eventually pops the ID off the CDIQ.

(6) The ID is decoded and it is determined that a console ENABLE was requested.

(7) EXEC acquires the consolename using the ID and does an ?ILKUP to get its control port.

(8) A ?GPORT is done using the control port to determine who the console belongs to.

(9) Once it has been confirmed that the console belongs to the PMGR/SVTA (PMGR in this case), the logon descriptor for this console must be found. The control port is hashed giving and index into the Logon Hash Table.

(10) If this is not an attempt to re-enable a console there should be no LD currently allocated for the console. If this is true a new LD is allocated and initialized and an entry is made in the LLPT and the LHT.

(11) Now everything is initialized and a call is made to the state machine with the console's current state having been set as "WAIT_FOR ENABLE".

(12) The state machine indexes into the state table using the current state of the console,"WAIT_FOR_ENABLE", and the input type, "ENABLE".

(13) The table entry indicates that the action to be performed is the "ASSIGN ACTION".

(14) Control is passed to this action routine where a request is put up to the PMGR to ASSIGN the console to EXEC.

(15) Next the action routine indicates that the console should be put in the "WAIT_FOR_ASSIGN" state.

(16) The PMGR listener task in ring 6 then receives the response from the PMGR.

(17) The local port is decoded giving the console's number and an action code which indicates whether this is a response to a write, read, or control request. In this case it is a control response (the console number and the action code were used to form the local port when EXEC made the assign request).

(18) The console number is then used as a direct index into the Logon Local Port Table and the console's LD is accessed.

(19) Next an ID is allocated and initialized indicating that an ASSIGN request has been fulfilled. The ID is pushed onto the CDIQ for the Console Driver.

(20) As before the Console Driver pops the ID off the CDIQ and decodes it.

(21) The state machine once again gives the next action and state of the console.

(22) This EXEC/PMGR interaction continues until all the following actions are performed on the console:

- o OPEN - Here the console is readied for I/O operations and its read and write ports are acquired and stored in its LD.

- o GET CHARACTERISTICS - Used later (Modem, Hard Copy).

- o SET TIMEOUT CONSTANT - The timeout constant for read actions is set to 30 seconds (This is for USERNAME/PASSWORD reads).

o WRITE LOGON BANNER - EXEC gets the SYSID when it comes up and stores it in the banner string. Dim mode is set. the screen is cleared and the banner is written out.

o READ NEWLINE - Read is put up pending user's newline (note that EXEC does not pend here but processes other inputs pertaining to other consoles.

At this point the console has been successfully enabled.

Logging On

When the user types a newline to the enabled console, he triggers a furthur series of requests and responses between EXEC and the PMGR which result in various actions. These actions generally occur as follows.

o WRITE GREETING LINE - Operating system and revision information are combined with the date, time, and console-name and printed at the console.

o ENABLE TIMEOUTS - Timeouts enabled so that requests for USERNAME and PASSWORD pairs will timeout after 30 seconds (note that setting and enabling timeouts are two separate actions.

o WRITE USERNAME PROMPT

o READ USERNAME

o WRITE PASSWORD PROMPT

o READ PASSWORD - No echo on this read.

o VALIDATE USERNAME/PASSWORD - User's :UPD profile is read in and used for the validation. If correct pair entered EXEC checks to make sure user has proper privileges to logon on. If incorrect pair, then the prompt is reissued unless the user has exceeded the number of logon tries permissable at the console.

o DISPLAY LOGON MESSAGE - (If it exists) The first 512 bytes of of the file LOGON.MESSAGE is displayed (the file must exist in the same directory as EXEC's program file. EXEC keeps 512 byte buffers worth of logon message around and only reads a new one if the file has been modified since the last time it read it.

o DISPLAY LAST PREVIOUS LOGON DATE AND TIME - This information is available in the user's profile.

o ?PROC THE USER - The user's process is ?PROCed using the privileges and settings specified in her/his profile. Control is passed to the user's program. Temporary console memory such as the profile buffer and I/O buffer is released. Last previous logon information is updated in the user's profile and finally a :SYSLOG entry is made if the user possesses SUPERUSER, SUPER-PROCESS, or ACCESS DEVICES privileges.

Now the user is logged on and running the initial program specified in his/her profile. EXEC does not regain control until this process, EXEC's son, terminates for any of several reasons.

Termination/Logging Off

When the user's program which is running as a son of EXEC terminates, the operating system sends an IPC message to the Term Listener Task in ring 7. This message is partially decoded and passed through to ring 6. This initiates EXEC's logoff actions. Also, note that the PMGR returns control of the effected console to EXEC when the user's program terminates.

The following is a brief list of the major actions that are taken as a result of this termination notification.

(1) The ring 6 Process Termination routine receives notification that one of EXEC's processes has termed. The term code is then used to determine what type of term it was.

(2) Next, an ID is allocated, initialized appropriately and pushed on the CDIQ.

(3) The Console Driver Task then pops the ID off the CDIQ and processes it.

(4) Having determined that this ID is of the "termination" variety, the Console Driver calls the state machine with the console's current state, (which should be "wait for termination"), and the input type, ("termination").

(5) The State Machine then indexes into its table using these two fields and finds the next action should be the "write logoff action". This routine is passed control.

(6) Here a message buffer is allocated and various pieces of text are placed in it. The first message placed there indicates why the console job is terming

(7) To this is added the pid information as well as the connect time.

(8) The Username Hash Table is then indexed to determine if other EXEC sons with the same username are running on the on the system. If there are then this info is added to the buffer.

(9) Various other things such as the consolename, username, and a time stamp are added and a write request is issued on behalf of the console to the PMGR.

(10) EXEC then logs the termination to :SYSLOG, removes this pid's entry in the Pid Array, and either decrements the count of jobs in the Username Descriptor for this username or deletes the UD if this was the only job with that username.

This sort of communication between the PMGR and EXEC continues from this point until the following actions complete.

- o The Console is Closed - incidental console I/O buffers are released and the console is closed.

- o The Console is Deassigned - If the console is simply terming, as in this case, it is not actually deassigned, but instead reopened (if the console was to be disabled, it would be deassigned and its LD as well as LLPT and LHT entries would be released).

Once these actions are completed, barring any errors or disable attempts, the logon banner is rewritten at the console and it is returned to the "wait for newline" state.

Special Cases

Modems

The characteristics data that EXEC gets from the PMGR indicates whether a console is genned as a modem or not. If it is a modem it will be subject to several special case actions.

(1) EXEC will enable timeouts while waiting for the user to type newline instead of waiting and enabling them in time for the username/password prompts.

(2) The user must have modem privilege to log on.

(3) When the user's job terminates, the console is closed and the PMGR breaks the connection on the line. EXEC cannot immediately re-open the console in this case and throw up a new banner. EXEC must first wait while the PMGR holds the line for 15 seconds to clear noise. Then the console can be re-opened and the banner will be available to the next user who dials up.

VCONs

Virtual consoles are consoles logged on across the network. When SVTA comes up, it creates VCON files in :PER for the virtual consoles (i.e., VCON1 VCON2 ..).. As with modems, EXEC must perform some special actions for VCONs.

(1) EXEC determines whether a console is a VCON or not by doing a ?GPORT on the control port of the console and checking the pid that is returned. Presently, if the pid returned is not the PMGR's pid it is assumed that SVTA is the owner. This will change when the addition of new terminal servers is supported.

(2) If OP is enabling a VCON, EXEC must connect to SVTA. This ensures that EXEC will receive a termination message if SVTA terminates.

(3) Users must have the 'virtual console' privilege to log on across the network.

(4) If SVTA dies, EXEC must perform the actions specified in the previous section on the VCON Logon Hash Table.

Console Memory

EXEC must allocate and free 3 types of memory for consoles:

- logon descriptor
- I/O buffer
- Profile buffer

(1) Logon Descriptor - Logon descriptor's are allocated and initialized when a console is enabled. This logon descriptor (LD) stays around until the console is disabled and deassigned.

LD allocation is handled in CONSOLE_DRIVER_TASK.PL1. Freeing up of the LD occurs in DISABLE_ACTION.PL1.

(2) I/O Buffer - During various portions of the logon sequence, EXEC must allocate i/o buffers for each console. EXEC will use constant strings when it can (e.g., "*** AOS/VS 3.0.0.0 / PRESS NEWLINE TO BEGIN LOGGING ON ***"). However, when reading/writing user specific strings, EXEC must allocate an i/o buffer for the console (e.g., reading username and password, writing out last previous logon, the termination message, etc)..

EXEC allocates i/o buffers in 2 sizes: small and large. When allocating an i/o buffer, EXEC places the address in the logon descriptor and sets a bit indicating what size

buffer it is. During logon, EXEC need only allocate a small buffer (for username, password, etc).. During logoff (and also if EXEC cannot ?PROC the user for some reason), EXEC requires a large i/o buffer because the message(s) are bigger.

(3) Profile Buffer - At various portions of the logon/logoff sequence, EXEC must read in the user's profile. Thus, EXEC allocates a profile buffer and places the address of the buffer in the logon descriptor. Then EXEC reads the user's profile into this buffer. EXEC will, of course, free up the profile buffer when it is no longer needed.

The profile is needed for username/password validation, previous logon time/date, ?PROC parameters and privileges, etc.

CHAPTER 8 - PMGR (Peripheral ManaGeR)
(AOS/VS Revision 4.00)

This chapter deals with the peripheral manager. the IOP, and IAC. It covers the internals of the PMGR software. the IAC software. and the IOP software.

Glossary of Terms and Key Data Bases

AGENT: Ring 3 of a user's process.

ASYNCHROUNOUS: An event that happens randomly in time. An event that is not expected to happen at a certain time, although it is expected. In the Data Communications world, this is a type of protocol.

BAT/WOMBAT: A furry marsupial mammal of Australia & Tasmania about the size of a Badger.

CHAINS: A singly linked-list data structure that does not use the MV/8000's queue instructions.

COMMUNICATIONS INFORMATION BLOCK (CIB): A data base used by both the HOST and IAC to communicate requests between each other. Does not require a lock, yet works in a queue like manner.

GLOBAL: The Pid 1 peripheral Manager process on both the IOP and IAC systems. The GLOBAL PMGR functions include unshared memory management. control command process and local request processing.

INTERRUPT (VECTOR) TABLE: An IOP Database used with the VCT instruction to transfer control to the appropriate interrupt handler when an interrupt is received by the IOP.

INTELLIGENT ASYNCHRONOUS CONTROLLER (IAC): An independent 16-bit ECLIPSE-like CPU with 16KW of local memory used as a front-end processor for asynchronous character I/O.

INPUT/OUTPUT PROCESSOR (IOP): An independent 16-bit ECLIPSE CPU with 32KW of local memory used as a front-end processor for asynchronous character I/O on MV/8000s and M600s.

LOCAL: The portion of the PMGR that lives in the AGENT.

PERIPHERAL INFORMATION BLOCK (PIB): The PIB contains all the information and status for a particular device supported by the PMGR. Each device has exactly one PIB associated with it.

PERIPHERAL: A peripheral is a device that is connected to the HOST CPU by various means. These include consoles, card readers, plotters, IAC's, and IOP's.

PERIPHERAL TABLE (PERTB) : It is a table of all the peripherals genned for the system and supported by the PMGR. It contains all the information that was acquired at sysgen time for this device including device code, initialization word, etc.

PID TABLE (PIDTB): A table that anchors a chain of PIBs together that are owned by the same PID. For example, after EXEC has enabled all the consoles but before anyone has logged on, the pib for each console that EXEC has enabled will be on this chain.

QUEUES: Those data bases upon which queue instructions are used.

REQUEST BLOCK (RB): A block of data which contains all info which the PMGR needs in order to satisfy the user's request. There is one RB per user request.

REQUEST TABLE (RTABLE): In an IOP system: when the host has requests for the IOP, this table tells the IOP which lines are awaiting activity.

In at IAC system: this is part of the CIB, again it tells which line a request(s) is waiting for service.

SHADOW PORT TABLE (SPRTB): A table that maps IPC ports into PIBs. Each entry contains the address of the PIB and a routine to start execution at for this type of port. There are three types of ports: Control, Read and Write.

STATE SAVE AREA (SSA): A block of data containing everything unique about the process at the time it did a PROC and BLOCK. Such things are: Characteristics, delimiter tables, edit program and open count.

SYNCHRONOUS: An event that is syncopated with another event. In the Data Communications world, this is a type of protocol used to communicate. According to Webster: arrangement of contemporaneous events.

USER: Any process but pid 1, rings 4 thru 7.

Introducing the PMGR

The PMGR supports all serial Asynchronous character I/O devices (terminals, card readers, etc.), versus block I/O devices (disks and tapes).

The PMGR on AOS/VS a seperate process (Pid 1) and is not part of the kernal as is traditionally done. This has the advantages of having a well defined interface, is more easily maintained since it is not "mixed" in with ther kernal, the kernal is (much) smaller, and it is easier to migrate the PMGR onto a separate processor (IOP/IAC). On the other hand, being seperate can cause confusion due to rev locks and causes the PMGR to be slower since it behaves the same as a user process.

The PMGR supports a wide variety of hardware, including:

Hosts:

MV/8000, MV/6000, MV/4000, MV/10000, MV/8000 II.

Peripheral devices:

IOP, IAC, ALMs, Plotters, Card readers, OPCON, Terminals and modems

The PMGR does NOT support:

Hosts: Any 16 bit Eclipse

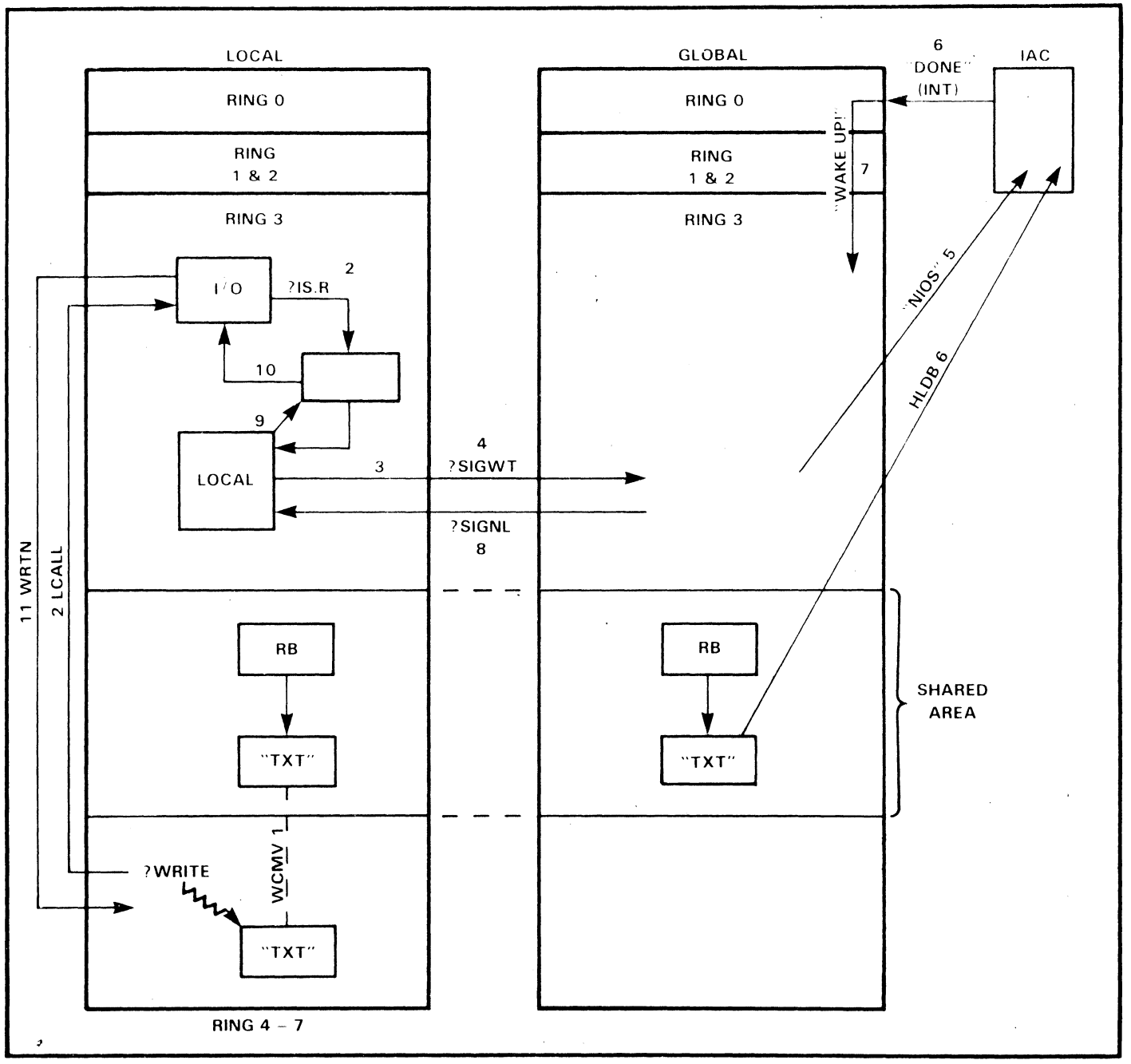
Peripheral devices:

ALM tied to the Host, ULM/ASLM, Line printers (LPA), Plotter on IAC systems.

The System calls processed by PMGR include ?READ, ?WRITE, ?OPEN, ?CLOSE, ?STOM, ?GCHR, ?SCHR, ?ASSIGN, ?DEASSIGN and more. Indirectly, the PMGR also processes ?PROC, ?TERM, ?RETURN, ?CHAIN via Kernal calls to the PMGR. Also, there a few "special" system calls used by the Agent/Local for shared memory management and initialization.

Quick Overview of ?WRITE

The following is a brief description of a ?WRITE system call as processed by the Agent and PMGR. The numbers correspond to the paragraphs below.



CS 03321

1) The user issues a ?WRITE system call, passing a ?WRITE packet. This is transformed into an LCALL to the Agent by URT_32.

2) The Agent dispatches the system call to the Agent I/O world which translate the ?WRITE packet into a ?IS.R packet suitable for the PMGR. The Agent then does a ?IS.R.

3) The ?IS.R system call processor detects that the IS.R is destined for the PMGR and deflects the call to the local PMGR (label PMGR.ISR).

4) The local PMGR: translates the IS.R packet into a RB, allocates a buffer large enough to do the ?WRITE, copies the user's buffer into the new allocated buffer (called a temp buffer) via a WCMV instruction, enqueues the RB/buffer onto the Global request queue, ?SIGNLs the global that it has work to do, and waits to be signalled of the completion.

5) The global PMGR: removes the RB from the request queue, finds the device that the ?WRITE is for, sets up the ?WRITE to the IAC, and issues an NIOS I/O instruction to the IAC.

6) The IAC reads a character at a time from the temp buffer into the ring buffer. After the buffer has been completely moved into the ring buffer, the IAC interrupts the HOST.

7) The interrupt world in the HOST detects which IAC/line caused the interrupt and wakes up the global PMGR.

8) The global PMGR then ?SIGNLs the local PMGR that the ?WRITE has completed, which wakes up the local PMGR.

9) The local PMGR the translates the RB back into the ?IS.R packet for the Agent and takes the good return to the ?IS.R code.

10) The ?IS.R takes the good return to the Agent I/O code.

11) The Agent I/O then translates the ?IS.R packet back into a ?WRITE packet and takes the good return from the LCALL into the user's ring.

And the ?WRITE is completed.

Proc'ing the PMGR

Global PMGR Initialization

The first process to run on AOS/VS is called CLIBT. It is bound into AOS/VS at system generation time and contains a table generated by VSGEN called PERTB (Peripheral Table).

After CLIBT has done an initial load, if necessary, it ?PROCs either PMGR or IPMGR. This is unconditionally pid 1. The system detects that pid 1 is being ?PROCed at IGHOST time and instead of loading the Agent into ring 3, loads either LPMGR or LIPMGR. This means the PMGR (Global) does not have an Agent but rather masquerades as one.

As soon as the PMGR gets control, it immediately sets itself up to do I/O instructions by turning LEF mode OFF and enabling I/O mode. Next the PMGR makes itself resident and turns superuser ON. Next, the PMGR ?WIREs from 0 up to INIT code. Next Page 0 is execute protected and the shared file in :PER is created (@PMGR.SF). The PMGR then allocates the first page of the Shared file.

The PMGR then ?MEMIs four pages of memory to recieve a copy of PERTB from CLIBT. This has a problem - do you know what it is? The PMGR then puts up an ?IREC to recieve its initial IPC.

The PMGR then scans PERTB once to determine how many devices have been genned so that the size of the shadow port table (SPRTB) can be determined. The SPRTB is then allocated from shared memory.

A second pass of PERTB is then made, this time initializing all the devices gen'd, ?DPMGR'ing (?IDEF) the devices and building all databases needed. This includes one PIB per line, one CIB and Line table per IAC. As each IAC is encountered is tested and then loaded with its resident code and databases. If this is an IOP system. then the IOP is loaded.

The PMGR sends an IPC back to CLIBT so it may continue on.

Then the runtime memory manager is initialized. Based on the number of devices gen'd, the size of the shared area is allocated. The shared area must be fixed in size since it is NOT possible to dynamically grow the shared areas simulataneously between the Global and all the locals. Why? Basically, the size of the shared area is is computed as 3 <= 3*devices <= 100. The shared area starts at page 400 (decimal). By limiting the shared area to less than 112. pages, this keeps the PMGR and Agent to a single level PTE. Double level PTEs are very undesireable since they require an additional page per process.

There are a variety of Init failures possible; usually due to an incorrectly gen'd system. If any of the system calls fail, the PMGR will panic, passing the error code returned by the system in AC1. It is possible that an IAC or IOP is bad, which can cause the PMGR to panic. On IAC systems, if the PMGR can detect the IAC failure soon enough (at test time), it will just report the failure and skip that IAC. For IOP systems, the PMGR just panics (easy way out). Another popular reason for failure is due to a rev lock between the system, the Agent and the PMGR. This can cause a variety of problems.

Local PMGR Initialization

When the Agent first gets control of a process just starting up, a routine is called in the local PMGR which is responsible for local initialization. Basically, all it does is fire off an IPC to the global. The global then gives that local Read/Write access to the shared file, allocates and deallocates 512 words of shared memory, and then fires an IPC back telling:

1) the internal revision of the PMGR
2) where the shared area starts
3) how big the shared area is and
4) the ID of the shared file.

Both messages must be IPCs, and must come from ring 3.

After the IPC is received, the local verifies that it is compatible with the global by checking the revision number and the starting address of the shared area. Next the local will establish (?SSHP) the shared area to the size as passed in the IPC. Then the local will open the shared file using the file ID passed to it, and ?SPAGE in the entire file with one system call. If any of these steps fail, the Agent will TRAP.

Internal Mechanisms

Memory Manager

There are three types of memory managed by the PMGR:

* Unshared Wired,
* Unshared Unwired,
* Shared Wired.

Each type of memory is managed the same way using a very modified buddy system (see Appendix A for a memo describing this). Briefly, the PMGR memory manager:

1) Does not recombine at RMEM (release) time but rather at GMEM time IFF that is the only way to get memory short of allocating more.

2) Does not shrink the memory pool. This is to limit the number of page faults. Once upon a time, page faults were on a per process basis, not a per task basis. It seems quite reasonable that we should shrink.

3) Does not require locking since Atomic Queue instructions are used to manage the buddy queues.

4) Is distributed between the global and ALL the local PMGRs. This means that memory requests for the locals are not single threaded thru the Global.

The PMGR's MM is a very important portion of the PMGR since every I/O request requires two (2) memory requests. It is very important that these requests be a quick as possible.

Each type of memory is used for different purposes:
- o Unshared Wired is for stacks, SSA, Console Chains.
- o Unshared Unwired is to spool ?SEND buffers, upto 2048 characters long. This allows users to send messages to consoles that have ^S up without taking an error.
- o Shared Wired is for ALL other databases.

There is one more memory manager used for the EDIT READ programsms, called CHAINS. There is one chain per unique memory type (ie Edit Programs) which contains a queue header, maximum memory for this

chain... As a request is made to allocate an EP, the manager first checks to see if this EP is already on the EP chain. If it is then the use count is bumped by one. Otherwise the size of the new EP is added to the current amount for all EPs. Then the EP is added to the EP chain and the use count set to one.

PID Table (PIDTB)

The PID table is used to translate pid numbers to owner's consoles. It is statically allocated at MASM time (unshared data). It is used when a user ?PROCs, ?TERMs or ?CHAINs to find the devices associated with that process. The pid table can be found in PGLOBAL.SR.

Peripherial Information Block (PIB)

The PIB is a per device database that tells current device state. The PIB is built at INIT time during the second pass of PERTB. The PIB contains such things as:

- o Status information (ie waiting on read)
- o Blocked queue pointers
- o Device specific drivers
- o Ring buffer pointers
- o Current and default characterisitics
- o Delimiter tables, current cursor position and edit read information
- o Open counts, current owner. shared console chains, and owner chains
- o Anchor for Read/Write/Control chains
- o Physical addresses for IAC/IOP processors

The PIB is the most heavily used of all PMGR databases and is the largest. It is defined in the parameter file PARP.SR. The PIB contains all information regarding a specific device that must be available between user requests. The PIB is also used to single thread user requests, one at a time, to the device via R/W/C chains. Only user requests at the head of the chain can be active.

Request Block (RB)

The request block (RB) is used to describe a user's request. There is one RB per user request, created at request time. Request blocks are defined in PARP.SR. A request block contains:

- o IPC port info
- o PID/UTID of requestor
- o UTID of Global PMGR task
- o Screen Edit/Edit Read information such as cursor position
- o User's buffer information such as address and size
- o Pointer to next request on chain
- o Global PMGR stack information

A request block is usually created by the local pmgr, with a temporary buffer associated with it. The RB is then passed onto the global, who assigns a stack to it and enqueues it to the appropiate PIB.

Shadow Port Table (SPRTB)

The Shadow Port Table (SPRTB) is used to translate IPC ports associated with a particular device to the corresponding PIE. The SPRTB is allocated between pass 1 and pass 2 of INIT time and is filled in during pass 2. Its definition can be found in PARP.SR. Each entry in the shadow port table contains two items:

- Address of the PIB,
- Address of a routine to pass control to.

There are three entries per device: Read, Write, and Control. Some devices, such as card readers, do not have a write entry since it is a read only device. By looking at the local port in the IPC information of the RB, the PMGR can easily lookup the PIB and quickly start the request.

Global - Local Queues

There are two queues maintained between the local and global PMGRs called:

- Global Request queue (requests to the Global),
- Global Completion queue (requests completed by the global).

These queues are statically allocated at MASM time and created with the shared file. Their definition can be found in PARP.SR.

The request queue is used by locals to enqueue work to the global without using IPCs. The local enqueues a RB onto the request queue using atomic queue instructions. If the RB just enqueued is the first one. then the local will ?SIGWT the global. If the RB is not the first one. then the local will just do a ?WTSIG knowing full well that the global will eventually see the RB just enqueued. This has the characteristic that as there is more work for the PMGR to do, the more efficient it gets.

The completion queue works in a very similar manner, although it is only used for unpended I/O requests. When the global has completed a request for a local and it was an unpended I/O request, the global puts the RB on the completion queue and ?SIGNLs the requestor. The requestor then searches the completion queue for a RB of interest (must match by PID, Ring and Key) and atomically dequeues it from the completion queue.

Task Scheduling in the PMGR

The PMGR process runs with a preemptive scheduler and six tasks. It has been found that this number of tasks can handle all of the PMGR duties. Task rescheduling time and idle time are also at a minimum due to the small number of tasks. How does the PMGR get everything done?

Only one of the six tasks is dedicated to any particular job. This is the timertask, which has task ID of 1 and priority of 0. The timertask is responsible for timing out all reads and writes which are issued with timeouts enabled. Therefore, to allow time-outs to work as accurately as possible, this task must be guaranteed to be scheduled before any other task. This is accomplished by giving it a higher priority than any of the other tasks.

The other tasks take turns running. They all have a priority of 1. They are able to do any of the other jobs the PMGR has to do. The PMGR does it's own task scheduling, since we are in essence our own Agent. Therefore. we can very easily assign a priority to our jobs, even though all the tasks run at the same priority. Job priorities are determined by the order in which the PMGR assigns jobs to tasks. The jobs the PMGR has to do and the priority they are allocated in are:

- o service the SUNBQ
- o wait for an event
- o wait for an IPC
- o wait for a signal

Service the SUNBQ

The name SUNBQ stands for Start of UNBlocked Queue. This is a queue of PIB's that are either blocked or done and waiting to be restarted. A PIB gets placed on the SUNBQ if the request block associated with it becomes blocked for some reason. The possible blocked bits are:

- o blocked on input of a terminator,
- o blocked on number of bytes,
- o blocked on output room,
- o blocked on connect,
- o read blocked on memory, or
- o write blocked on memory.

The blocked state is designated by which bit(s) are set in the status words of the PIB. The status bits in the PIB only refer to the head request on the read, write, and/or control queue.

For easier control of the SUNB queue there are two counters associated with it. These counters are NUNB and NEVEN and they reside in PMGR page 0. They must be in page 0 so the drivers in the system can access them. NEVEN is the number of requests that are blocked waiting for an event. NUNB is the number of requests that are unblocked and waiting to be restarted. The total of these two counters should be equal to the number of blocked and done bits set in all the PIB's on the entire queue. PIB's are allowed to have more than one blocked or done bit set at the same time. So, the total of these numbers does not necessarily have to be the number of elements on the queue. [An example of a PIB having more than one blocked or done bit set follows. Assume that a user's terminal is at a CLI prompt. This means that there's a read up and it is blocked on input of a terminator. The person sitting at this tube decides to type a ^S. Then someone decides to do a lot of ?SENDS to this user. The ?SENDS all get turned into writes to the target terminal. Since ^S is up on the target terminal, none of the writes will be able to write to the screen. As soon as the output ring buffer fills up the write will block on output room. Now there are two blocked bits set in the one entry on the SUNBQ.]

Why use a queue?

The main reason a queue is used to pend a request is to free up the task which is processing the request. If a queue was not used the task itself would have to pend. This is not a desirable condition since the request could be blocked indefinitely. Therefore, by queuing up the request the task is free to work on some other request. In this manner the PMGR can get by with only six tasks to do everything, since no task is unnecessarily idle.

Servicing of the SUNBQ

The method of servicing the SUNBQ involves the use of the counter NUNB. First, NUNB is checked to see if there are any requests on the queue that need to be resumed. If NUNB is zero the SUNBQ needs no service. If NUNB is non-zero the first PIB found on the queue which has any unblocked (done) bits set is dequeued. Then, the done bit which was found is zeroed. After which, the PIB is dequeued from the SUNBQ. If any other blocked or done bits are set, the PIB is put back on the tail of the queue. NUNB is then decremented to reflect the fact that there is one less request waiting to resume. The head request of either the read, write, or control queue, depending on which done bit was set, is started up.

Wait for an Event

To wait for an event means to wait for an interrupt to come in through the Kernal. A task that is being rescheduled determines if it should become the event waiter by first looking at NEVEN. If NEVEN is non-zero then a request is blocked waiting for some event to take place, such as a person to type a terminating character on their terminal. Before waiting for an event the task checks a word

in page 0 which has the address of the current task waiting on events. If there is no task address in this word, known as PBTCB.W in the Kernal or BTCB in the PMGR, this task becomes the event waiter by putting his/her address in this word to signify that there is someone waiting on events. It then forces a reschedule of the tasks so some other task can start running.

When an interrupt is received by the drivers in the Kernal, the following actions are performed. NEVEN is decremented to show one less event to wait for and NUNB is incremented to show that the request is ready to be unblocked. Then, the event waiter word is checked. If there is a task waiting the pend bit is cleared and word PBTCB.W is cleared so the PMGR will know that no task is waiting on interrupts. If there is no task waiting on events the Kernal does not clear any bits. This is no problem since NUNB has been incremented and the request will be processed on the next task reschedule of the PMGR. The reason for the event waiter task rather than just let NUNB get incremented and found later, is for speed in processing. If NEVEN is non-zero there should definitely be an event coming in soon. Therefore, if we know an interrupt is coming, we should wait for it and process it immediately, rather than wait for the next reschedule.

Wait for an IPC

The third job a task will try to do, when being scheduled, is wait for IPC messages being sent to the PMGR. There is a flag word which tells whether there is already a task doing this job. The flag is bit 0 of word TWOIR (Task Waiting On IRec) in PMGR page 0. If this bit is set there is already a task with a global ?IREC up. Otherwise, the task being rescheduled becomes what we call the IPC listener task. When an IPC is received and validated this task will process the new request. Bit 0 of TWOIR will be cleared and a task reschedule will take place, possibly resulting in another task becoming the IPC listener. If there is no IPC listener and an IPC is sent to the PMGR, the system will simply put it in an IPC spool file for the PMGR to receive later.

Wait for Signal

The final job a task will try to do is become the signal listener. The signal listener is responsible for any requests coming through the local PMGR rather than through IPC's issued to the PMGR. There is a flag word and bit used in the same manner as TWOIR is for the IPC listener. The word which signifies whether there is a task waiting on signals is TWOSIG (Task Waiting On SIGnal). Once a task becomes the signal waiter it checks the global request queue for any requests. Requests are enqueued to this queue by the Local PMGR and dequeued by the Global PMGR. If a request is found on the queue a ?WTSIG is not necessary. The request is simply dequeued and processed. If the global request queue is empty, a global ?WTSIG is issued. When the local gets the next request it will enqueue it onto the global request queue and if the queue was empty it will issue a ?TSIGNL to the global PMGR. The PMGR's ?WTSIG will

receive the signal and the request just placed on the queue will be processed. If there is no task waiting for signals when one comes in the signal is dropped, but the request is still enqueued to the global request queue. When a task finally is available for signals, it will see the requests on the request queue and begin processing them.

Go To Sleep

If there are no jobs for this task to do, or if it can't get memory for either the request block or stack, it suspends itself. We call this sleeping. In fact, the label of the PC placed in the suspended task's TCB is "SLEEPING".

Locks Used by the PMGR

There are three different types of locks used by the PMGR. The different locks determine the level of locking.

INTDS/INTEN

The most powerful type of locking is INTDS/INTEN. This lock does not allow any interrupts to be handled. This locking is actually done by instructions to modify the interrupt flag in the hardware.

Task scheduling disabled/enabled

Task scheduling is controlled through the UST (User Status Table). If bit ?BUPH is 0 task rescheduling is allowed. If the bit is 1 it is not allowed. With this lock in effect no other task will be allowed to run until scheduling is re-enabled. This lock is used when modifying queues, or status bits, etc. that other tasks could also want to modify. This is used in conjunction with INTDS/INTEN so that both interrupt world and PMGR's base level will not be able to modify data in critical regions.

PMGR Lock Words

The mildest form of locking is the use of lock words. This type of lock is used to prevent other devices from accessing key databases. On AOS/VS the only device which uses two-key locking is the IOP. This form of lock does not prevent the host or IOP from running unless it is trying to access a particular set of databases. The form of lock words the PMGR uses is known as a two-key lock. This type of lock consists of two words, or two bits, which make up the lock. These words are referred to as keys of the lock.

Locking on IOP Systems

When the Host is locking it checks to see if the IOP already has the lock. If the IOP's 'key' is zero the IOP does not have the lock, so the host sets it's key to one. thereby obtaining the lock. If the IOP's key is set. the host waits for the IOP to unlock before obtaining the lock. To insure that the IOP is functioning properly. the host PMGR times how long it waits for the lock. If the IOP has the lock longer than 1 second the PMGR panics, since the IOP seems to have died.

When the IOP tries to get the lock it first checks if the host's key is set. If it isn't the IOP sets it's key. Then it has to check that the host hasn't obtained the lock during the time it took to set it's key. If the host's key is still zero the IOP can be confident that it has obtained the lock. On the other hand, if the host's key is now set, the IOP was unsuccessful in getting the lock and must wait for the host to release the lock. The IOP merely does a busy wait until the lock is available. This is referred to as spinning on the lock. In this manner the host has priority over the IOP. This type of lock can be used with any device. In fact, the AOS/VS PMGR used to use it with IAC devices, but no longer does.

Locking on IAC systems

The IAC does not have to be explicitly locked out of databases that exist in the host. This is due to a number of reasons. One is that the IAC has it's own copies of key databases, such as the PIB and request block, and updates them to and from the host at well defined times. Another reason is that AOS/VS supports atomic read-modify-write instructions. This type of instruction means that a location can be read, modified, and updated with the modified result and is guaranteed not to be interrupted between any of these steps. For purposes of scheduling all words and bits are modified using these atomic instructions. The PMGR was not able to get rid of all the locks until revision 1.60. Before that revision the PMGR did not use the atomic instructions to update the communication interface between the Host and the IAC, so a two-key lock had to be used with this device, too.

Request Aborts

Now that we know how to schedule a request and insure the integrity of data, it is important to know how to abort a request, should the need arise. Request aborts have to be implemented when a request is being termed, such as ^C^A, or when an entire process is being termed. The tricky thing about request aborts is that the task which is processing the termination is not in control of the request which is being termed. Therefore. the task handling the termination does not easily know what state the request(s) is in that will need to be termed. It finds out what requests need to be termed by looking at the PID (Process ID) of the process terming. The PID is used as an index into the PIDTB. The PIDTB associates a PID with the PIB it owns. All PIBs with the same owner are linked together through the PIB itself. Once the head PIB is found, the entire chain is checked for any request outstanding for the PID terming. If only a request is terming, rather than the entire process, the PID is not used to determine which requests to abort. Instead, the global port number is tested and all requests on the PIB which have the same global port number are aborted. Also, rather than go through the whole chain of PIBs, only the one which issued the request termination is looked at.

What Does An Abort Do?

Once the task processing the termination request knows which PIB to deal with. it determines what state the head read, write, or control request is in. This is done by looking at the status bits in the PIB. These bits refer only to the head request of any of the queues, depending on which bits are set. since only the head request can be active.

How does it work?

When doing a request abort the first thing that is done is the abort bit is set in the request block at the head of the queue. Then the state of the request is checked. If there are any blocked or done bits set the request is assumed not active. The SUNBQ counters are updated appropriately and the request is removed from the PIB queue. The PIB is then removed from the SUNBQ if there are no more blocked or done bits set. If the request is blocked and was issued to an IAC device the IAC must also be notified of the abort. This is done by either issuing a 'stop read' or 'stop write' command to the IAC, depending on whether a read or write is being aborted. Commands issued to the IAC will be discussed in further detail in the next section dealing with IAC's.

If no blocked or done bits are set the request is assumed active. In this case the request is left running until it finds the abort bit set. When it finds the abort bit set it will term him/herself. By letting the request term itself it is not necessary for the task doing the abort to have to decide what state the other task is in and clean up after it. Since the task being termed knows what state he/she is in, it is quite simple for him/her to clean up before terming.

The abort bit is checked by every request in a few definite places. There are exactly four places in processing that a request checks to make sure it hasn't been aborted. One of these is every time a character is taken out of the input ring buffer (i.e. every time a character is received from the terminal). The abort bit is also checked at the start of every write request. The third place the bit is checked is before blocking a request block for any reason. The fourth place a request checks whether it should term itself is immediately before resuming a process after unblocking. The check is made here just in case an abort took place after the done bit was cleared but before the request actually started processing. When the request sees the abort bit it clears up all locks it might have, frees up any memory it can, and then sends the task to find something else to do.

Now we've covered how head requests are aborted, but what about requests that aren't at the head of a queue? All request blocks that are outstanding for a PID that is terming, in the case of a process termination, must also be torn down. Once the head request is taken care of the rest are much easier, since they are not active. Any non-active requests are simply dequeued from the PIB queue and any memory allocated to them is freed up.

IACs

What is an IAC?

According to the IAC Programmer's Reference Manual "an IAC (intelligent asynchronous controller) is a single, 15-inch square board that allows you to connect either 8 or 16 terminals and other asychronous devices to your computer. The IAC contains a microprogrammed processor with local memory, a host interface, and a communications interface." An IAC resembles an Eclipse in that it is bit sliced. It doesn't have a front console, but it does have two LEDs. These LEDs are used by diagnostics to signify what state the IAC reached before failing. When the system is not running the LED on the left is in a steady on state. When running normally. the software causes the LEDs to blink in decreasing numerical succession (3 -> 0). When the IAC panics the software lights both LEDs and does a 'JMP .'. This instruction is necessary since the IAC can not be halted. When the host wants to stop the IAC from running it must issue an I/O reset to the device.

How The Pmgr Uses The IAC

The PMGR uses the IAC to do a lot of the manual work of read and write requests so the host CPU is freed up to do other jobs. Most of the code which processes reads and writes is loaded into the IAC. Then, all the host has to do is set up the request, issue it to the IAC, and wait for the IAC to signal that it has completed. All of this will be covered in greater detail later.

Databases Used By The IAC

The IAC uses a number of databases, some of which exist solely in the IAC, some which exist solely in the Host, and others that exist in both in slightly different forms.

Those databases which exist in both the IAC and the Host are:

- o the PIB (Peripheral Information Block),
- o the request block,
- o line table, and
- o stacks.

The PIB and Request Block in the IAC are used for the same type of information as the databases which exist in the Host. The reason the IAC has it's own set of these is so data chnnel activity will

be at a minimum. With it's own PIB and Request Block the IAC can do much of the read and write processing without needing the Host. There are a few differences between the PIB and Request Block in the IAC and those in the Host. One is that the IAC's copy is much smaller than the Host's. It contains a subset of the Host's with a few added words that only the IAC needs to know. Another difference is that the order of contents is different between the two.

All other databases exist in only one of the locations. Some databases that are only in the Host are referenced by the IAC, but the Host never directly references any of the databases that exist solely in the IAC. Those databases that are shared by the Host and IAC, but which are located in the Host, are:

- o the CIB,
- o the temporary buffer,
- o delimiter tables.

The IAC must be able to read and possibly write to these databases. The Host also has a line table and stacks, but these are never referenced by the IAC. Those databases that are in the IAC only are:

- o the line table,
- o ring buffers, and
- o stacks.

Internal Mapping Unit (IMU)

One of the most important features of the IAC is how it moves data back and forth between the host. To do this it uses a non-standard map in conjunction with some unique instructions. The map consists of what is called the internal mapping unit or IMU. This map uses the data channel map slots to actually access the host. Before referencing the data channel, however, other calculations are done. The actual translation involves a number of steps. First, bits 1 through 5 of the logical address given in the instruction are used to determine which internal map slot to use. Each IAC has 32 internal map slots do with what he/she wants. Bits 6 - 15 of the logical address go unchanged to become the lower 10 bits of the data channel address. In the meantime, the upper 5 bits of the logical address have been converted to the upper nine bits of the data channel address. These nine bits select the data channel map slot to use by the internal map slot. The data channel map translates the 9 bits into 14 bits which specify the uppper bits of the host physical address. The low order 10 bits of the data channel address become the low order 10 bits of the physical page. Finally, the data on the necessary page is accessible.
Of the 32 internal map slots available to the IAC, AOS/VS only uses seven slots. This means that only seven Host data channel slots are needed for each IAC in the system, which leaves some extra slots available for other users to use. The IAC uses one of the IMU slots for mapping PMGR page 0 and one for mapping the PMGR database table page. The database table page is used when initial-

izing the IAC so that it can load the data from the Host. Three of the slots are used for mapping all of the Host's PIBs associated with the 8 or 16 lines for the given IAC. This infers that all PIBs for each IAC are allocated contiguously in Host memory. It also assumes that the PIBs will not take more than 3 pages of memory, which potentially could be a problem. Currently, however, we can fit ten PIBs on a page and need only sixteen PIBs at most for an IAC so we're well below our limit. The other two slots of the IMU the IAC uses are dynamic slots. The IAC is able to change the page these slots map to using the HMSTA (host map store accumulator) instruction. This instruction specifies the data to be loaded into a given Host data channel map slot. In this way one DCH slot can be used to map various pages. These two dynamic slots are used to map all non-static databases. One slot is used to map the request block, temporary buffer, or user's user array as needed by the request. The other slot is used to map the delimiter table, insertion buffer, or edit table as needed.
The instructions which use the IMU are what are called the 'host instructions' on the IAC. These include instructions such as HSTA (host store accumulator), HLDB (host load byte), HINS (host increment and skip if zero), HLDA (host load accumulator), HSZBO (host skip if zero bit and set to one), etc. All of these instructions take the logical address given in the instruction and perform the mapping translation on it. If the logical address is bad, undetermined results will occur.

Initing the IAC

The IAC memory is statically allocated at initialization time. The Host moves all the necessary information to the IAC when it is initing it. The first thing the Host does when it finds an IAC device genned is check that the IAC device is alive and well. It does this by issuing a "Get Info" request to the IAC and waiting for it to respond. If the IAC does not respond within a certain amount of time, it is pronounced dead. (This time varies from one CPU to another.) A message is sent to the operator's console stating that the IAC with device code N does not respond and will be bypassed. In this way bad IAC's will not prevent the system from coming up. Once it has been determined that the IAC is in good working order, the Host begins loading the IAC resident code into the IAC. It does this by doing a read block of IACRS.PR and writing the file to the IAC in 400 word blocks. Once all the code is loaded, the Host begins writing out the databases. The order of allocation in IAC memory of the databases is as follows:

- IAC PIB
- Input Ring Buffer
- Output Ring Buffer
- IAC Read Request Block
- IAC Write Request Block

This is repeated for every line on the IAC. When all the lines have been initialized the Host writes out the IAC line table. This table has been filled in as the Host set up the PIBs in both the

IAC and Host memory. Then the Host starts the IAC running at location 1. These steps are repeated for each IAC in the system.

Starting a Request

Now that the IAC is all set up and running it would be nice to let it do some work. The interface between the Host and IAC is mainly the CIB (Communication Interface Block). The definition of this database can be found in PARP.SR. It basically looks like:

- o Device code of IAC.
- o Number of lines on this IAC.
- o IAC panic register.
- o Link to next CIB on chain.
- o Host -> IAC command word. This word contains a command number for those commands that should be done immediately by the IAC, even before other interrupts. It is used in conjunction with the Pulse interrupt. The uses of this word will be explained later.
- o Completion Register (IAC -> Host). This register is used for completions due to asynchronous completions. Specifically these are ^C^x sequences and modem disconnects.
- o Completion Line # (IAC -> Host). This is the line number which issued the asychronous completion which is found in the previous word.

Host Line Table Address.

# of Requests outstanding to IAC. This value is incremented by the Host every time it issues a request to the IAC. It is decremented by the IAC when a request is processed.

# of Completions outstanding to Host. This value is used exactly opposite of the previous word. The IAC increments it everytime a request is completed and the Host decrements it every time it sees the completion from the IAC.

Request table - 2 words per line. The first word of the entry for each line contains the request bits that correspond to the request count above. The total of all the bits in the first word of all the entries should be equal to one more than the number of requests outstanding, since the count starts at -1. The second word for each entry contains the bits that correspond to the completion count above.

The Host uses interrupts to inform the IAC that there is a request for it to process. When the Host finds something for the IAC to do it sets the appropriate bit in the request table at the end of the

CIB. Then, it increments the request count in the CIB. If the count becomes zero the Host interrupts the IAC, since it is not currently processing any requests from the table. If the count is non-zero the IAC is already processing requests in the table and will find the new request sooner or later. Because the count and bit in the CIB are updated using atomic instructions, no locking between the Host and IAC is necessary.

Processing by the IAC

Once the IAC receives an interrupt it reads the request count from the CIB to see how much has to be done and it looks at the request table to determine what request to process. It will do all the requests for one line, then move on to the next line, etc. until the request count goes to -1. As each request is found the request count is decremented and the bit in the request table is cleared. The possible requests that can be issued to the IAC and briefly what function they perform are:

- o Start input - initialize device for input. This truncates the input buffer.
- o Start output - initialize device for output
- o Clear Wombat device request - clears some of the status bits in the IAC's PIB.
- o Process a modem on request - this brings up the hardware signals DTR and RTS so the modem can begin to communicate.
- o Process modem off request - allows all present output to complete, then shuts off the modem if the line is on a modem.
- o Start Wombat read request - enqueue read request onto eligible queue and set the ready to run bit so scheduler will run it.
- o Stop Wombat read request - preempt the currently running read, if there is one. Else, just tell Host that the control request is done.
- o Start Wombat write request - enqueue write request onto eligible queue and set the ready to run bit so scheduler will run it.
- o Stop Wombat write request - preempt the currently running write, if there is one. Else, just tell Host that the control request is done.
- o Set Wombat characteristics - set the characteristic words in the Wombat's PIB to the newest current characteristics. This needs to be done everytime the characteristics change on the Host side.
- o Set timeout constants - set the timeout constants in the Wombat PIB to be the newest timeout value set by the user.

If any bit is set in the request table that is undefined, the IAC will panic. The order the requests appear above is also the order of priority. The left most bit of the word has the highest priority, decreasing as you move to the right. Also, the lines themselves have a priority structure, namely from the lowest number line on an IAC to the highest line. These priorities are assigned by the software, due to the method used to find a request to process.

Currently, all of these requests are handled at interrupt level, i.e. with interrupts off. In Revision 4 this will be changed so that a base level task will look through the request table once the initial interrupt from the Host has been received. This will cause reads and writes to get processed more quickly, since less work will be done at interrupt level.

When a read or write request actually starts processing in the IAC, the first thing it does is set up the Wombat's PIB and Request Block from the Host's. It does this using Host instructions, the IMU, and the data channel. Once these two databases are set up properly, the IAC only needs to access the Host when reading from or writing to those databases that exist solely in the Host. These are the temporary buffer, the delimiter tables, and the editread databases. The IAC code contains all the necessary read and write code, including screen edit and editread functionality.

Scheduling

The way the IAC schedules requests is in a round robin type fashion. As reads and writes come into the IAC, they are enqueued onto the eligible queue. When the scheduler is invoked, the queue is searched from the request following the current request all the way through the queue. The queue is a circular queue so the search will continue indefinitely until it finds a request with the ready-to-run bit set. The first two tasks put on the queue are the timer task and the checksum task. The checksum task is always ready to run, so the scheduler will never loop forever without finding something to run.
The IAC does not have any type of pre-emptive scheduler. A request runs until it has to block waiting for an event, such as if there are no more characters to read from the input ring buffer. Only when the request blocks itself or when it completes is another request allowed to run. This type of scheduler can cause problems. If a request takes a lot of processing time and never has to block, it will prevent other requests from running. There are also a couple of advantages to this scheduler. One is that it is very simple and straight forward to implement. Another, which is related to this simplicity, is the fact that it takes very little code space. This is especially important in the IAC, where memory is scarce.

Request Completion

When a request completes the first thing the IAC does is update the Host databases. Throughout the processing of the request most of the data was only altered in the IAC's databases. Now, the Host's databases must be updated so the user will receive the correct information. All the data is moved over across the data channel map, just opposite of when the IAC started the request. Once the

data is moved, the IAC sets the completion bit in the CIB so the Host knows what type of request was just completed. The IAC then increments the number of completions in the CIB. The number of completions start at -1, just like the number of requests do. If incrementing the number of completions causes it to become 0, the IAC will issue an interrupt to the Host. Otherwise, the Host is already handling completions from the IAC and will see the new bit sooner or later.

On the Host side the driver in the Kernal will receive the interrupt. The request table on the end of the CIB is checked to determine what has completed. Processing then continues normally with notification to the user of how his/her request turned out. The possible completions the IAC can signify in the request table are; read done, write done, read control done, and write control done.

Modem Disconnects and ^C^x Sequences

Let's suppose the request did not complete normally. Let's say the user typed a ^C^A in the middle of output to the screen. The IAC wants the Host to know that this is not a normal completion. The way it does this is through a word in the CIB called the completion register. The IAC sets the appropriate bit, either that a control sequence or a modem disconnect occurred, and then interrupts the Host. The IAC driver in the system looks at the completion register and dispatches to the correct routine which will handle the completion.

Panics

When the IAC's software finds an inconsistency it can't deal with, it will panic the system. To do this it puts the panic subcode into the CIB and interrupts the Host. When an interrupt comes into the IAC driver in the system, the first thing that is checked is the panic word in the CIB. If it is non-zero, the Host assumes the IAC has requested a system panic. If the panic word is zero, the driver will then check the completion register and then the completion count. The panic value written on the screen is a 17002. The subcode is found in the first word output on the screen and is usually greater than or equal to 2000 (there are still a couple of panic subcodes that were inherited from the IOP that are less than 2000, but they are slowly disappearing). [As a side note, the IOP also panics with a 17002, but none of it's subcodes are in the 2000 range.] The definition of the 17002 panic subcodes can be found in the AOS/VS panics listing under 12300 panics.

Powerfail

The latest addition to the IAC repertoire of commands has been powerfail support. The unique thing about a powerfail request is that it is on a per IAC basis, rather than line by line. Before this command all requests to the IAC were issued and dealt with

line by line. Also. a powerfail command needs to be handled before any other commands are processed, so nothing gets lost. To implement this new feature a new word called the command word, was added to the CIB. Upon notification that a powerfail is in progress, the Host sets the command word in the CIB and interrupts the IAC. For the IAC to realize that the interrupt is not the normal interrupt that is used when processing requests, the Host uses the PULSE interrupt. The IAC gives this interrupt priority so powerfail requests will be processed immediately.

When the IAC realizes that a powerfail has occurred it does very little processing. It simply clears the command word in the CIB, clears the pulse interrupt and disables RTC (real time clock) and busy interrupts so they will not interrupt the IAC. The IAC then waits for another pulse interrupt from the Host signifying a power restart command. All other interrupts are ignored, so no processing in the IAC is lost. When the power restart command is received the first thing the IAC does is simulate a modem disconnect on all modem controlled lines. Then, all receivers and transmitters are turned on, and business continues as though nothing happened.

MCP1 (COMBAT BOARD or ALPHA)

The MCP1 is a board that has eight asyncronous lines (similar to the IAC), two synchronous lines, and a data channel line printer on it (thus combat). At init time the pmgr sizes the board to see whether it is an iac or an MCP1. If the board is an MCP1 then the pmgr loads ALPHARS down into the board.

MODULES FOR THE ALPHA

In general, the modules that were changed for the ALPHA have an alpha extension to the module name.

WINTS/WPRDR --

WPRDR.ALPHA is a general module that handles output service, interrupt level output, disconnects, and modem service from the host. WINTS.ALPHA is harware specific code for dealing with alpha interrupts.

SETCHAR

This routine sets up the uart to reflect the user-specified characteristics (baud rate, etc).

MAPIT

This routine maps a host physical page to an alpha logical page.

WINTIT/PCI

WINIT is a routine that sizes the alpha and initializes it. PCI deals with modems by turning transmit on or off and turning the modem on or off.

IOP

IOP Hardware

The IOP is a slightly modified S-130 ECLIPSE CPU that plugs into the MV/8000 chassis. It has an I/O only chassis into which Asynchronous Modem Interfaces (AMI/8) or Asynchronous Terminal Interfaces (ATI/16)'s, Card Readers (CRA's), Plotters (PLA's) can be plugged in. The IOP has a fixed 64KB address space in local semiconductor memory of which any logical page can be mapped to HOST memory using the data channel. The IOP device code is fixed at 65. The HOST appears to the IOP as device code 4. Communication between the IOP and the HOST CPU is accomplished by manipulating DONE and BUSY flags that can be seen by the other CPU. The IOP's BUSY flag is set whenever the IOP is running and cleared when the IOP halts. The IOP's DONE flag (as seen from the HOST) is set/cleared in the IOP by issuing NIOS/NIOC to device code 4.

Front Panel Console

Instead of an S-130 CPU with all the gaudy lights and switches, the IOP has a programmed I/O interface that allows the software to control the IOP like a computer operator controls the front console of a standard ECLIPSE processor. The interface consists of:

* Console Switch Register - replaces data switches on standard console.
* Console Function Register - replaces function switches.
* Console Address Register - last address referenced by IOP.
* PC Save Register - holds the PC when the IOP halts.
* Console Register - replaces the data lights on standard console.

Modified ECLIPSE MAP

The ECLIPSE MAP is defined by a set of status and control words and a 32 word block of data describing each of 32 pages in the ECLIPSE logical address space. The IOP map is very similar to the standard ECLIPSE MAP. The main difference is that The IOP can select any or all of its 32 logical pages to be either LOCAL (using IOP local

memory) or HOST (mapped through the HOST DATA CHANNEL A,B,C,D). In addition, the IOP can invisibly load a HOST DATA CHANNEL slot without HOST intervention. It does this by issuing an I/O instruction to the IOP map with a special bit (DML) to allow DATA CHANNEL MAP loading. This means that the MV HOST CPU need not be interrupted while a DATA CHANNEL slot is being modified. The first 28 pages are always mapped to local IOP memory. The remaining 4 pages are used to access PMGR PAGE ZERO (including the HOST-to-IOP REQUEST TABLE), PIB and Ring buffers. The latter two pages are mapped dynamically to Physical pages in HOST GLOBAL PMGR memory of the current device being acted upon.

Real Time Clock

The IOP has a Real Time Clock that runs at 60 Hertz. This means it generates an interrupt 60 times per second. This clock is used to time modem operations. Every 30 seconds, the RTC interrupt service routine scans the IOP line table to timeout modem events. One function of the RTC is to time out the loss of carrier detect (CD). If CD has been down for more than 5 seconds, on an active modem line, then the modem is disconnected. Another function of the RTC is to insure that a modem connection is successfully established by waiting an initial period of 40 seconds for CD to be raised by the modem.

Data Bases In the IOP

There are a number of IOP-only data bases that are used to process PMGR I/O. These are the DEVICE TABLE, COMPLETION TABLE, INTERRUPT VECTOR TABLE, the LINE TABLE and the IOP STACK. The following describes these in greater detail.

Device table

The IOP Device table is a block of data containing the Physical addresses of all the HOST PIBs that are of interest to the IOP. It does not contain CONO's PIB because CONO is handled entirely in the HOST. The table has three sections; Card readers, Plotters and ATI lines. There is a maximum of two card readers and two plotters that the IOP supports. There are up to 128 ATI lines supported in the IOP. Since only one IOP is allowed on an MV/8000 system, the total PMGR device count is 133.

Completion Table

The Completion table is a structure to contain the interrupting events (^C^x sequences and Modem Disconnects) that the HOST PMGR is not explicitly waiting for. These events are handled in the AOS/VS KERNAL module IOPDR.

This table is only used for ATI lines because there is no such thing as a ^C^A on a card reader or Plotter.

Interrupt Vector Table

The Interrupt Vector Table is a table of Interrupt Service Routine addresses ordered by device code. This allows a hardware interrupt to execute a unique set of instructions defined for each type of device. The IOP instruction VCT is used for this purpose. This instruction is identical to the standard ECLIPSE instruction. The simplest form of the VCT instruction is used in the IOP to minimize the time it takes each interrupt-level path to run to completion. This is why mode A of the VCT instruction is used in the IOP. In this mode, the VCT instruction determines the DEVICE CODE of the device needing service and immediately dispatches through the table to the corresponding device driver.

There are 5 different interrupt service routines in the IOP: (numbers in parentheses are fixed device codes)

1) Dummy Powerfail routine (0)

2) HOST REQUEST and RTC (4)

3) Plotter output

4) Card Reader

5) ATI's and AMI's (34)

There is a dummy powerfail routine in the IOP because the powerfail interrupt only occurs in the HOST. In case there is a problem in the hardware when an INTA instruction is issued and no device responds with its device code, what comes back is device code 0. This will cause a panic 17002 subcode 303.

All entries in the table that correspond to undefined device codes contain a pointer to a NOP interrupt service routine that counts the number of undefined interrupts, clears the interrupt and returns to base-level.

Line Table

The IOP line table is a structure (two words per line) containing modem status bits and a timer word for controlling modem lines. The length of the Line Table is 256 words for a maximum of 128 console lines. The Line Table is scanned by the RTC for non-zero timer words. If any are found, they are decremented. When any timer word decrements to zero, then the modem on that particular line is disconnected.

A stack

The IOP contains a single stack used both for Interrupt-level and base-level processing. There are two base-level functions in the IOP. The first function is to dequeue any completions stored in the COMPLETION table and interrupt the HOST for each one. The

second function is to insure that the static areas of IOP memory (constants and code areas) do not get corrupted.

Data bases in the Host

The data bases in the HOST are used by the IOP, KERNAL and GLOBAL PMGR. In ring 0 this includes the IOP DCT and HOST line table accessed by IOPDR when IOP completions are processed. In ring 3 there is the REQUEST TABLE, PIB, Ring buffer, and various GLOBAL PMGR PAGE ZERO variables.

Device Control Table and HOST DEVICE Table

The Device Control Table (DCT) is used by the KERNAL interrupt service routine IOPDR to process interrupts from the IOP. This table contains the physical address of the HOST line table in PMGR space. The IOP generates interrupts to the HOST when an asynchronous completion is dequeued from the IOP completion table and sent to the HOST. The DEVICE TABLE contains physical addresses of PIBs that correspond to the entries in the IOP completion table. Physical addresses are needed because the LINE TABLE and PIB are not part of the logical address space of the KERNAL. This allows the IOP driver to find the PIB associated with a particular completion.

IOP Request Table

The IOP request table is the forerunner of the CIB for IAC's. It is defined in PARIOP.SR and made up of double word entries each being for a specific device the IOP supports. The first byte contains a flag bit for each of the following:

1) Start device - do something to cause an interrupt on the device

2) Clear device - undo it, this will also clear any other specific request flag bits in this byte

3) Modem on - Raise the modem control signals Data Terminal Ready (DTR) and Request To Send (RTS)

4) Modem off - Prepare to shutoff the modem device (lower DTR and RTS after output completes)

5) General request - flag to decide whether to enque this cell into RTBLE.

The second byte is used to store the control character following the ^C. The second word of each entry is used to link all the request cells in the request table. RHEAD points to the first cell and each link contains a cell number to the next cell. RTAIL contains the cell number of the last cell enqueued. The last cell always contains -1 in the link word. If RHEAD = -1 then there are no requests outstanding to the IOP.

Peripheral Information Block

The PIB, as was mentioned in previous sections, is a data base consisting of all necessary information for a specific device. The main items in the PIB that are specific to IOP operation are:

a) Device code and line number - to enque a request in the HOST-to-IOP request table (RTBLE) and to issue I/O instructions to the device.

b) PIB status - very briefly, there are 6 major states:

* INACTIVE (no read or write request in progress)

* INPUT BLOCKED (nothing in the input ring buffer)

* INPUT DONE (resuming a read request for input)

* OUTPUT BLOCKED (waiting for output ring buffer to empty)

* OUTPUT DONE (resuming a write request for more output)

* RUNNING (processing the input and/or output request at the head of either the READ or WRITE request chain)

c) PIB characteristics - to define special situations, for example:

* /PM to do output in page mode

* /EB0 & /EB1 define how characters echo

* /ESC to cause ESC char to act like ^C^A

* /MOD to do extra processing for modem devices

d) Ring buffer information - to allow passing data to/from interrupt-level (this will be explained in more detail below)

e) Timer constant and counters - to allow read/write timeouts

Ring buffers

Ring buffers are holding places between interrupt-level and base-level. A base-level task processing a write request will fill an output ring buffer and block (waiting for buffer to empty). An output interrupt will trigger an interrupt service routine to take a character out of the buffer and send it to the device with a DOA (or DOB or DOC) I/O instruction. An input interrupt will cause an interrupt-level routine to read the character and put it in the input ring buffer. When the blocked base-level task is re-

activated to resume a read request, it will take characters out of the input ring buffer and transfer them to a Text BUFfer (TBUF) for later disposition by the LOCAL PMGR.

There are 8 words for each ring buffer. The first 2 is a double word containing the byte pointer to the beginning. This is also used to decide if the buffer has been allocated for this device. For example, a PIB for a card reader does not have an output ring buffer, so the byte pointer is zero. Or a device that is not enabled will not have a ring buffer allocated to it. The next three words define the physical page and beginning and ending offsets to the buffer. This is used by the IOP and KERNAL device drivers to determine the location and extent of the buffer. The sixth word is the Insert Byte Offset. This is used to point to the next location to store a character. The seventh word is the Remove Byte Offset used to point to the next byte to remove from the ring buffer. And lastly, the eight word defines the size of the buffer. This is used at OPEN or ASSIGN time to decide how much memory to allocate for the ring buffer.

Page Zero Variables

Among the various ZREL locations used by the GLOBAL PMGR, the following are especially important to good communication between the GLOBAL PMGR and the IOP:

* NUNB - Number of unblocked events waiting for a task to resume processing
* NEVEN - Number of events waiting for external action (keystroke to come in or output to go out)
* BTCB - Physical address of a blocked Interrupt-Event-Waiter (IEW) TCB
* UBTCB - Physical address of IEW TCB currently being unblocked
* IOPKEY - IOP's key to the IOP <-> HOST lock
* HOSTKEY - HOST's key to same
* IOPCPL - Asynchronous completion word from IOP (^C^X or modem disconnect)
* IOPLIN - Line number of the device for above
* IOPPAN - Non-zero when IOP wants to panic the system
* RHEAD - Cell number of the first enqueued request to the IOP request table (RTBLE)
* RTAIL - Last request enqueued to IOP RTBLE
* RTBLE - Address of HOST to IOP request table

IOP initialization

When the GLOBAL PMGR initializes, part of its job is to load up the IOP from file ":IOPRS.PR". Before doing this, the PMGR will determine if the IOP is in good working order. First the PMGR stores zero into each of the 32k words to initialize the IOP semiconductor RAM. Then the IOP resident code is loaded by opening the file containing the IOP-resident code and reading a block at time into a buffer. Then it loads each word through the IOP front console interface. While each word is being loaded, the PMGR reads back the contents of that location to insure the IOP RAM is functioning properly. If an error is detected in this conversation with the IOP, the PMGR will panic with various subcodes indicating what operation failed.

When all the IOP is loaded, it is started at location 1. This location contains "JMP @.START" and the IOP proceeds to do its own initialization:

1) Jump into the IOP debugger (if enabled)

2) Change location 1 to contain the interrupt service routine

3) issue IORST to initialize all controllers attached to the IOP and turn off interrupts

4) Load the IOP MAP that defines 28 pages of local memory and 4 dynamically mapped pages to the HOST

5) Load the MAP/MEMORY control word to:

   * Disable Address Save mode

   * Enable parity checking

   * Enable DATA CHANNEL mode and USER mode

   * Use DATA CHANNEL MAP D

6) Initialize interrupt vector table with plotter and card reader interrupt service addresses. The HOST and ATI addresses are assembled in because they have fixed device codes.

7) Clear all ATI lines by explicitly turning off the three sections of each line: receiver. transmitter and modem

8) Initialize only the ATI lines that have been VSGEN'ed. This is to turn on the receiver and specify LINE CHARACTERISTICS word (character length. even/odd parity and baud rate).

9) Enable interrupts (INTEN)

10) Calculate the checksum

11) Go to the IDLE loop

Once in the idle loop, the IOP waits for Interrupts from the HOST, RTC, and I/O devices.

Processing an I/O Request

When the PMGR receives an I/O request for a device, it either puts it at the end of the chain of requests (if there is already an active request there) or initiates the request in one of two ways depending on whether it is doing input or output. What follows are two discussions, one for input and the other for output.

Input

In the HOST

The user task points to a buffer to contain the input data and issues a ?READ system call. The Agent determines that the call is to a PMGR device and prepares an IPC block. The call is deflected to the LOCAL PMGR who allocates a read request block and a TBUF in the shared file PMGR.SF. Then the request is enqueued to the GLOBAL request queue and the LOCAL tells the GLOBAL PMGR there is something to do (if necessary by ?SIGWT, else just ?WTSIG). Now the AGENT is pended waiting for the input data to be delivered by the GLOBAL PMGR.

The GLOBAL PMGR will now process the request enqueued by the LOCAL. It does this with the TWOSIG task (Task Waiting On SIGnal) which dequeues it from the GLOBAL request queue and enqueues it to the intended PIB's read request chain at the end of any read requests already issued to the same keyboard. When it becomes the head of the chain (it could happen immediately), the GLOBAL PMGR TWOSIG task will allocate a stack and start processing the read request.

Note: What has just occurred was a switch of the role of the TWOSIG task into a processing task. The TWOSIG created another task and appointed itself a processing task (POOF!).

The processing task will now attempt to get a character received from the device. If it succeeds in removing a character from the input ring buffer, it will continue to execute in what is called the "main input loop". This loop is the main processing code that takes a character from the input ring buffer, checks whether a delimiter is typed, echoes the character, places it into TBUF, and goes back to the top of the loop to get another character. The current task will stay in this loop until all the characters have been extracted from the input ring buffer and processed into TBUF.

If it requires another character (the read request has not been satisfied) and the input ring buffer is empty, then the current task will block the PIB for this request and go find something else to do.

In the IOP

Meanwhile, the IOP is busy fielding the interrupts from all the active devices who require output data and are supplying input data. The input request mentioned above will be blocked until a character is available for it. This character will be processed by IOP interrupt-level in the following way:

1) An input character (typed on a keyboard) causes an interrupt in the ATI hardware.

2) The interrupt service routine for that device is invoked which reads the character and places it into the input ring buffer.

3) The PIB status is checked for the presence of a blocked condition (waiting for input).

4) If the PIB is not blocked on input, the interrupt service routine is completed and the interrupt is dismissed (Go to IDLE loop).

5) If the PIB is blocked on input, the IOP will unblock the PIB and wake up the blocked Interrupt-Event-Waiter (IEW) task in the GLOBAL PMGR (if necessary).

Back to the HOST

The GLOBAL PMGR IEW task will scan the SUNBQ of blocked PIBS looking for a PIB request to restart. When the scan is successful, the following will occur in the HOST PMGR (GLOBAL and LOCAL):

1) The IEW task will cease to be an IEW task and become a processing task by resuming at the place where the previous task had blocked on input. Now this task will resume the input request by taking the input character from the ring buffer and placing it in the TBUF (combining it with previous characters if doing a multi-byte request).

2) If the request has not been satisfied, it will go get another character from the ring buffer. If there are no more characters in the input ring buffer, then it will block this request again (waiting for input).

3) If the request has now been satisfied, the read is unpended by doing ?TSIGNL to the LOCAL PMGR task waiting for the input data.

4) The LOCAL PMGR will then move the data to the user and return to the AGENT I/O code.

5) The AGENT will then return control to the user task that made the ?READ system call.

Output

In the HOST

The user's task makes a ?WRITE call which gets the AGENT to call the LOCAL PMGR in ring 3 of the user's process. This causes the LOCAL PMGR to allocate a TBUF in the shared area and enqueue the request to the GLOBAL PMGR's GLOBAL REQUEST QUEUE. The GLOBAL PMGR (using its TWOSIG) will enqueue the request to the proper PIB. If there are no requests at the head of this chain, the TWOSIG will transform itself into a processing task and proceed with the write request at the "main output loop". Here, the task will take a character from the TBUF and insert it into the output ring buffer. This loop will continue until the output ring buffer fills. When this happens, this request will block on output room and the current task will go find something else to do. This request will be blocked until the IOP removes all the characters from the output ring buffer. When the IOP empties the output ring buffer, it will unblock the PIB by setting the output-done flag and wake up the IEW task in the GLOBAL PMGR. This process continues until the write request is satisfied.

In the course of processing the first character, (the first character to enter the ring buffer) the GLOBAL PMGR will post an OUTPUT START request to the IOP. This is accomplished by setting a bit in the request table corresponding to the line of the request and issuing an NIOS to the IOP to tell it that a new request has been added.

In the IOP

When the IOP processes the HOST output start request, it will issue an I/O instruction to start the device for output (turn on the transmitter).

When the device is started, an interrupt will occur immediately. This will cause the interrupt service routine to take the characters out of the output buffer (one character per interrupt). When the output ring buffer is emptied, the IOP will unblock the PIB and start the process of unpending the GLOBAL PMGR's IEW (if necessary). This is done by storing the physical address of the IEW task into PUBTCB.W, clearing PBTCB.W and doing "NIOS HOST".

Back to the HOST

The unblock of the interrupt event waiter is only started by the IOP. The IOP has enough to do with all the I/O interrupts and maintaining the NUNB and NEVEN counts to tell the GLOBAL PMGR task scheduler what task to invoke.

The KERNAL driver (IOPDR), in response to the "NIOS HOST" from the IOP (which causes an interrupt to the HOST), will then unpend the blocked IEW. To do this, IOPDR will check the contents of PUBTCB.W in PMGR page zero. If it is non-zero, then it is the address of the IEW task. The pended bit (?BTPN) in this TCB is cleared and KERNAL location PMGRWU is set to -1. This causes the next pass of the AOS/VS scheduler to pass control to the PMGR IEW.

When things go wrong

Modem disconnects and ^C^X sequences

Modem diconnects are handled by the PMGR because no one else wants it. A modem disconnect is an unexpected termination of the telephone connection due to a variety of reasons:

1) The phone line has become noisy due to bad equipment or a disturbance on the line.

2) A power failure at the remote site

3) The user hangs up the phone

4) The phone bill has not been paid

When this occurs an interrupt is generated that invokes the modem service routine to check the status of the modem. The IOP detects that the modem has been disconnected by a change in the modem status signal DSR. When this signal drops the following is done in the IOP:

1) The transmitter is turned off

2) The modem signals DTR and RTS are lowered (modem is turned off)

3) The data in the input and output ring buffers are invalidated by setting the remove pointer equal to the insert pointer in each buffer

4) An asynchronous event (^C^B for modem disconnect) is posted in the completion table (CTBLE).

5) Any blocked condition in the PIB is unblocked to allow completion of the request (with MODEM DISCONNECT ERROR)

The completion table is set up to allow any or all devices to have a completion posted to the HOST. Only one completion is processed at one time because the IOP must set IOPCPL and IOPLIN in PMGR page zero and do "NIOS HOST" for each completion.

IOP Panics (17002)

When the IOP decides to panic because of a condition it cannot resolve, all it has to do is set IOPPAN to non-zero and interrupt the HOST.

Powerfail

When a power failure is detected in the HOST, the IOP is not notified. The HOST forces the IOP to halt and reads the IOP's PC and AC0. After the HOST saves this information, the IOP is made to start at a known location which contains an "EJMP PWRFL" in IOP code space. This code saves the IOP state and halts.

When the power returns, the HOST will start the IOP at another known location containing "EJMP PWRRS" which will:

1) Reload the IOP map

2) Restart the clock

3) Reset all ATI lines (turn off the receivers and transmitters)

4) Cause a modem disconnect on any modems that were active

5) Re-initialize all ATI lines (turn on the transmitter)

6) Restore previous IOP state that was saved and HALT

Meanwhile the HOST was waiting for the IOP to HALT (IOP busy flag=0). When the HOST detects that the IOP has halted, it restores IOP AC0 and starts it up at the PC that was saved previously.

Kernal - PMGR Interface

Several hooks have been put into the system to support the PMGR and PMGR like processes (SVTA). They are:

1) ?PROC/?TERM/?CHAIN system call processing,
2) Crude connection management,
3) ?SGNL system call,
4) ?DPMGR system call,
5) PMGR panic gate,
6) Scheduler considerations, and
7) PMGR interrupt drivers.

?PROC/?TERM/?CHAIN System Call Processing

The kernal sends the PMGR an IPC message for every ?PROC, ?TERM and ?CHAIN system call.

Currently, only three messages from the system are acceptable:

1) ASCON - handle a ?PROC (assign a console to a process).

2) CHAIN - handle a ?CHAIN, and

3) PTERM - handle a process ?TERM.

The kernal passes these messages while running on a deamon (control block). While the PMGR is processing these requests for the kernal, the deamon is pended, waiting for the PMGR. Deamons are a rather rare item so it is important that the PMGR process these requests AQAP.

The kernal uses a special IPC port into the PMGR: port 0. Normally, port zero is not a valid port number but since the kernal enforces these rules, it can also break them. If the PMGR recieves an IPC on port 0 and from Pid 0, then it knows the message was from the system. Internally the PMGR uses a special PIB, called the dummy pib, to single thread all kernal requests. By single threading, potiential race conditions between the kernal - PMGR and internally to the PMGR are avoided. Further, performance is NOT lost since there is only one CPU and nothing can be gained by parallelism.

When the PMGR has completed the kernal's request, it will unpend the deamon by issueing a ?SGNL system call [see below].

Crude Connection Management

The very fact that the PMGR is told about ?PROCs, ?TERMs and ?CHAINs is roughly equivalent to connection management primitives. Further, the kernal gives the PMGR a chance to cleanup (by pending) before allowing any more processing on behalf of that process.

?SGNL System Call

?SGNL is used to reply to the kernal that its PMGR request has completed. The system call has two arguments: Error code and PID of process on behalf the request was made.

Any ring of any process that has PMGR privilege can issue a ?SGNL system call (basically, anyone who is a direct son of PID 2). The kernal will validate that the target process is pended on a PMGR function. If it is, it will be unpended. Basically, very little validation is performed.

If the kernal returns any error on ?SGNL to the PMGR, the PMGR will panic with a 12300 subcode 5.

?DPMGR System Call

The ?DPMGR system call is similar to the ?IDEF system call. It defines a device as "belonging" to the PMGR. The device must previously be gen'd as a PER device type and the DCT must be setup accordingly. The first successful ?DPMGR establishes that process as the "Master PMGR".

Not only does the ?DPMGR allow the PMGR to ?IDEF device but also allocates and initializes DCH map slots a la ?STMAP. The PMGR passes a table of logical addresses and ?DPMGR will load the DCH slots with the corresponding physical page numbers, returning the DCH slots used.

?DPMGR is basically the PMGR's own system call to do any initialization necessary to the kernal and is subject to change from rev to rev.

The PMGR can force a system panic (12300) by LCALLing into the PMGR panic gate in ring 0.

The kernal will verify that (1) the caller is from a system ring and (2) the caller is the "master PMGR". A race condition exists. If the PMGR wants to cause a panic before it has done the first ?DPMGR, the kernal will not know who is the master PMGR, and will ignore the panic request resulting in a system hang.

The kernal also only allows the PMGR to pass only one AC to the panic routine. Currently that AC is used as the subcode. If would be nice to be able to pass additional information (ie device codes) to the panic routine.

User Requests

A wide variety of user requests to the PMGR are available. Here we will attempt to describe each of them. The requests are seperate into four categories:

1) Miscellaneous,
2) Control, and
3) I/O (Read/Write).

Miscellaneous User Requests

There are four user requests in this category:

1) Initialize local,
2) Get memory for local,
3) Get PMGR statistics, and
4) Reset PMGR statistics.

The first two requests have already been discussed in previous chapters: initialization and memory management.

The GET Statistics request returns the following information:

Characters Read
Characters Written
Control and Miscellaneous Requests
Systems Calls
Devices Currently Assigned
of Spins in LOCK
# of Calls to LOCK
# of Read Requests
# of Write Requests
# of Event Waiters Started

The RESET Statistics zeros out the above information except for the # Devices Currently Assign word.

Control Requests

Control request consist of the following:

1) ?Assign
2) Ascon
3) ?Open
4) ?Close
5) ?Deassign
6) Pterm
7) Chain
8) ?Rterm
9) Get/Set Characteristics
10) Get/Set Delimiter Table
11) Set Time out Constant
12) ?Send Message

?ASSIGN

?ASSIGN allows an user to reserve a device for their exclusive use. A device that is already assigned (either explicitly via a ?ASSIGN or implicitly via ?OPEN or ASCON) cannot be ?ASSIGNed again. That is, you can not ?ASSIGN a console that is already owned. The only way a device can be deassign is by issueing a ?DEASSIGN or by process termination.

A process that has assigned a device is considered to be the owner of the device but the device is NOT considered to be open or a process console. The PMGR prepares the PIB for the new owner by

1) Resetting the delimiter tables to the default,

2) The edit program to the default,

3) Time-out constants to default,

4) Setting the current characteristics to the default characteristics,

5) Setting the device open count to zero,

6) Saving the PID number of the new owner. Finally,

7) The PIB is linked onto the owner's chain via the PID table (PIDTB) so it can easily be found at term/chain time.

ASCON

ASCON explicitly re-assigns a console to a process. There may or may not be a previous owner. The kernal passes to the PMGR the father (current owner) if there is one and the son (new owner) of the console.

If there wasn't a previous owner an explicit ?ASSIGN is performed first.

If there was a previous owner, the PMGR then builds a State Save Area (SSA) for the current owner and links it off the PIB of the console. The SSA contains the pid of the owner, delimiter tables, current edit program, current characteristics, time-out constant and the local open count. Lastly, the PIB is removed from the previous owner owner's chain.

ASCON then initializes the PIB for the new user by setting the delimiter tables to the default, setting the edit program to the default, and setting the process console bit. Lastly, the PIB is linked onto the new owner owner's chain.

?OPEN

?OPEN initializes the device for I/O. If the requestor is not the current owner of the device, the open will fail with an error "Not you device". Two open counters are maintained by the PMGR: Local open count or how many times the current owner has opened the device and Global open or how many times the device has been opened.

If the device is currently unowned then an implicit ?ASSIGN is performed for the requestor. [On final process close the device will be implicitly ?DEASSIGNed]. The device can already be owned by a previous ?OPEN, by an explicit ?ASSIGN or by a ?PROC (ASCON).

OPEN first increments the local open count and the global open count. If this is the first open on the device (global open count goes from 0 -> 1) an initial device open is performed. This is:

1) Input ring buffer is flushed,

2) Any errors on the device are cleared,

3) If the device is modem controlled, ^S is cleared,

4) Finally, the modem is "turned on".

If characteristic bit ?CFF is set, the PMGR will then output a form feed to the device.

?CLOSE

Close is the inverse of OPEN.

On final process close, that is when the local open count goes to zero, if the device was implicitly assigned it is deassigned. If the device was explicitly assigned (either by ?ASSIGN or ?PROC (ASCON)) the device is still owned by the requestor, it is just not open for I/O. Any outstanding I/O requests are aborted and an error is returned to the requestor "I/O Termed by CLOSE".

On final device close (global open count goes to zero) the device is prepared to be shutdown. Since there isn't any outstanding I/O requests (a final device close implies a final process close) the output ring buffer is checked to see if any output is left. If so a bit is set saying to shutdown the device on output completing [this is performed by the interrupt world so the request can continue on]. If there isn't any output, the device is shutdown immediately, ie modem is turned off. A special kludge is put in for PID 2, in that we will pend the request waiting for output to complete since the PMGR is needed to finish the outputting of any termination messages.

If the device is modem controlled, the IAC or IOP will then hold the line closed for at least 15 seconds to give the hardware and the phone lines a chance to settle down.

?DEASSIGN (Release)

Deassign is the inverse of ?ASSIGN. Deassign makes the device available to the rest of the users in the system.

Deassign will not allow you to release the device if the local open count is non zero or if the device is a process console. Deassign removes the PIB from the owner's chain, releases any delimiter tables and releases the edit program if any.

PTERM

PTERM logically does the inverse of the ASCON, ?OPEN or ?ASSIGN, restoring the exact condition of the PIB from the information in the SSA. The kernal passes to the PMGR the PID of the process terming and PID of the new owner (father).
For each device on the owner's chain, all outstanding Control and I/O requests are aborted, and the local open count is passed to the CLOSE routine forcing a final process close. See CLOSE for more details. Since the PMGR maintains the owner's chain, the PMGR can find all the devices owned by the process terming. If the device was explicitly ?ASSIGNed then an explicit ?DEASSIGN is performed. If the device was ?ASSIGN via an ASCON request and there is an SSA, the SSA is restored for the father process. If the device was implicitly ?ASSIGNed via ?OPEN then the final process close will take of the ?DEASSIGN. See ?DEASSIGN.

CHAIN

CHAIN is the same as PTERM, except that ?DEASSIGN is not called.

RTERM

RTERM lets the requestor abort all requests currently in the PMGR on their behalf. Two arguments are supplied: (1) the control port of the device and (2) the user's local port the request originated on.

RTERM is used almost exclusively by the Agent to do Task Aborts in response to ^C^A sequences (?IDGOTO).

Note that control requests are not aborted since they are considered by the PMGR as "unpending" and will complete immediately.

Get/Set Delimiter Table

Get/Set delimiter table is used to specify the delimiter table used for data-sensitive Writes, Reads and Priority Reads. Further, the caller may specify default delimiter tables by passing an address of -1.

Using the default delimiter table saves 16. words of resident memory.

Get/Set Characteristics

Get/Set characteristics is used to specify physical and logical characteristics of the device. This includes such things as baud rate, parity, CPL, LPP, etc. The PMGR maintains two sets of characteristics: current and default.

The current characteristics are modified by the owner. The default characteristics are the ones specified at gen time and are used when characteristics are reset at initial device open time.

The PID 2 CLI can over ride the default characteristics by the "CHAR/DEF/xxx @CONnnn" command. The user can see the default characteristics by the command "CHAR/DEF". The user can reset the current characteristics to the default by the command "CHAR/RESET"

SET Time Out Constant

The user may specify a time out constant for read or writes in the range 2 < TO < 65536. On every key stroke the read time out counter is reset the time out constant. Ditto for output.

?SEND a message

A user may not ?WRITE to a device that is not its own. The PMGR does provide a mechanism where one user may send a message to another user's screen; this is called ?SEND. ?SEND may be used IFF the device is a process console (?PROC'ed) and the characteristic NRM (No Receive Messages) is NOT set. PID 2 can over ride the NRM restriction.

A maximum of 255. characters can be sent in one ?SEND request. The message is always preceded by the string "<NL>From Pid xxx : " where xxx is the pid of the requestor.

Certain control characters are filtered out of send messages in order to prevent malicious users from annoying others. A bit map in the PMGR is used to to specify which characters to filter out.

I/O Requests (Read/Write)

There are two basic types of I/O requests: read and write. Both types of requests provide a wide variety of options and types of I/O.

Usually the current owner is the only process allowed to do I/O to the device. There is one major exception: Father - Son I/O. This where the son owns the device, but the father is allowed to do read and writes also. The father is NOT allowed to do control requests to the device.

Read

There are several flavors or ?READ supported by the PMGR:

1) Data-sensitive reads
2) Dynamic reads,
3) Binary reads,
4) Priority reads,
5) Screen Edit reads.

Screen Edit is perhaps the most popular of all the types of reads. Nearly every DG product uses screen edit reads. Screen edit originated in Line Edit, and became so popular that it was decided to put it in the PMGR. That marked the beginning of the end for the PMGR. In a later discussion we will talk about screen edit.

The first three types of reads are used about evenly. In fact it is even possible to combine binary reads with any other type of read. That is you can have binary data-sensitive reads. When doing a data-sensitive read, you specifiy the delimiter table to terminate the read on using the control request talked about above. For dynamic reads, a delimiter table is not used, but the read is terminated when the number of bytes requested has been read. Needless to say dynamic reads are more efficient that data-sensitive reads since the PMGR does not have to check each character to see if it is a delimiter.

There are a variety of options one can have on a read:

* Don't echo characters typed,
* Don't echo delimiters typed,
* Look for function keys,
* Drop type ahead,
* Position cursor before read,
* Return cursor position after read,
* Disable redisplay of user buffer before read,
* Priority read, and
* Position cursor in user buffer.

The largest possible read the PMGR supports is 511. characters. This means to read the whole screen of most tubes, it will take four reads.

Writes

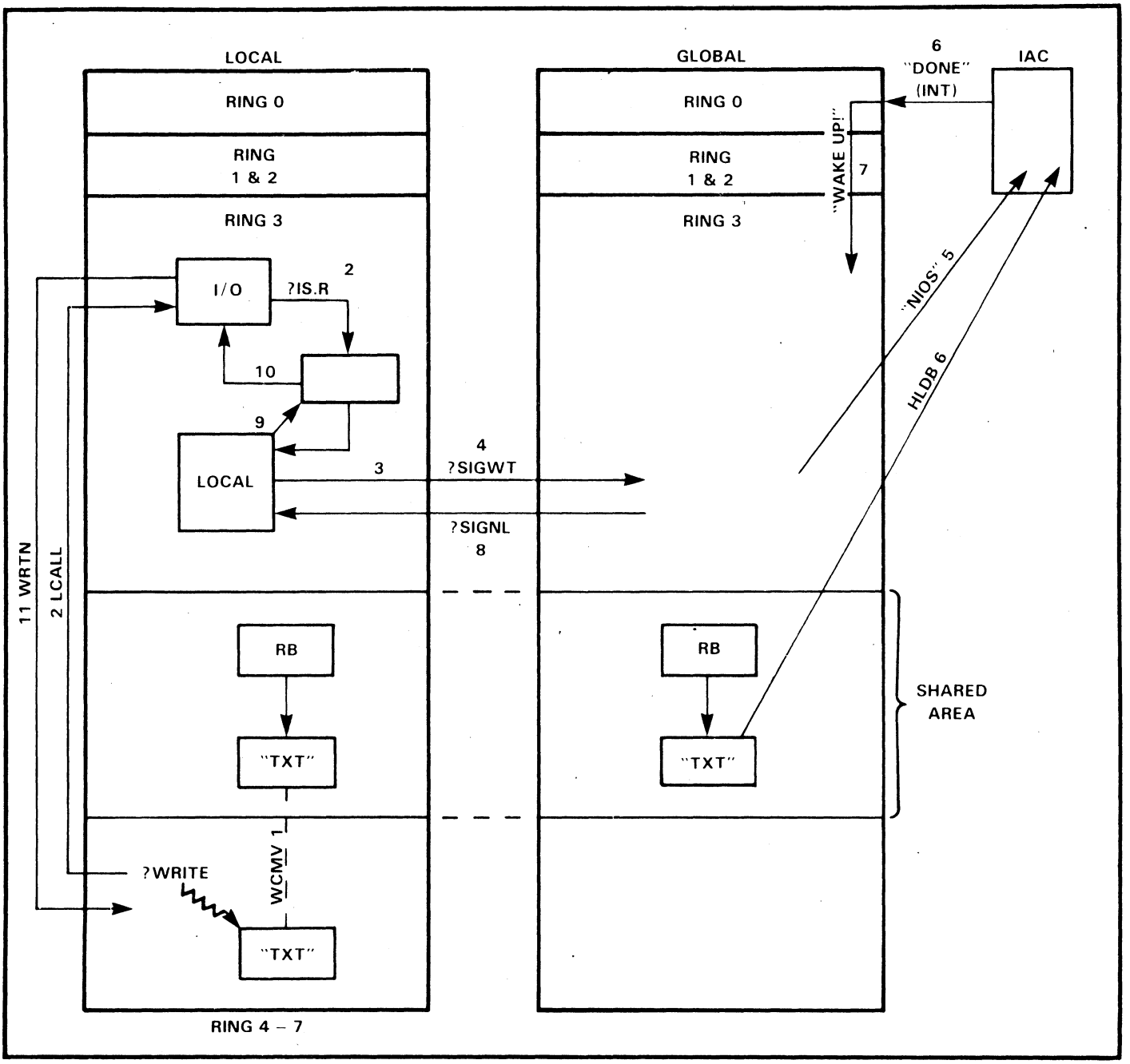
There are only three types of Writes supported by the PMGR:

* Binary writes,
* Data-sensitive writes, and
* Dynamic writes.

Data-sensitive write. unlike reads, are very rarely used. Dynamic writes are used almost exclusively. Occasionally, a use will be found to do binary writes.

?WRITE Action

The following is a description of a ?WRITE system call as processed by the Agent and PMGR. The numbers correspond to the items on the next page below.



CS-03321

1) The user issues a ?WRITE system call, passing a ?WRITE packet. This is transformed by URT_32 into an LCALL to the Agent.

2) The Agent dispatches the system call to the Agent I/O world which translate the ?WRITE packet into a ?IS.R packet suitable for the PMGR. The Agent then does a ?IS.R.

3) The ?IS.R system call processor detects that the IS.R is destined for the PMGR and deflects the call to the local PMGR (label PMGR.ISR). This is done by doing a .GPORT and seeing if the PID is 1. .GPORT is an Agent primitive to do ?GPORT code without incurring the system call overhead.

4) The local PMGR translates the IS.R packet into a RB, allocates a buffer large enough to do the ?WRITE. copies the user's buffer via a WCMV into the new allocated buffer (called a temp buffer), enqueues the RB/buffer onto the Global request queue, ?SIGNLs the global that it has work to do if this is the first RB on the global request queue, and waits to be signal'ed of the completion.

5) The global PMGR removes the RB from the request queue, assigns a stack to the RB, finds the device that the ?WRITE is for by using the local port # and indexing into the shadow port table, sets up the ?WRITE to the IAC by setting the appropiate request bit in the CIB, and issues an NIOS I/O instruction to the IAC. The "WRITE BLOCKed" bit is set in the PIB, and the PIB is enqueued onto the SUNB queue waiting for the write to complete (the RB has a stack during this time).

6) The IAC copies over the portions of the PIB and RB that it needs to do the ?WRITE and maps in the temp buffer. The IAC reads a character at a time from the temp buffer into the ring buffer. After the buffer has been completely moved into the ring buffer. the IAC sets the appropiate completion bit in the CIB and interrupts the HOST.

7) The interrupt world in the HOST scans the CIB looking for bits set in the completion word. Finds the PIB by indexing into the line table for the that line, sets the "WRITE DONE" bit in the PIB and kicks NEVEN. The driver then finds the PMGR's event waiter task, clears the pended bit and then does an .IWKUP primitive on PID 1. The scheduler eventually runs the PMGR's event waiter task who searches the SUNBQ looking for the PIB which has a done bit set. Finds the PIB and resumes the code path.

8) The global PMGR then ?SIGNLs the local PMGR that the ?WRITE has completed, which wakes up the local PMGR. De-assigns the stack from the RB (returns the memory to the free memory pool) and the task returns to the scheduler.

9) The local PMGR then translates the RB back into the ?IS.R packet for the Agent and takes the good return to the ?IS.R code. In the case of a ?READ, the local would move via a WCMV the temp buffer into the user's buffer.

10) The ?IS.R takes the good return to the Agent I/O code.

11) Agent I/O then translates the ?IS.R packet back into a ?WRITE packet and takes the good return from the LCALL into the user's ring.

And the ?WRITE is completed.

Screen Edit

The Peripheral Manager (PMGR) supports several types of ?READs for asynchronous character devices. The screen edit type ?READ, hereafter referred to as screenedit, allows users to specify initial cursor positioning and data display information, to use screen control characters to edit data in the current input buffer, and to receive information such as delimiter type and cursor position upon ?READ termination. Several screenedit features may also be selected on a ?WRITE.

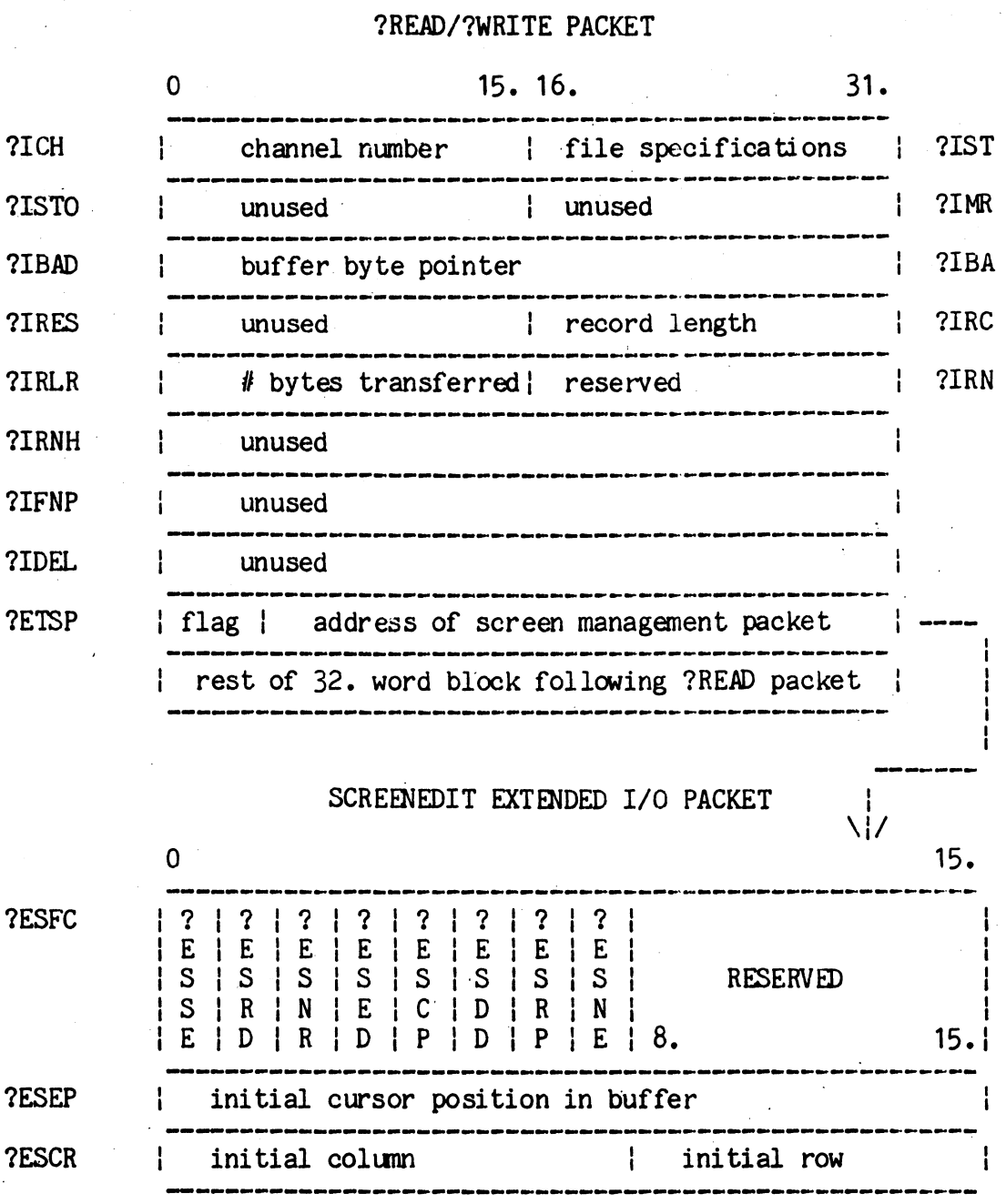
How to Issue a Screenedit I/O Request

Using ?READ/?WRITE System Calls

For purposes of clarity, only screenedit ?READ requests will be addressed in the following section. The procedure to issue a screenedit ?WRITE is the same, although only a subset of screenedit options are applicable to ?WRITE requests.

In order to issue a screenedit ?READ, a three-word screen management packet must be supplied along with the normal ?READ packet. A 32. word block must be reserved directly after the ?READ packet in your user program. The word address of the extended packet should be stored in offset ?ETSP of the ?READ packet. In order to enable extended packet processing, bit ?IPKL of word ?ISTI must be set in the ?READ packet. On the first screenedit ?READ issued, bit zero of word ?ETSP must also be set. Setting this bit establishes the screenedit extended packet supplied with this ?READ as the 'default' extended packet. This packet will be used on all subsequent screenedit ?READs and ?WRITEs until a new packet is established by providing the new packet address in ?ETSP and setting bit zero so that the packet will be read. The AGENT maintains a copy of the default packet so that it is unnecessary to specify a new extended packet on the next I/O request if screenedit functions which modify one or more of the extended packet locations (such as return cursor position) are selected for the current I/O request. The ?READ/?WRITE and screenedit extended packet formats are presented in Figure 1. Screenedit options are discussed in section 3.

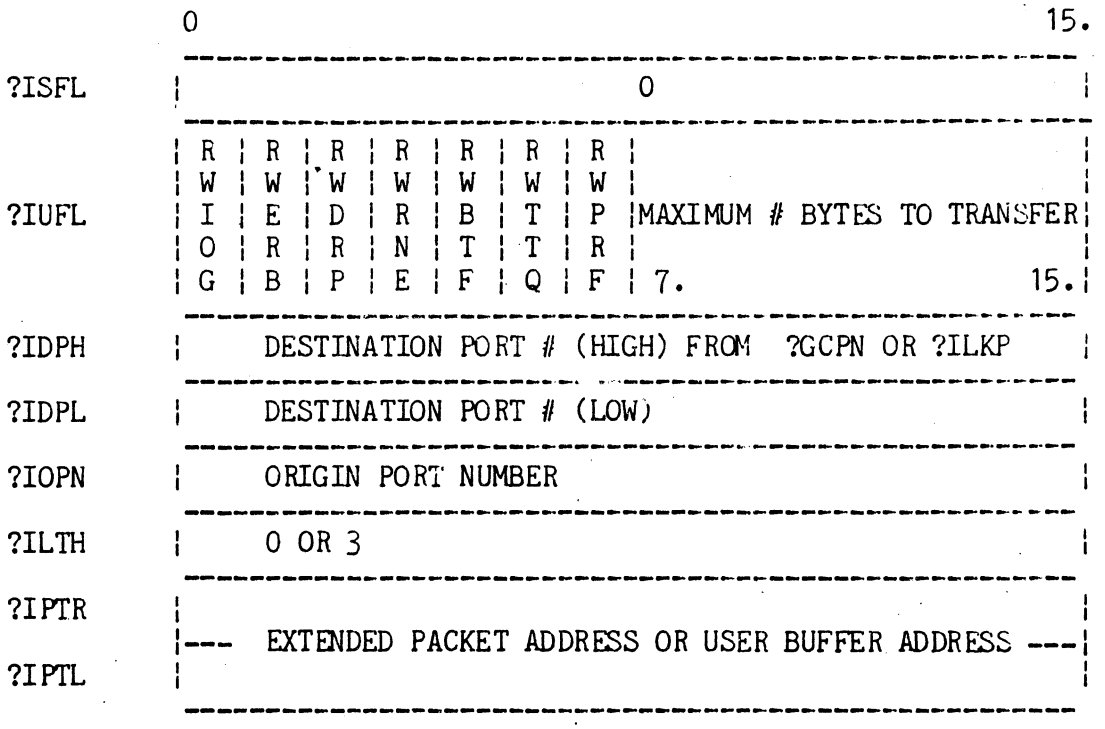
Figure 1: SCREENEDIT DATABASES



Screenedit Data Base Pre-Processing

Before the PMGR can process a ?READ or ?WRITE request, the I/O packet. and optionally. the screenedit extended packet, must be converted into IPC format by the AGENT. This structure is outlined in Figure 2.

Figure 2: CONVERTED IPC PACKET FOR PMGR



?IUFL flags:

RWIOG - input: unused
output: reserved

RWERB - input: don't echo delimiters
output: error bit

RWDRP - input: drop typed-ahead characters
output: ?READ ended with a function key sequence

RWRNE - input: do not echo any characters entered
output: reserved

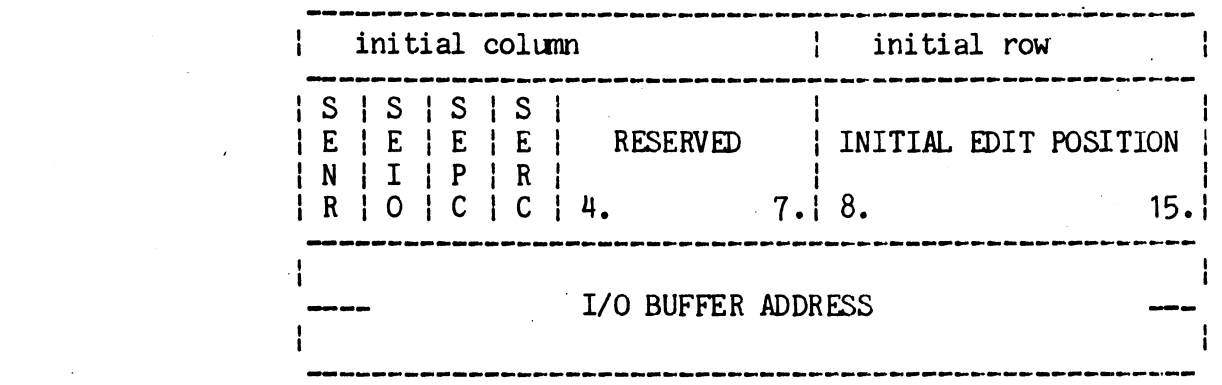
RWBTF - input: binary ?READ (specifying binary mode disables screenedit control character intrepretation. In addition, no characters will be echoed even if RWRNE is not set).
output: reserved

RWTTQ - input: ?READ will not terminate until the number of characters specified in word ?IRCL of the ?READ packet is entered (or an error is encountered). If RWTTQ is not set then the ?READ will terminate when a delimiter is entered. If the buffer is filled before a delimiter is encountered then the ?READ will terminate with a line-too-long error (ERLTL).
output: reserved

RWPRF - input: this ?READ is a priority ?READ (a priority ?READ to a device will preempt an active non-priority ?READ for the same device after the current character has been processed).
output: reserved

On a ?WRITE request. bits 6. - 15. of ?IUFL contain the maximum number of characters to output. For a ?READ, bits 7. - 15. of ?IUFL contain the maximum number of characters to read.

CONVERTED SCREENEDIT PACKET FOR PMGR*



SENR - do not predisplay the user's buffer. If SENR is not set then bits 8.- 15. of the extended packet contain the position within the input buffer to leave the cursor after the buffer contents are predisplayed.
SEIO - enable screenedit control characters
SEPC - position the cursor before the ?READ to the initial column and row stored in offset zero of this packet
SERC - return cursor position to user at request termination

Note: * The extended packet is supplied to the PMGR only if one or more of the following screenedit options is specified: initial cursor positioning, return cursor position or enable screenedit control characters. If the extended packet is supplied, then IPC packet offset ?ILTH is three and offset ?IPTR contains the address of the converted screenedit extended packet. Otherwise, ?ILTH is zero and buffer address is stored in ?IPTR.

Screenedit Options

The following options are available on a screenedit ?READ.

?ESSE

The ?ESSE option enables screenedit control character processing for the current ?READ request on non-hardcopy devices. Selecting this screenedit option on a hardcopy device will result in a ?READ error. The following characters have special significance if ?ESSE is selected:

CNTL-A - move cursor to the end of the current line and output a space if the current character position does not contain a DEL (177) character. Outputting a space allows users to distinguish between the current screen display and data displayed during previous I/O requests

CNTL-B - move cursor to the end of the previous word, or to the beginning of the line if the cursor is currently positioned within the first word. To find the end of the previous word, the cursor is moved left to the first character which is not a space or a tab

CNTL-E - enter/exit insert mode. While in insert mode the current insert position is marked by a 'hole' on the screen. This 'hole' is closed when another CNTL-E is entered, which resets the current input mode to overstrike mode, or when the ?READ terminates

CNTL-F - move cursor to the first character of the next word in the line, or to the end of the current line if the cursor is currently positioned within the last word. To find the beginning of the next word the cursor is moved right to the first character which is not a space or a tab. If the cursor is moved to the end of the line, a space is output if the current character position does not contain a DEL (177) character

CNTL-H - move cursor to the beginning of the line (striking the 'home' key on D200-compatible terminals performs the same function)

CNTL-I - insert a tab at the current cursor position. The rest of the line is shifted right to the next tab stop. Tab stops are located in columns 8, 16, 24, 32. 40. 48, 56, 64, and 72 of each physical line on the screen. Striking the 'TAB' key on D200-compatible terminals performs the same function

CNTL-K - erase all characters displayed on the screen from the current cursor position to the end of the line

CNTL-M - (carriage return) erase all characters displayed on the screen from the current cursor position to the end of the line. then terminate the ?READ

CNTL-U - erase this line from the screen

CNTL-X - (for non-4010I devices)
move the cursor one character to the right. Striking the -> key on D200-compatible devices performs the same function. If the cursor moves past the end of the current line a space will be appended to the input buffer and echoed on the screen, unless the current character position contains a DEL (<177>) character. In this case no space will be echoed. Trying to enter CNTRL-X when the current character is a DEL will cause the ?READ request to auto-terminate.

CNTL-Y - (for non-4010I devices)
move the cursor one character to the left. If the cursor is positioned at the beginning of the buffer the bell is sounded. Striking the <- key on D200-compatible devices performs the same function.
(for 4010I devices)
functionally equivalent to CNTL-X

CNTL-Z - (4010I devices only)
functionally equivalent to CNTL-Y for non-4010I devices

If ?ESSE is not selected on a screenedit ?READ, CNTL-I and CNTL-U characters function as outlined above. The other control characters are echoed according the current device characteristics. The default echo mode for control characters is an up-arrow (^) followed by the upper case character entered, ^H for example.

?ESRD

Selecting screenedit option ?ESRD causes the contents of the buffer pointed to by offset ?IBAD of the ?READ packet to be displayed on the screen before ?READ processing begins. Data is displayed up to the first delimiter or DEL (<177>) character in the buffer, if one is found, otherwise the entire buffer will be displayed. Note, however, that this is an error condition and the ?READ will shortly terminate with a line-too-long error (ERLTL). After predisplaying the data, the cursor is positioned after character N on the screen, where N is the initial offset in the buffer to leave the cursor. N is supplied in word ?ESEP of the screenedit extended packet. If N is zero then the cursor is positioned at the beginning of the line. If N is greater that the number of characters predisplayed, then the cursor is positioned after the last character displayed on the screen. ?ESRD is ignored if ?ESSE is not set.

?ESNR

Option ?ESNR should be selected if typed-ahead characters are to be ignored on this ?READ. If ?ESNR is set then the PMGR will flush the input ring buffer during ?READ initialization. ?READ processing will begin with the first character entered after the ring buffer is flushed.

?ESED

If ?ESED is selected then delimiters and auto-terminators (characters entered on top of a DEL (<177>) character in the buffer) are not echoed on the screen. Otherwise these characters are echoed normally. 'Delimiters' in the screenedit ?READ context include function key sequences if the ?CFKT characteristic (function keys are input delimiters) is set, auto-terminators and any character flagged in the input delimiter table with the following exceptions:

- CNTL-D characters will terminate the ?READ with a end-of file error if ?ESSE is not set, or if ?ESSE is set and CNTL-D is not flagged in the input delimiter table

- carriage returns are always treated as delimiters, if ?ESSE is set

In the screenedit ?WRITE context, a 'delimiter' is any character flagged in the output delimiter table as being a data-sensitive delimiter. Note that ?ESED does not apply to any characters other than those included in the above definitions for screenedit ?READ and WRITE delimiters.

?ESCP

If option ?ESCP is selected then the cursor will be positioned before ?READ processing begins. Word ?ESCR of the screenedit extended packet contains the initial cursor column (left byte) and row (right byte). If ?ESCP is not selected then the cursor is left in the current position on the screen. Contrary to previous documentation, ?ESCP is valid even if ?ESSE is not set.

?ESDD

?ESDD is actually a flag returned by the PMGR rather than a user-selectable option. If device characteristic ?CFKT (treat function keys as input delimiters) is set and a function key in entered, then the screenedit ?READ terminates normally (provided there is enough room in the user buffer to store the function key header and the second character in the function key sequence) and ?ESDD is set in the screenedit extended packet.

?ESRP

Upon ?READ termination, if ?ESRP is selected, then the current cursor position is returned to the user in word ?ESCR of the screenedit extended packet (left byte - current column, right byte - current row). If option ?ESCP is also selected, then the row returned is the physical row on the screen (row numbering is zero-relative) otherwise it is the row relative to the starting row. This value is also zero-relative. The column position returned is always relative to the current row.

?ESNE

If option ?ESNE is selected then characters entered during this ?READ are stored in the input buffer but are not echoed on the screen.

Record Formats and Screenedit I/O

Screenedit ?READs with the ?ESSE option set are inherently datasensitive and are treated as such by the PMGR, regardless of the record format specified in word ?ISTI of the ?READ packet. The only exception to this is binary mode, in which case the ?ESSE option is ignored. If ?ESSE is set. a copy of the N character buffer (where N equals the buffer length in ?IRCL) pointed to by ?READ packet offset ?IBAD is moved into the PMGR before request processing begins. This buffer must contain either a delimiter (one present in the input delimiter table) or one or more consecutive DEL characters or the ?READ request will be rejected on a line-too-long error. If ?ESRD is set. the substring beginning with the first character in the buffer and continuing up to but not including the first delimiter or DEL in the buffer will be displayed on the screen. Note that this substring can be null, i.e. the first character in the buffer is the delimiter or DEL. If ?ESRD is not set. the data will not be predisplayed, although entering CNTL-A will have the same effect as setting ?ESRD. Entering CNTL-X will 'uncover' the character in the buffer at the current cursor position.

If ?ESSE is set. the screenedit ?READ must be terminated with a delimiter (unless the buffer contains a DEL character. see Section 6 for more information). This means that the maximum record length specified in ?IRCL should account for at least one delimiter. two if the 'function keys are input delimiters' characteristic is set (?CFKT). If ?IRCL characters are entered before a delimiter is encountered, the ?READ will terminate with a line-too-long error. In this case, the last character entered is not (!) considered to be a delimiter and is therefore echoed on the screen unless ?ESNE is set.

If ?ESSE is not set on a screenedit request, fixed- or even variable-length record formats may be specified. In this case ?IRCL is the absolute record length and the I/O request will terminate after ?IRCL number of characters have been transferred.

Figure 3 summarizes the screenedit options which may be selected by I/O request type.

Figure 3: Valid Screenedit Options by I/O Request Type

	BINARY READ	BINARY WRITE	DATASENS READ	DATASENS WRITE	FIXED READ	FIXED WRITE
?ESSE	X	X	*	X	see note 4	X
?ESRD	X	X	see note 3	X	see note 4	X
?ESNR	*	X	*	X	*	X
?ESED	X	X	*	X	see note 4	X
?ESCP	*	*	*	*	*	*
?ESDD	X	X	*	X	see note 4	X
?ESRP	see note 1	see note 1	*	*	*	*
?ESNE	see note 2	X	*	X	*	X

* = valid option
X = option ignored

notes:

1) by definition, the PMGR does not maintain an updated cursor position while processing characters during binary I/O. Therefore the cursor position returned, if ?ESRP is selected, will be the initial position specified in the extended packet, if ?ESCP is set, or the PMGR default initial cursor position.

2) by definition, characters entered during a binary ?READ are not echoed to the device, therefore setting ?ESNE on a binary ?READ is redundant.

3) ?ESRD is ignored unless ?ESSE is set

4) complex situation. Read this section.

Screenedit Error Conditions

Screenedit I/O requests terminate normally when:

1) a delimiter or auto-terminator is entered for a ?READ with ?ESSE set

2) a delimiter is entered for a data-sensitive ?READ or ?WRITE. regardless of ?ESSE setting

3) ?IRCL characters are entered on a fixed length or binary ?READ or ?WRITE if ?ESSE is not set

The following is a summary of possible errors returned from a screenedit I/O request:

ERROR MNEMONIC	OCTAL CODE	ERROR CONDITION
ERRAD	(33) -	trying to ?READ from an output only device
ERFNO	(2) -	trying to ?READ from a device you do not own
ERFIL	(75) -	trying to perform a screenedit ?READ with screenedit control characters enabled (?ESSE) on a hardcopy device
ERLTL	(67) -	if screenedit control characters are enabled (?ESSE), the initial buffer pointed to by offset ?IBAD of the ?READ packet does not contain a delimiter or a DEL character in the first N positions, where N is the record length from offset ?IRCL of the ?READ packet
	-	the record length was exceeded before a delimiter was encountered, or before the second character of a two character function key sequence was received on a ?READ, if function keys are input delimiters.
EREOF	(30) -	end-of-file character (CNTL-D) entered
ERISO	(234)-	invalid screenedit options specified
	-	the flag bit in word ?ETSP of the ?READ packet is not set on the first screenedit request issued

Auto-Terminating Screenedit ?READS

Screen management primitives can also be used to enable auto-termination of a ?READ. Auto-terminating ?READS do not require a delimiter to be entered before the system call completes and can also be used to protect data displayed on the screen during a previous ?READ/?WRITE. In order to perform an auto-terminating ?READ, you must set bit ?ESSE in word ?ESFC of the screen management extension packet. The data buffer pointed to by offset ?IRCL of the ?READ packet must contain two consecutive DEL characters (<177><177>) at the end of the buffer. The number of character positions in the buffer before the first DEL character defines the length of the auto-entry field. The ?READ will terminate normally when either a full field of characters is entered, a delimiter is typed, or if the cursor is positioned over the DEL character in the buffer and a non-screenedit control character is entered. The only situation in which entering a full field does not cause the ?READ to terminate is if data is currently being entered in insert mode. While in insert mode, if entering a character fills the field then a bell will be echoed and insert mode will be turned off. The ?READ will continue in overstrike mode. If the user attempts to enter insert mode when the field is full then a bell well be echoed. The ?READ will continue in overstrike mode.

If you enter a delimiter to terminate the ?READ, the delimiter is stored in the data buffer after the last character in the buffer which is visible on the screen. For example, if you specified the predisplay option (bit ?ESRD) or positioned the cursor at the end of the buffer by issuing a screen edit control command, the delimiter will overwrite the first DEL character in the buffer. If a function key delimiter is entered, both DEL characters are overwritten.

When the cursor is positioned over a DEL character at the end of the buffer. the next character, or character sequence in the case of a function key, entered is treated as an auto-terminator, if the character is not a screen edit control command, or if the character is one of the following screenedit control characters: CNTL-I, CNTL-X. Auto-terminators are handled like delimiters: they cause the ?READ to terminate. they are returned to the user in the data buffer and they are not echoed if bit ?ESED of word ?ESFC in the extension packet is set.

Developers should note that issuing auto-terminating screenedit ?READs does not present an iron-clad guarantee that the user will not be able to position the cursor beyond the current data field on the screen since there can exist a one-to-many correspondence between the character entered from the keyboard and the resulting character sequence echoed to the device (for example, control characters are usually echoed as an up-arrow ^ followed by the printable rendition of the ASCII character). In order to provide maximum protection for the screen display, any character which could be echoed as a character sequence should be flagged as a delimiter in the input delimiter table. Tabs should be delimiters as well.

Screenedit Support for Terminal Types

Most of the information contained in this section applies not only to screenedit requests but to all modes of PMGR I/O (binary mode excluded). Due to numerous inquiries we have received regarding the issue of terminal support, and due to the fact that terminal type and terminal device characteristic settings can affect screen-edit functionality, it seems appropriate to include the following two sections in this document.

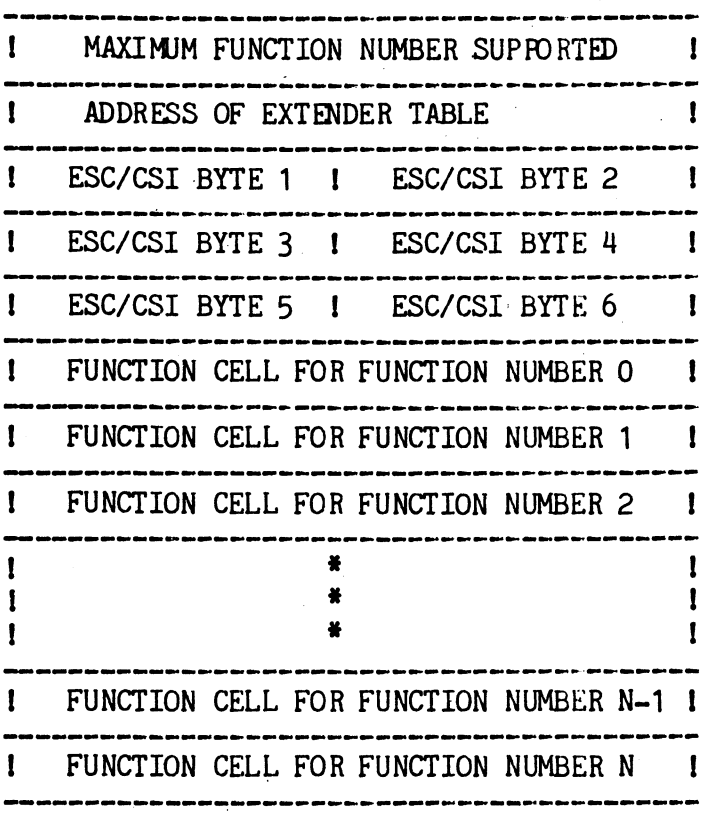
Most screenedit control functions involve changing the rendition of the current screen display, either by positioning the cursor elsewhere in the buffer. inserting or deleting single characters, or erasing a portion or the entire line from the screen. These control functions are normally accomplished by outputting a sequence of control characters to the terminal which indicate the function to be performed, and optionally, one or more parameters which are needed to complete the control function. Since these control function sequences are terminal-dependent. i.e. the character sequence which tells terminal type A to delete a character may be different from the sequence telling terminal type B to perform the same function, a set of 'Terminal Dependent Function Tables' (TDFT's) have been established. TDFT's allow the PMGR to determine the appropriate control sequence to output to a device when a terminal dependent control function is to be performed. The following section briefly outlines the TDFT format and how TDFT's are used by the PMGR.

The following terminal dependent functions are currently supported by the PMGR:

FUNCTION NUMBER	FUNCTION	FUNCTION NUMBER	FUNCTION
TF0	HOME	TF10	DELETE CHARACTER
TF1	CURSOR LEFT	TF11	FREEZE WINDOW
TF2	CURSOR RIGHT	TF12	THAW WINDOW
TF3	ERASE EOL	TF13	CLEAR ALTERNATE MARGINS
TF4	CURSOR UP	TF14	SET ALTERNATE MARGINS
TF5	CURSOR DOWN	TF15	reserved
TF6	WRITE CURSOR ADDRESS	TF16	MARGIN INSERT LINE
TF7	reserved	TF17	MARGIN DELETE LINE
TF8	INPUT FUNCTION KEY HEADER		

The format of a Terminal Dependent Function Table is presented in figure 4.

Figure 4: General TDFT Format



ESC/CSI BYTES 1 - 6: these are escape or control sequence introducer bytes which are output to the terminal to indicate that an escape or control sequence follows

FUNCTION CELL: a function cell contains all the information necessary to initiate a specific control function. If a single cell cannot hold enough information to complete the function cells may be linked together via the use of an extended function table. The address of this table is stored in offset 1 of the primary TDFT.

EXAMPLE for type CRT7 terminal:

```
                DCRT7:
000000 PD 000003        TF3
000001 PD  00000000000  0               ;ADDR OF EXTENDER TABLE IS ZERO
000003 PD 017000        36*400+0        ;<CSI1 = 036><CSI2 = 000>
000004 PD 000000        0*400+0         ;<CSI3 = 000><CSI4 = 000>
000005 PD 000000        0*400+0         ;<CSI5 = 000><CSI6 = 000>
000006 PD 000010        CCBS              ;TF0- HOME
000007 PD 020031        1B(FCC1)+CCEM   ;TF1- CURSOR LEFT
000010 PD 020030        1B(FCC1)+CCCAN  ;TF2- CURSOR RIGHT
000011 PD 000013        CCVT            ;TF3- ERASE END OF LINE
```

For this terminal the highest function number supported is TF3 as indicated as the first word in the table. There is no extender table so the 16. bit address is zero. The terminal will require a control sequence introducer 036 octal to be output as the introducer byte for a function so it is defined as the first CSI/ESC byte in the next word. This is the only one required so the second byte in the word is zero and the next two words are also zero. The next word is the first supported function for the device and this terminal defines 10 octal as the home character, no introducers are required so the left byte is zero. The next function supported is the cursor left function and this function requires the control sequence introducer to be output as part of the sequence to perform the function. This terminal defines a left cursor as a two character sequence of 036 followed by a 031 to perform the left cursor. The cell defined for this function has the FCC1 bit set to indicate that the 036 must be output before the 031.

TDFT's provide a flexible and efficient mechanism for supporting different terminal types. In order to optimize performance, screenedit tries to utilize terminal dependent function sequences while performing screenedit control functions. What happens if, while processing a screenedit control function, a particular terminal dependent function is needed but not supported by the device? In most cases, the PMGR will try to simulate the function for the device. The following section briefly discusses how the PMGR uses TDFT's for each possible screenedit control function.

CNTL-A: (CNTL-A is not currently supported by a TDFT function) The line is redisplayed from the current cursor position up to the last visible character displayed on the screen (or up to the first delimiter in the buffer, if ?ESRD was not set)

CNTL-B: (CNTL-B is not currently supported by a TDFT function) CNTL-B functionality utilizes the 'Cursor Left' function.

CNTL-E: CNTL-E is supported by TFDT function TF9. If TF9 is not supported by the terminal, a space is output at the current cursor position and the cursor is moved left under the space to indicate the current insert position (note - this uses the cursor right function). The rest of the line is then redisplayed. This is repeated each time a character is entered while in insert mode.

CNTL-F: (CNTL-F is not currently supported by a TDFT function) CNTL-F functionality utilizes the 'Cursor Right' function.

CNTL-H: ('Home' in this context is not supported by a TDFT function). The cursor is positioned at the starting row and column of the I/O request. See cursor positioning for further details.

CNTL-I: (CNTL-I is not currently supported by a TDFT function) If the simulate tabs characteristic (?CST) then a tab character is output to the device. Otherwise spaces are output until the cursor is positioned at a tab stop. At this point the remainder of the line is redisplayed on the screen.

CNTL-K: CNTL-K uses successive 'Erase End of Line' functions one for each time the line has wrapped on the screen. See CNTL-U for further details.

CNTL-U: CNTL-U is supported by TDFT function TF3 - 'Erase End of Line'. If TF3 is not supported by the terminal, an attempt is made to position the cursor at the starting column and row of the I/O request. If this fails, ^U <NL> is output to the device and the simulation is complete. Otherwise the current line is 'erased' by overwriting the current screen display with spaces.

Cursor
Left: Cursor Left is supported by TDFT function TF1. If TF1 is not supported by the terminal, this function is a no-op.
Cursor
Right: Cursor Right is supported by TDFT function TF2. However, screenedit does not utilize TDFT support for this function. To move the cursor right one character the character on the screen directly to the right of the cursor is redisplayed.

Position
Cursor: Position Cursor is supported by TDFT function TF6. If TF6 is not supported by the device, then an attempt is made to perform a 'Home' function (TF0 - move cursor to upper left hand corner of the screen). Successive new line characters are output to position the cursor to the starting row followed by successive 'Cursor Rights' to position cursor in the starting column in this row.

Screenedit Support for Device Characteristics

As mentioned in the previous section, PMGR support for device characteristics when a screenedit I/O request is issued does not differ significantly from characteristic support for non-screenedit I/O. Potential screenedit application developers and users should, however, be aware of the impact upon screenedit functionality of setting or not setting the device characteristics discussed in the following section.

?CSFF - SIMULATE FORM FEEDS

The 'simulate form feed' characteristic is currently restricted to hardcopy devices only and, if specified for CRT devices, may result in an incorrect cursor position being returned to the user if a form feed is encountered during the I/O request. This restriction is necessary due to different methods of cursor position maintenance for hardcopy and CRT devices.

?C8BT - 8-BIT ASCII CHARACTERS

Screenedit ?READ results may be unpredictable if the user buffer contains characters with the high bit set when the ?C8BT characteristic is not set and screenedit option ?ESSE is set. Briefly, when a character is entered from the keyboard or echoed on the screen, the PMGR strips the high bit if ?C8BT is not set. Screenedit, however, assumes that no characters in the input buffer have the high bit set, if ?C8BT is not selected (remember that this buffer has been copied from the user program and data within the buffer is accesible during the current ?READ). This causes problems, for example, if a CNTL-B or CNTL-F is entered and the initial buffer contains <211>'s rather than TAB characters (<11>'s). This issue is currently under investigation by the PMGR section.

?CEOL - TRUNCATE LONG LINES

The 'truncate long lines' characteristic suppresses character output once a full line of characters has been output to the device. A 'line' in this context is defined as N characters where N is number of columns per line currently defined by device characteristic CPL. Output resumes after a <CR>, <NL> or <FF> is encountered, or upon receiving another I/O request. Users should be aware that setting ?CEOL does not truncate the data in the input buffer on a ?READ. All characters entered during the request are returned in the buffer, even if they are not echoed on the screen.

?CULC - ACCEPT BOTH UPPER AND LOWER CASE INPUT

If ?CULC is not set then all lower case characters entered are converted to upper case before being echoed. Both screenedit and non-screenedit ?READs will convert lower case characters to upper case before checking to see if the character is a delimiter if ?CULC is not set. Users should note that currently no conversion is performed on the characters in the input buffer copied from the user program if ?ESSE is set. Therefore it is possible for lower case characters to be echoed to the device if the predisplay user buffer option (?ESRD) is set. This issue is under investigation by the PMGR section.

?CWRP - HARDWARE WRAP

All screenedit control functions support multiply-wrapped lines, provided that the ?CWRP characteristic is set. If ?CWRP is not set, the PMGR executes a 'software wrap' by outputting a <NL> (or <CR> <NL> sequence for non-ansi-standard devices) each time a full line of characters is output. Software wrapping will only work correctly if the number of columns per line (CPL) is accurately specified for the device. Screenedit control functions will work for multiply-wrapped lines on a device without the hardware wrap characteristic provided that the device supports the 'cursor up' function (TF4). If 'cursor up' is not supported, the PMGR will perform as much of the screenedit control function as possible, then output a bell to indicate that hardware restrictions prevent the function from completing normally.

Programming tips for the user

To maximize your I/O thruput in the PMGR, you should try to do the following things:

* Minimize the number of I/O requests, by
* using as large I/O buffers as possible,
* do dynamic I/O rather than data sensitive I/O,
* if possible, use binary dynamic I/O, since it is even faster, and
* gen the appropiate size ring buffers for your lines.

Miscellaneous

Shared Consoles

Shared consoles is a mechanism where several consoles can own the console at one time. Setting a characteristic bit enables the sharing of a console. After that, each process that is ?PROCed with the console will be considered an owner of the console, but not the interruptable or master owner. Only the process that orignially set the shared bit can clear the bit, since is considered the interruptable owner. Further, when the user types a ^C^x sequence, that sequence is sent only to the interruptable. It is the interruptable owner's responsibility to pass the sequence on to whoever. A new system was coined, ?CISND, to allow the interruptable owner to do this.

The PMGR implements shared consoles by hanging a chain off of the PIB called the shared console chain. Whenever a request comes in for that PIB, and the requestor is not the interruptable owner, the PMGR scans the chain to see if the request was from one of the shared owners. If it was then the request is honored. The shared console chain is part of the SSA area so several sub-trees of shared owners may exist.

TDFT

TDFT is the Terminal Dependent Function Table. This table is defines cursor control sequences for each unique terminal in the system. The table is used for ALL cursor control sequences such as up, down, left, right, delete, insert a character, delete a character, clear screen, etc.

The user fills in the TDFT by patching the PMGR. There is one TDFT area for each CRT type (1-16). CRT3, CRT4 and CRT6 are already defined for 605x, 4010I, and 6130 terminals. All other areas are undefined and available for the user.

Modem Control

Modem control involves the control of several signals. After the signal name is either modem or host; this tells which device that controls the signal.

a) RI - Ring Indicator (modem). A signal that the phone is ringing.

b) CD - Carrier Detect (modem). The two modems are "talking" to each other.

c) DSR - Data Set Ready (modem). The modem is alive, ie ready.

d) DTR - Data Terminal Ready (host). The PMGR is alive.

e) RTS - Request To Send (host). The PMGR has something to send.

f) CTS - Clear To Send (modem). The modem is willing to accept characters.

To start a modem connection, the line is first ?OPENed and a ?WRITE is performed. The host then waits for some one to call up by monitoring RI (ring indicator). When a ring is detected, the PMGR asserts DTR (data terminal ready) and RTS (request to send) meaning that the host has data to send to the modem. Optionally, RI can be ignored if the characteristic bit ?MRI is not set; in that case DTR/RTS will be asserted when the initial device is open is performed.

The modem will acknowledge seeing DTR/RTS going high by raising DSR (data set ready). At this point, the two modems will try to establish a carrier between them (this is that loud squeal you here when you first call up). Once the carrier has been established, the modem will raise CD (carrier detect).

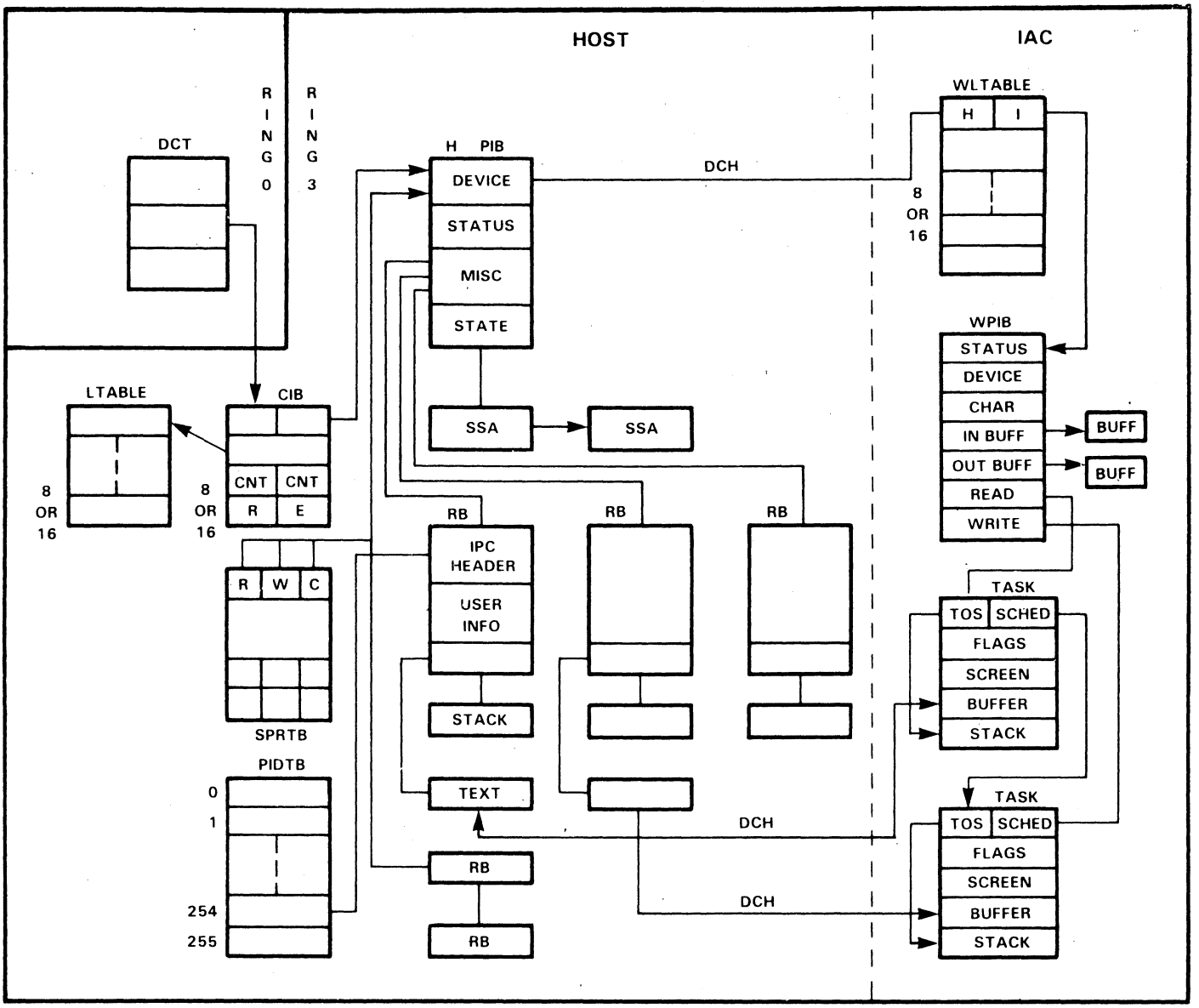
If during the conversion carrier should go away for more than five seconds (ususally due to a noisy line), the PMGR will time-out and force a disconnect.

If the modem ever lowers DSR (data set ready), the PMGR will force a disconnect.

If the device is closed for the last time (final device close), the PMGR will force a disconnect.

A disconnect is forced by lowering DTR (data terminal ready) and RTS (request to send) and waiting for DSR and CD to go low (if they haven't already). Once DSR/CD go low, the PMGR will not raise DTR/RTS for at least fifteen seconds in order to give the modem and phone line a chance to "clean up". Note the PMGR will wait forever on DSR/CD going low. If another user tries to ?OPEN the line before DSR/CD goes low an error will be returned "Modem busy - Can't open line".

Overall Diagram of IAC interaction with PMGR.



CS-03323

Chapter 9 - CLI

Introduction

The general topics discussed in this chapter will deal with the CLI's structure and command processing. An example will be used to explain some basics regarding the operation of CLI.

The Structure

Overview

The structure of CLI can be broken down into four categories: its tasks, its size, stack and operating environment.

There are three types of tasks - primary, ^C^A interrupt and utility. The primary task of CLI is responsible for issuing a read from @INPUT, transferring the input to stack and editing, expanding any macros, validating switches and arguments with command descriptors and executing the command. The ^C^A interrupt task is responsible for taking care of interrupts from the primary (main) and utility tasks, resetting the stack, list flag, command input pointer and output buffer pointer and issuing error message. The utility task coordinates the handling of I/O buffering.

The sizing of CLI is determined by the number ZREL and NREL locations used. ZREL locations will contain processing mode flags, temporary values and pointers. NREL locations are divided into shared and unshared code. The unshared area contains the buffers, packets, stacks and names. The shared code region contains the reentrant programs.

The stack structure is set up to independently handle each of the tasks of CLI. Additionally, stack space is allocated for CLI initialization.

CLI's environment is made up of databases containing buffers, lists, filenames and flags controlling the various processing modes. Level control through the use of the stack is also part of CLI's environment.

Tasks

Upon initial entry into CLI, initialization takes place. The stack gets initialized. The @INPUT and @OUTPUT packets get opened for the primary (main) task. Initialization packets are invoked for ^C ^A and utility tasks followed by CLI's initial message CLI then enters the primary task.

Primary task

The following outline defines the basic steps in CLI's primary task.

1. Type CLI header message
2. Set-up the input buffer
3. Close temporary list file if open
4. Execute prompt commands if any
5. Read line from @INPUT (.ISTR)
6. Push contents of input buffer onto the stack
7. Initialize variables controlled by global switches
8. Check for more lines on input
9. Begin editing the input
10. Check for angle brackets, < >
    if present, expand the brackets and return to step 9, else
11. Check for parentheses, ( ) if present, expand the parentheses and return to step 9, else
12. Check for macro, if present, expand the macro and return to step 9, else
13. Look up the command (.LKUP) if command is unknown add macro brackets and return to step 9, else if the abbreviation is not unique, issue an error message, (.EO) and return to step 5 else
14. Build the switch and/or argument descriptors
15. Verify that argument existence is consistent with commands description, if not, issue an error message (.EO) and return to step 4, else
16. Dispatch to the command
17. Remove the command from the buffer and return to step 7

^C^A Interrupt Task

The function of the ^C^A interrupt task is to interrupt the task controlling the console. In the area of CLI this may be either the primary or utility task. For the primary task, a ?IDGOTO interrupt is issued by the ^C^A. ?IDGOTO is disabled for the utility task. In both cases, the stack is reset together with the list flag, command input pointer and output buffer pointer. An error message is then issued before CLI goes to get another line from @INPUT.

Utility Task

The primary function of the utility task is to handle double buffering for loads and dumps. This is accomplished through a JMP 0, MESSAGE where MESSAGE contains the address for a read or write of the other buffer.

Size

The following defines the size constraints on CLI. ZREL locations - ?ZMAX=374; this area contains processing mode flags, temporaries and indirect addresses. NREL locations - ?NMAX=1677; the unshared code area (approximately 500 words to start with) contains buffers, packets, stacks and names. The shared code area occupies the area (34000-77271).

Stack Structure

Initialization - when CLI begins execution stack requirement is 30.words in unshared NREL.

Primary - begins at ?NMAX; extended by ?MEMI calls when stack fault occurs. If the ?MEMI fails, the stack is reset to ?NMAX, the stack fault is set to abort, and a ?CHAIN is made to the same CLI.

^C^A - uses 20.words in unshared NREL.

Utility - uses 20.words in unshared NREL.

Environment

This section is a detailed breakdown of AOS/VS's operating environment. The environment consists of databases and levels. The data base is made up of buffers and flags. The following describes the buffers and their use in the database.

IBUF - the input buffer
size: 128 bytes (with screenedit off)
size: 76 bytes (with screenedit on)
BUF - the output buffer
size: 128 bytes
STRING - current buffer
size: 128 bytes

Also, there are a series of flags used to control CLI I/O and maintain CLI status. The most important ones are:

BCMODE	-1 for batch on 0 for console on 1 for son of exec
DINI	-1 for @DATA or byte pointer to data filename
FRAME	frame pointer at start of command execution
GOCHNS	?GOPENED channel bit map
LINI	-1 for @LIST or byte pointer to list file name
LSTIT	0 for all or address of list output packet non-zero if command changes directories
OCHNS	?OPENED channel bit map
SCGBL	total number of system calls made
SCLCL	number of system calls since last performance command
SCRMOD	screenedit mode 1: off -1: on
SPPMOD	superprocess mode 1: off -1: on
SUPMOD	superuser mode 1: off -1: on
EMODE	class 1 error mode
QMODE	squeeze mode
WMODE	class 2 error mode

Levels in CLI's environment are maintained and controlled by copies of the prompt list, string, datafile name, listfile name, search list, working directory, home pointer previous top of stack and "new top" of stack and a set of processing mode flags. The buffers maintained for level control are:

DIRBUF	contains the working directory at start of previous commands processing size: 128 bytes
PRMLST	the prompt list buffer size: 9 words
LNAM	list file name size: 128 bytes
DNAM	datafile name size: 128 bytes

The flags used for level control are:

BLV1 - level at last input
LEVEL - environment
TLVL - level last typed out

Command Processing

Sequence of Operations

The CLI begins by reading a line from @INPUT and pushing it onto the stack. Each character is examined for angle brackets, "< >", parentheses, "( )", square brackets, "[ ]", and the null at the end of the command.

The expansion of angle brackets will cause the previous "word" to be repeated for each argument in the brackets while parentheses expansion will cause the entire command to be repeated for each argument. The use of square brackets will cause the macro to be expanded into the command space on the stack.

The command string is then edited. That is, the initial spaces are removed and multiple spaces and/or tabs are compacted into single commas. This puts the CLI command into a standard form with fields delimited by commas.

The command is then isolated and looked up in the command table (CTBL). When abbreviations are used, the entire command table is searched to insure that there are no multiple matches.

Switches and arguments associated with the various commands are validated with respect to the command descriptor. As in the case of the commands, unique abbreviations for switches and arguments are accepted. An error is returned when there is a multiple switch.

The command is actually executed by being dispatched through a table via a JSR.

Commands remaining in the input buffer are shifted to the beginning of the buffer and processing continues.

Macro Processing

The macro file is first opened and read. Each character is then examined and pushed onto the stack. When a delimiter is found in the command string namely a null, new line, formfeed, or carriage return, the next character is checked for an ampersand (&). Whenever an ampersand is found neither the ampersand nor the delimiter will be pushed on the stack.

When a percent sign (%) is encountered the dummy argument is replaced with the calling argument. If the dummy argument is % n / ... all the switches on the argument in the call are searched for every / on the argument in the dummy.

When the dummy argument is %/n\... all the switches on the argument in the dummy are searched for every \ on the argument in the call.

Command Data Bases

Command Table Data

The following defines the Command Table entry format for both commands and pseudo macros.

```
entry0 : byte pointer to ASCII string for command or
         pseudo macro
entry1 : command entry point
entry2 : 0
```

This table is found in module ZDISP

Switch Descriptors

Commands with simple switches:

```
-2 : A-P switch bit map (all 16 bits)
-1 : Q-Z switch bit map (bits 0-9)
     and argument flags (bits 10-15)
 0 : command entry point
```

Commands with complex switches:

```
switch table entry
entry 0 : byte pointer to ASCII string for switch
entry 1 : address of routine for switch without a value: 0
          if the switch takes a value
entry 2 : 0 if the switch does not take a value; address
          of routine for switch with a value
     -2 : address of switch table
     -1 : argument flags (bits 10 - 15)
```

Stack Structure for Command Calls

Commands with Simple Switches (single character)

1. Command line as input <null>
2. Command <null>
3. SW1/SW2.../SWn<null>
4. Argument 1 <null>
   .
   .
   .
   .
5. Byte pointer to argument 1
   .
   .
   .
6. Argument count
7. A-P switch bit map
8. Q-Z switch bit map

Commands with Complex Switches

Complex Switches (multi-character)

1. Command line as input <null>
2. Command <null>
3. Switch 1 <null>
4. Argument 1 <null>
   .
   .
   .
   Argument n<null>

5. Address of switch 1 routine
6. Byte pointer to switch 1 value or 0)
7. Switch count
8. Byte pointer to argument 1
9. Argument count
10. 0
11. 0

Examples

The CLI PUSH Command

Module Name: ZCMD4
Entry Point: XPSH

First, ^C^A interrupts are disabled (.PIG). Seventeen (17) words of stack are then allocated and then filled with a word pointer to each one of the environment parameters. The contents of the parameters are then pushed on top of the pointers in the following order after which ^C^A is enabled:

1. The squeeze mode status
2. The CLASS2 mistake reaction
3. The CLASS1 mistake section
4. The prompt list
5. The string variable
6. The data file status and name
7. The list file status and name
8. The searchlist status and name
9. The working directory
10. The screenedit state
11. The superprocess state
12. The frame pointer
13. The superuser state
14. The beginning of the pointer block
15. The next available address after the parameters

The CLI POP Command

Module Name: XCMD4
Entry Point: XPOP

The ^C^A interrupts are disabled, the inverse of PUSH is performed and ^C^A enabled.

Template Expansion

Module Name: ZTEX

Several commands, namely: DELETE. FILESTATUS, TYPE, LOAD, DUMP, and MOVE use the template expander. The expander is a collection of subroutines that breakdown the input into modes, that is the pieces separated by colons and then search the directories encountered for files that match the template.

CLI Module Names

CLI module names and a description of their contents.

Main modules	Contents
Z	contains the primary task, ^C^A task and utility task
ZAWE	processes: warning and error messages
ZDISP	contains the command dispatch table
ZCMD0	bridges: BREAKFILE, BYE, CPUID, FILESTATUS, HELP, PROMPT, MESSAGE, PERFORMANCE, DATE. SYSID, SYSINFO, SYSLOG, TIME processes: COMMENT, WRITE
ZCMD1	bridges: BLOCK, HOST, PRIORITY, PRTYPE, RUNTIME, SCREENEDIT, SQUEEZE, SUPERPROCESS, SUPERUSER, TERMINATE, TREE, UNBLOCK
ZCMD2	processes: CHAIN, CHECKTERMS, DEBUG, EXECUTE, PROCESS, XEQ
ZCMD3	bridges: DATAFILE, LISTFILE, STRING processes: CHARACTERISTICS
ZCMD4	bridges: CLASS1, CLASS2, LOGFILE, PREVIOUS. TRACE processes: CURRENT, LEVEL, POP, PUSH

ZENV	bridges: ACL, DEFACL, DIRECTORY, SEARCHLIST processes: !ACL, !DEFACL, !DIRECTORY, !PATHNAME, !SEARCHLIST, !USERNAME
ZFMC0	bridges: CREAT, DELETE, RENAME, TYPE processes: COPY
ZFMC1	bridges: LOGEVENT, REVISION, SPACE
ZFMC2	bridges: ENQUEUE, PATHNAME PERMANENCE
ZFTA	connects/disconnect to FTA, file transfer request to FTA, process /BACKUP for FTA
ZIO	process INPUT/OUTPUT for CLI
ZLOCK	LOCK Flag
ZMAC	Macro expander
ZMATH	contains set-up for all arithmetic routines, pseudo macro !OCTAL, !SIZE, !DECIMAL
ZMOLD	processes: DUMP, LOAD, MOVE
ZOP	bridges: ASSIGN, BIAS, CONNECT, DEASSIGN, DISSCONNECT, INITIALIZE, PAUSE, REWIND processes: CONTROL, RELEASE, SEND
ZOPER	processes: OPERATOR ON/OFF
ZOV0	processes: CHARACTERISTICS, PREVIOUS
ZOV1	processes: DATA, HELP, LIST, MESSAGE, PERFORMANCE
ZOV2	processes: BLOCK, CLASS1, CLASS2, HOST, INITIALIZE PAUSE, PRIORITY, PRTYPE, RUNTIME, SCREENEDIT SQUEEZE, SUPERPROCESS, SUPERUSER, STRING, SYSID, SYSLOG, TREE, UNBLOCK, WHO
ZOV3	processes: ACL, DATE, DEFACL, DIRECTORY, ENQUEUE, LOGEVENT, PATHNAME, PERMANENCE, PROMPT, SEARCHLIST, SPACE, TIME
ZOV4	processes: CONNECT, CREATE, DELETE, DISCONNECT, INITIALIZATION, LOGFILE, PREFIX, REVISION, REWIND, TYPE
ZOV5	processes: QCANCEL, QDISPLAY, QHOLD, QUNHOLD
ZOV6	processes: QBATCH, QFTA, QPLOT, QPRINT, QPUNCH

	QSNA, QSUBMIT
ZOV7	processes: ASSIGN, BIAS, BYE, COPY, DEASSIGN, DISMOUNT, MOUNT, TRACE
ZOV8	processes: BREAKFILE, CHAIN, DEBUG, EXECUTE, PROCESS, VARIABLE, XEQ
ZOV9	processes: CPUID, TERMINATION MESSAGES
ZOV10	processes: FILESTATUS. RENAME, SYSINFO
ZOV11	processes: OPERATOR ON/OFF, OPERATOR COPY
ZPARS	contains the parameter file for CLI
ZPSM	processes: !EQUAL, !NEQUAL, !END, !ELSE, !READ, !PID, !HID, !STRING, !LOGON, !OPERATOR, !DATE. !TIME, !ASCII, !EXPLODE, !ENAME, !SYSTEM, !LISTFILE, !DATAFILE, !CONSOLE, !EFILENAME, !EPREFIX, !EDIRECTORY, !EEXTENSION
ZQUE	bridges: QDISPLAY, QBATCH, QSUBMIT, QPRINT, QPUNCH, QPLOT, QHOLD, QFTA, QSNA, QCANCEL, QUNHOLD, MOUNT, DISMOUNT
ZSTK	handles stack manipulation for CLI
ZSUB	contains common subroutines used by CLI modules
ZTEX	performs template expansion for CLI
ZURTRES	runtime interface resource manager

CLI Commands and their modules

CLI commands and their corresponding module names and module entry points. XXX -> YYY indicates that the entry point is actually a bridge to an overlay entry point. [overlay entry point] = [module Entry Point.]
For example, the overlay entry point for ACL is XACL<dot>.

Command Name	Module Name	Module Entry Point
ACL	ZENV -> ZOV3	XACL
ASSIGN	ZOP -> ZOV7	XASS
BIAS	ZOP -> ZOV7	XBIA
BLOCK	ZCMD1 -> ZOV2	XBLK
BREAKFILE	ZCMD1 -> ZOV8	XBRK
BYE	ZCMD0 -> ZOV7	XBYE
CHAIN	ZCMD2 -> ZOV8	XCHN
CHARACTERISTICS	ZCMD3 -> ZOV0	XCHA
CHECKTERMS	ZCMD2	XLSN
CLASS1	ZCMD4 -> ZOV2	XCL1
CLASS2	ZCMD4 -> ZOV2	XCL2
COMMENT	ZCMD0	XCOM
CONNECT	ZOP -> ZOV4	XCNX
CONTROL	ZOP	XCON
COPY	ZFMC0 -> ZOV7	XCOP
CPUID	ZCMD0 -> ZOV9	XCPU
CREATE	ZFMC0 -> ZOV4	XCRE
CURRENT	ZCMD4	XCUR
DATAFILE	ZCMD3 -> ZOV1	XDTA
DATE	ZCMD0 -> ZOV3	XDAT
DEASSIGN	ZOP -> ZOV7	XDEA
DEBUG	ZCMD2 -> ZOV8	XDEB
DEFACL	ZENV -> ZOV3	XDEF
DELETE	ZFMC0 -> ZOV4	XDEL
DIRECTORY	ZENV -> ZOV3	XDIR
DISMOUNT	ZOP -> ZOV4	XDCN
DISMOUNT	ZQUE -> ZOV7	XDIS
DUMP	ZMOLD	XDUM
ENQUEUE	ZFMC2 -> ZOV3	XENQ
EXECUTE	ZCMD2 -> ZOV8	XEXE
FILESTATUS	ZCMD0 -> ZOV10	XFIL
HELP	ZCMD0 -> ZOV1	XHLP
HOST	ZCMD1 -> ZOV2	XHST
INITIALIZE	ZOP -> ZOV4	XINI
LEVEL	ZCMD4	XLVL
LISTFILE	ZCMD3 -> ZOV1	XLIS
LOAD	ZMOLD	XLOA
LOGEVENT	ZFMC1 -> ZOV3	XLEV
LOGFILE	ZCMD4 -> ZOV4	XLOG
MESSAGE	ZCMD0 -> ZOV1	XMES

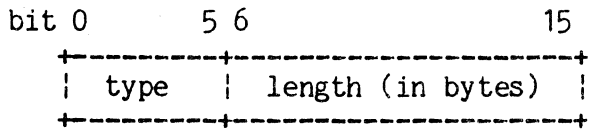
MOUNT	ZQUE -> ZOV7	XMOU
MOVE	ZMOLD	XMOV
OPERATOR	ZOPER -> ZOV11	XOPE
PATHNAME	ZFMC2 -> ZOV3	XPAT
PAUSE	ZOP -> ZOV2	XPAU
PERFORMANCE	ZCMD0 -> ZOV1	XPER
PERMANENCE	ZFMC2 -> ZOV3	XPMN
POP	ZCMD4	XPOP
PREFIX	ZCMD0 -> ZOV4	XPFX
PREVIOUS	ZCMD4 -> ZOV0	XPRE
PRIORITY	ZCMD1 -> ZOV2	XPPR
PROCESS	ZCMD2 -> ZOV8	XPRO
PROMPT	ZCMD0 -> ZOV3	XPRM
PRTYPE	ZCMD1 -> ZOV2	XPTY
PUSH	ZCMD4	XPSH
QBATCH	ZQUE -> ZOV6	XQBA
QCANCEL	ZQUE -> ZOV5	XQCA
QDISPLAY	ZQUE -> ZOV5	XQDI
QFTA	ZQUE -> ZOV6	XQFT
QHOLD	ZQUE -> ZOV5	XQHO
QPLOT	ZQUE -> ZOV6	XQPL
QPRINT	ZQUE -> ZOV6	XQPR
QSNA	ZQUE -> ZOV6	XQSN
QPUNCH	ZQUE -> ZOV6	XQPU
QSUBMIT	ZQUE -> ZOV6	XQSU
QUNHOLD	ZQUE -> ZOV5	XQUN
RELEASE	ZOP	XREL
RENAME	ZFMC0 -> ZOV10	XREN
REVISION	ZFMC1 -> ZOV4	XREV
REWIND	ZOP -> ZOV4	XREW
RUNTIME	ZCMD1 -> ZOV2	XRNT
SCREENEDIT	ZCMD1 -> ZOV2	XSCR
SEARCHLIST	ZENV -> ZOV3	XSEA
SEND	ZOP	XSEN
SPACE	ZFMC1 -> ZOV3	XSPA
SQUEEZE	ZCMD1 -> ZOV2	XSQU
STRING	ZCMD3 -> ZOV2	XSTR
SUPERPROCESS	ZCMD1 -> ZOV2	XSPP
SUPERUSER	ZCMD1 -> ZOV2	XSUP
SYSID	ZCMD0 -> ZOV2	XSID
SYSINFO	ZCMD0 -> ZOV10	XSIN
SYSLOG	ZCMD0 -> ZOV2	XSYS
TERMINATE	ZCMD1 -> ZOV3	XTER
TIME	ZCMD0 -> ZOV3	XTIM
TRACE	ZCMD4 -> ZOV7	XTRA
TREE	ZCMD1 -> ZOV2	XTRE
TYPE	ZFMC0 -> ZOV4	XTYP
UNBLOCK	ZCMD1 -> ZOV2	XUNB
WHO	ZCMD1 -> ZOV2	XWHO
WRITE	ZCMD0	XWRI
XEQ	ZCMD2 -> ZOV8	XRUN

CLI Pseudo macros and their corresponding module names and module entry points.

Pseudo Macro Name	Module Name	Module Entry Point
!ACL	ZENV	YACL
!ASCII	ZPSM	YASC
!CONSOLE	ZPSM	YCNS
!DATAFILE	ZPSM	YDTA
!DATE	ZPSM	YDAT
!DECIMAL	ZMATH	YDEC
!DEFACL	ZENV	YDEF
!DIRECTORY	SENV	YDIR
!EDIRECTORY	ZPSM	YEDR
!EEXTENSION	ZPSM	YEXT
!EFILENAME	ZPSM	YEFI
!ELSE	ZPSM	YELS
!ENAME	ZPSM	YENA
!END	ZPSM	YEND
!EPREFIX	ZPSM	YEPR
!EQUAL	ZPSM	YEQU
!EXPLODE	ZPSM	YEXP
!FILENAMES	ZTEX	YFIL
!HID	ZPSM	YHID
!HOST	ZPSM	YHST
!LEVEL	ZPSM	YLVL
!LISTFILE	ZPSM	YLIS
!LOGON	ZPSM	YLOG
!NEQUAL	ZPSM	YNEQ
!OCTAL	ZMATH	YOCT
!OPERATOR	ZPSM	YOPR
!PATHNAME	ZENV	YPAT
!PID	ZPSM	YPID
!READ	ZPSM	YREA
!SEARCHLIST	ZENV	YSEA
!SIZE	ZMATH	YSIZ
!STRING	ZPSM	YSTR
!SYSTEM	ZPSM	YSYS
!TIME	ZPSM	YTIM
!UADD	ZMATH	YADD
!UDIVIDE	ZMATH	YDIV
!UEQ	ZMATH	YUEQ
!UGE	ZMATH	YUGE
!UGT	ZMATH	YGT
!ULE	ZMATH	YULE
!ULT	ZMATH	YULT
!UMODULO	ZMATH	YMOD
!UMULTIPLY	ZMATH	YMUL
!UNE	ZMATH	YUNE
!USERNAME	ZENV	YUSE
!USUBTRACT	ZMATH	YSUB
!VARO - 9	ZMATH	YVRO - 9

AOS/VS Dump Format

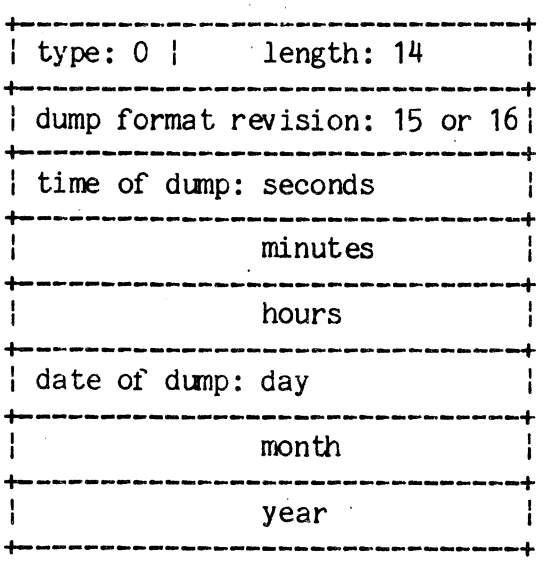
A dump file created by AOS/VS CLI consists of variable length records, each having a fixed format header containing the block's type and length. The format of the header is:



These blocks are not word-aligned, since the byte length of any of the blocks might be odd. The length refers to the length without the header. Specific block types are detailed below. There is one anomaly to the blocking scheme -- data from data files are not contained in some type of block, but rather follow data header blocks (block type 7 -- see below).

0 -- Start of dump

format:

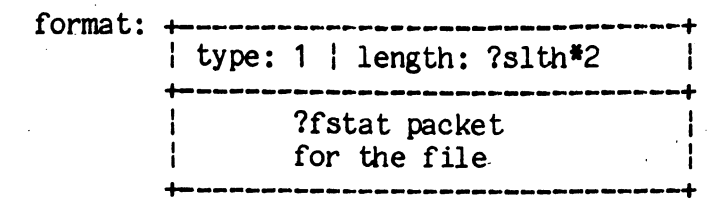


(all numbers are base 10)

may occur only once, at the start of the dump file

1 -- file status block
==========

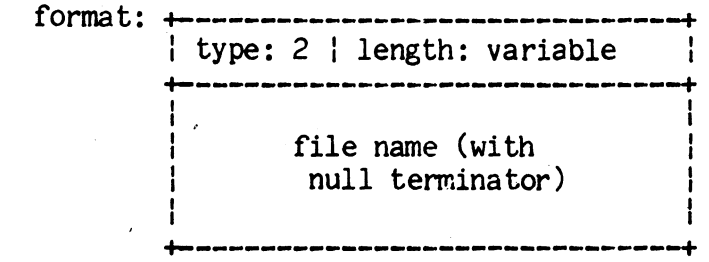
format:



This is the first you'll see of a file. It contains useful information like the file type and length. It is always followed by a name block.

2 -- Name block
==========

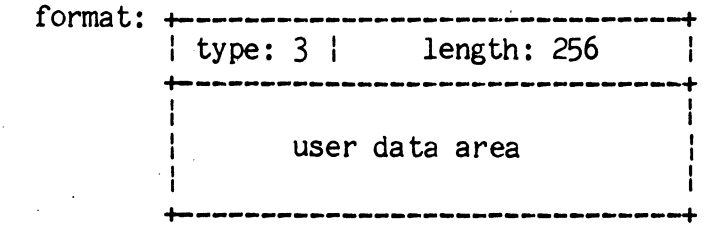
format:



the file name is a simple name (not a pathname). You have to look at the directory start and end blocks to tell what the pathname is.

3 -- Uda block
==========

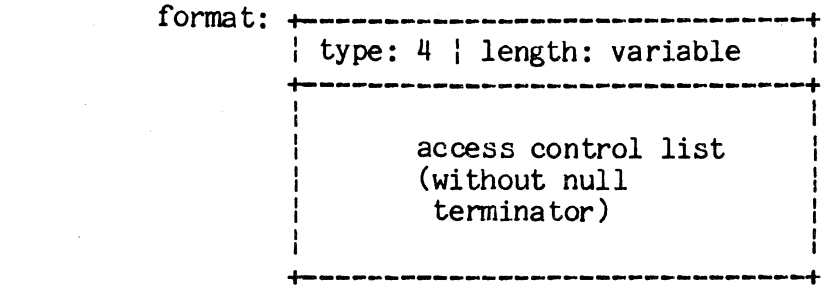
format:



only directories and data files may have uda blocks, and in both cases, it directly follows the end block. Most files do not have a uda -- currently they are used only by infos.

4 -- Acl block
==========

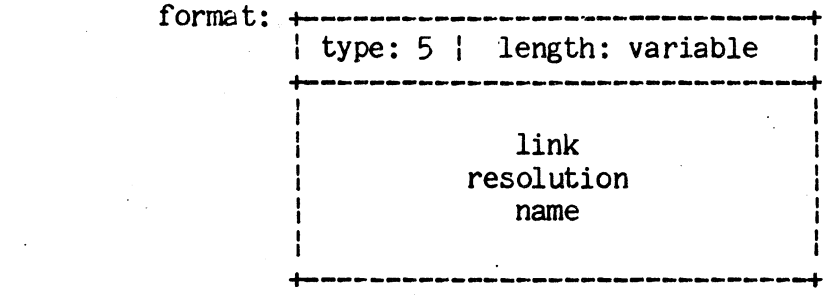
format:



the acl block is used only for directories and data files, and in both cases, it comes right before the end block. The acl block is not always present. and if it is not there. the acl should be set to the user's default.

5 -- Link block
==========

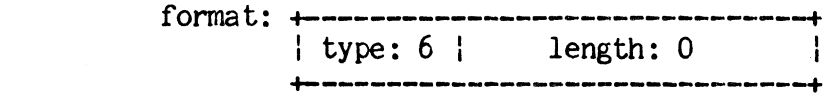
format:



the link block must be there for links, and it follows the name block.

6 -- Start block
==========

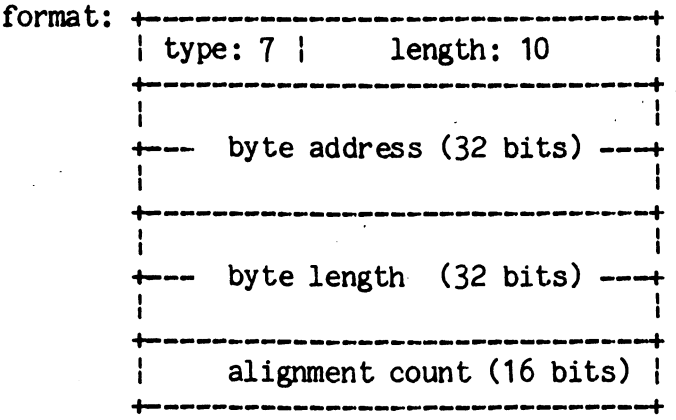
format:



the start block is always present for data files and directories, and is always matched by an end block. It follows the file name block, and after it come data blocks (for data files) or file status blocks of subordinate files (for directories).

7 -- Data header block
==========

format:

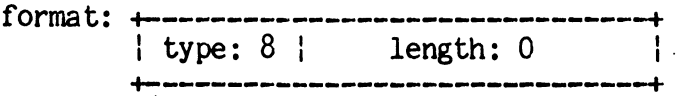


the byte address specifies where in the data file to put the following chunk of data. Currently, it must be a multiple of 512 bytes (one disk block). Minus one as the address means continue from where the last block left off (this convention is not now used). In general, hunks may be left out of the data (if one data block does not take up where the previous one left off) -- this happens if all the intervening area was full of zeros.

A data header block will be followed by <alignment count> bytes to be ignored (currently either zero or one), followed by <byte length> bytes of data. Next comes another data block, or an acl block, or an end block.

8 -- End block
==========

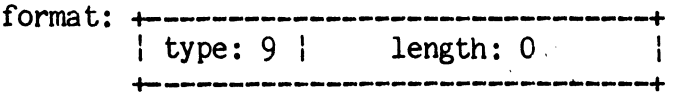
format:



an end block means the file can be closed, or the directory popped out of.

9 -- End of dump
==========

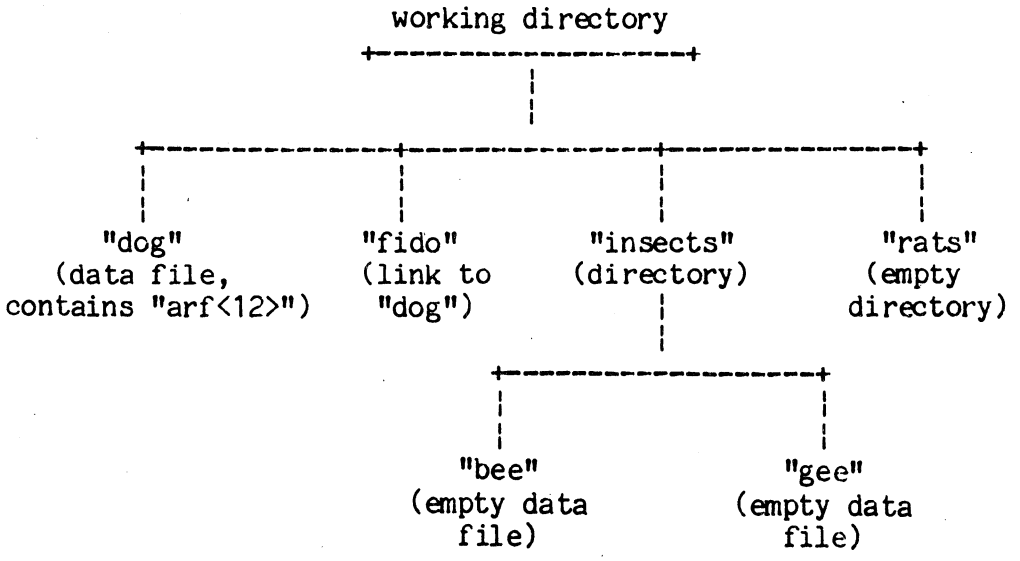
format:



occurs only once, at the end of the dump file.

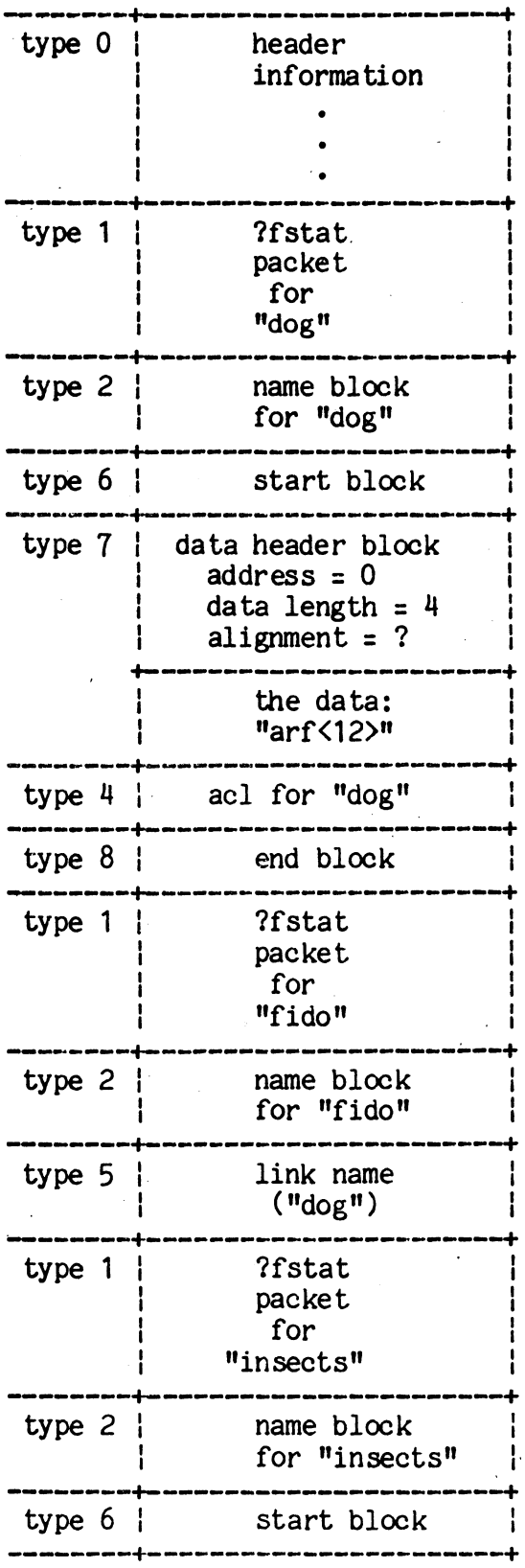
Picture of a sample dump file

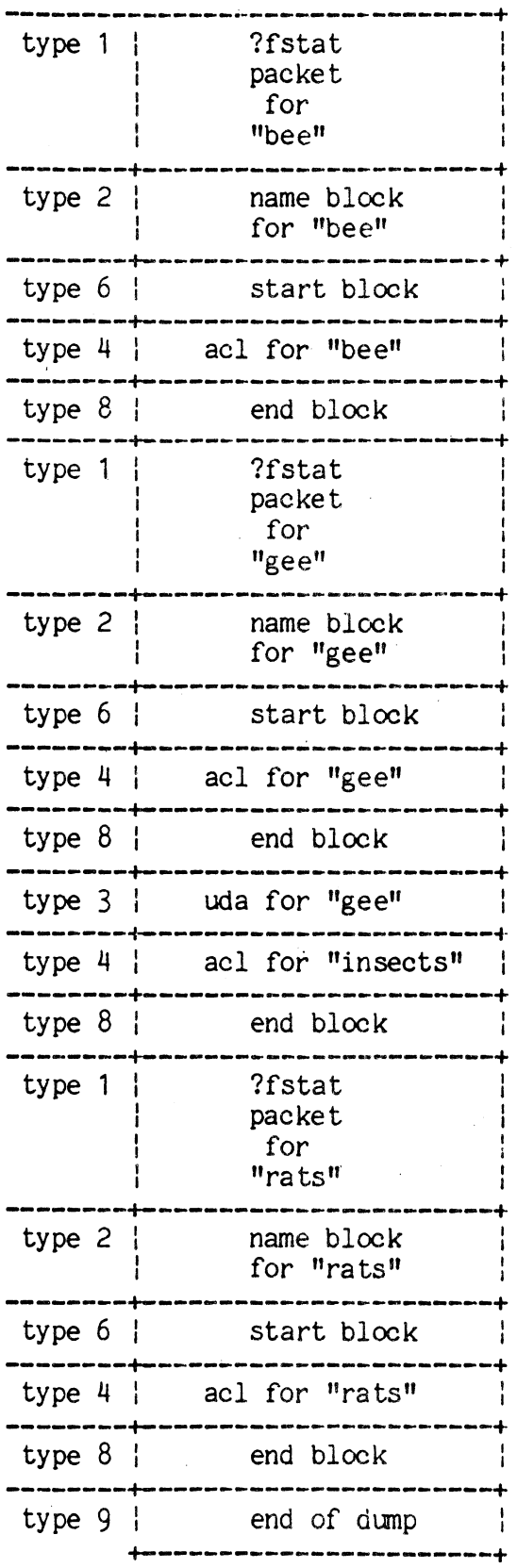
Suppose we have the following directory tree:



suppose also that the file "gee" has a user data area (uda).

In the following diagram, the lines are not necessarily on word boundaries. Remember, the information 'blocks' are byte aligned, not word aligned.

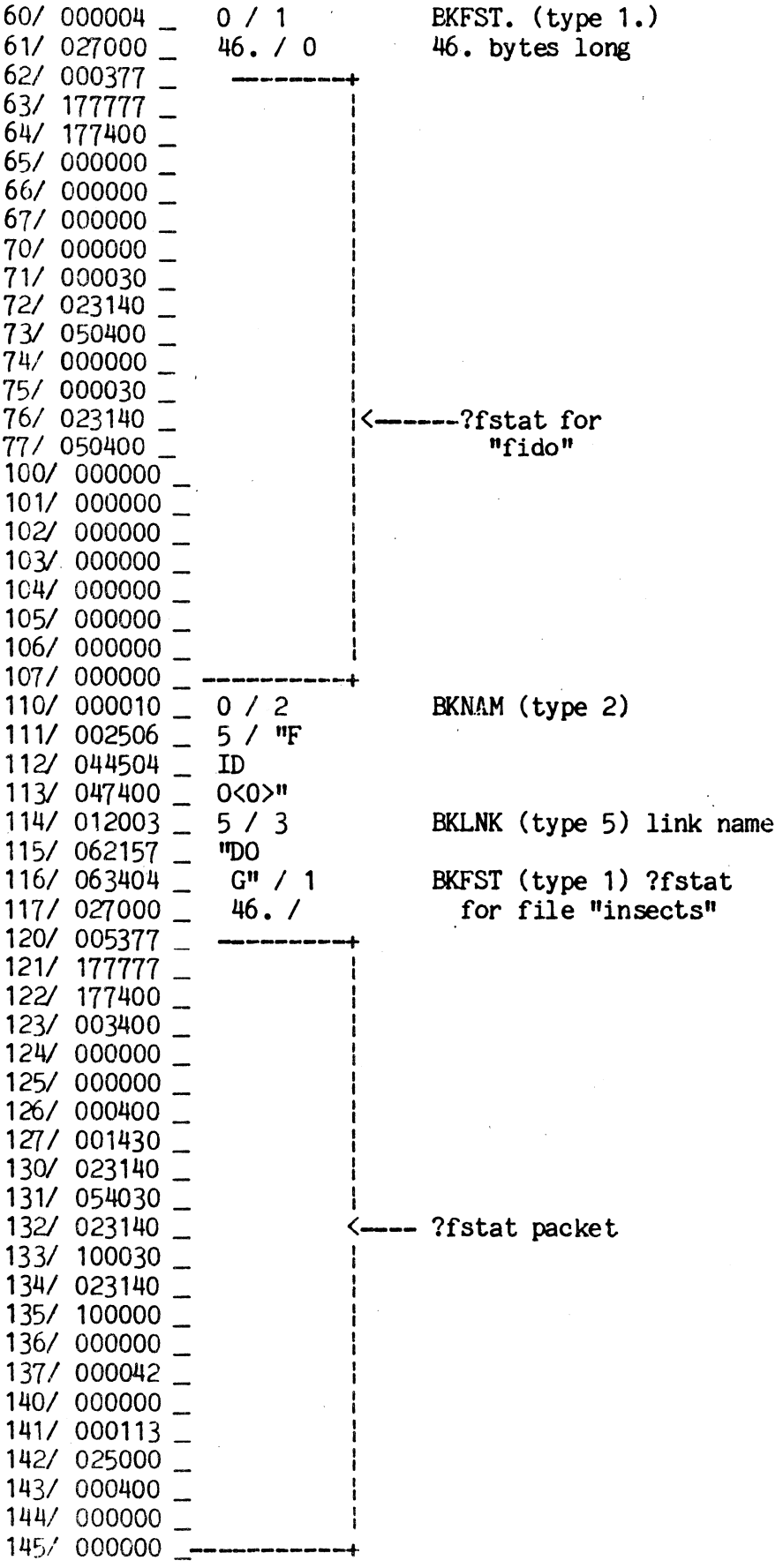



type 1
?fstat packet for "bee"
type 2
name block for "bee"
type 6
start block
type 4
acl for "bee"
type 8
end block
type 1
?fstat packet for "gee"
type 2
name block for "gee"
type 6
start block
type 4
acl for "gee"
type 8
end block
type 3
uda for "gee"
type 4
acl for "insects"
type 8
end block
type 1
?fstat packet for "rats"
type 2
name block for "rats"
type 6
start block
type 4
acl for "rats"
type 8
end block
type 9
end of dump

Sample FED of a dump file

The following is a real live sample of a CLI dump tape. The structure that was dumped is the same as described above.

```
0/ 000016 _    0 / 14.          BKHDR (type 0)
1/ 000017 _    15.          Dump revision number
2/ 000062 _    50.          Second of creation
3/ 000055 _    45.          Minute of creation
4/ 000015 _    13.          Hour of creation
5/ 000003 _    3.           Day of creation
6/ 000014 _    12.          Month of creation
7/ 000124 _    84.          Year of creation
10/ 002056 _   1 / 46.           BKFST (type 1)/ 46. bytes
11/ 000104 _    -------+
12/ 177777 _           |
13/ 177777 _           |
14/ 000000 _           |
15/ 000000 _           |
16/ 000000 _           |
17/ 000004 _           |
20/ 000003 _           |
21/ 014046 _           |
22/ 060107 _           |<-------?FSTAT Packet
23/ 014046 _           |          for "DOG"
24/ 060115 _           |
25/ 014046 _           |
26/ 060115 _           |
27/ 000000 _           |
30/ 000000 _           |
31/ 000004 _           |
32/ 000000 _           |
33/ 046734 _           |
34/ 000000 _           |
35/ 000000 _           |
36/ 000000 _           |
37/ 000000 _ ----------+
40/ 004004 _   2 / 4            BKNAM (type 2) / 4 bytes.
41/ 042117 _    "DO
42/ 043400 _     G<0>"
43/ 014000 _   6.               BKBEG (type 6) start block
44/ 016012 _   7 / 5            BKDAT (type 7) data header block
45/ 000000 _    ---|
46/ 000000 _    ---|            byte address of data
47/ 000000 _    ---+
50/ 000004 _    ---+            byte length  4 bytes
51/ 000000 _                    alignment count
52/ 060562 _   "ar
53/ 063012 _    f<012>"
54/ 010005 _   4 / 5            BKACL (type 4) ACL
55/ 041501 _   "CA
56/ 046000 _    L<0>"
57/ 017440 _   <37> / 8. 37 = OWARE / BKEND (type 8.)
```


60/ 000004 _ 0 / 1 BKFST. (type 1.)
61/ 027000 _ 46. / 0 46. bytes long
62/ 000377 _
63/ 177777 _
64/ 177400 _
65/ 000000 _
66/ 000000 _
67/ 000000 _
70/ 000000 _
71/ 000030 _
72/ 023140 _
73/ 050400 _
74/ 000000 _
75/ 000030 _
76/ 023140 _ <------?fstat for
77/ 050400 _ "fido"
100/ 000000 _
101/ 000000 _
102/ 000000 _
103/ 000000 _
104/ 000000 _
105/ 000000 _
106/ 000000 _
107/ 000000 _
110/ 000010 _ 0 / 2 BKNAM (type 2)
111/ 002506 _ 5 / "F
112/ 044504 _ ID
113/ 047400 _ 0<0>"
114/ 012003 _ 5 / 3 BKLNK (type 5) link name
115/ 062157 _ "DO
116/ 063404 _ G" / 1 BKFST (type 1) ?fstat
117/ 027000 _ 46. / for file "insects"
120/ 005377 _
121/ 177777 _
122/ 177400 _
123/ 003400 _
124/ 000000 _
125/ 000000 _
126/ 000400 _
127/ 001430 _
130/ 023140 _
131/ 054030 _
132/ 023140 _ <---- ?fstat packet
133/ 100030 _
134/ 023140 _
135/ 100000 _
136/ 000000 _
137/ 000042 _
140/ 000000 _
141/ 000113 _
142/ 025000 _
143/ 000400 _
144/ 000000 _
145/ 000000 _

```
146/ 000010 _ 0 / 1           BKNAM (type 2)
147/ 004111 _ 10. / "I
150/ 047123 _ NS
151/ 042503 _ EC
152/ 052123 _ TS
153/ 000030 _ <0>" / 6        BKBEG (type 6)
154/ 000004 _   0 / 1         BKFST (type 1)
155/ 027000 _ 46. / 0
156/ 042377 _----------+
157/ 177777 _          |
160/ 177400 _          |
161/ 000000 _          |
162/ 000000 _          |
163/ 000000 _          |
164/ 002000 _          |
165/ 001430 _          |
166/ 023140 _          < -------?fstat packet
167/ 060030 _          |          for "BEE"
170/ 023140 _          |
171/ 060030 _          |
172/ 023140 _          |
173/ 060000 _          |
174/ 000000 _          |
175/ 000000 _          |
176/ 000000 _          |
177/ 000000 _          |
200/ 000000 _          |
201/ 000000 _          |
202/ 000000 _          |
203/ 000000 _----------+
204/ 000010 _ 0 / 2           BKNAM (type 2)
205/ 002102 _ 4 / "B
206/ 042505 _  EE
207/ 000030 _  <0>" / 6       BKBEG (type 6)
210/ 000020 _  0 / 4          BKACL (type 4)
211/ 002503 _  5 / "C
212/ 040514 _  AL
213/ 000037 _  <0>" / <37>   O W A R E access
214/ 020000 _   8. / 0        BKEND (type 8)
215/ 002056 _  1 / 46.        BKFST (type 1)
216/ 000104 _----------+
217/ 177777 _          |
220/ 177777 _          |
221/ 000000 _          |
222/ 000000 _          |
223/ 000000 _          |
224/ 000004 _          |
225/ 000003 _          |
226/ 014046 _          |
227/ 060141 _          <-------- ?fstat packet
230/ 014046 _          |          for "GEE"
231/ 060141 _          |
232/ 014046 _          |
233/ 060141 _          |
```

```
234/ 000400 _          |
235/ 000000 _          |
236/ 000000 _          |
237/ 000000 _          |
240/ 000000 _          |
241/ 000000 _          |
242/ 000000 _          |
243/ 000000 _          |
244/ 000000 _----------+
245/ 004004 _ 2 / 4            BKNAM (type 2)
246/ 043505 _ "GE
247/ 042400 _  E<0>"
250/ 014000 _ 6 / 0            BKBEG (type 6)
251/ 010005 _ 4 / 5            BKACL (type 4)
252/ 041501 _ "CA
253/ 046000 _  L<0>"
254/ 017440 _  <37> / 8.       OWARE access / BKEND (type 8)
255/ 000015 _  0 / 3           BKUDA (type 3)
256/ 000106 _  256             length of UDA
257/ 041400 _---------+
260/ 000000 _         |
261/ 050000 _         |
262/ 042000 _         |
263/ 002000 _         |
264/ 042001 _         |
265/ 000401 _         |
266/ 000401 _         |
267/ 000401 _         |
270/ 000401 _         |
271/ 000000 _         |
272/ 000000 _         |
273/ 000000 _         |
274/ 000000 _         |
275/ 000020 _         |
276/ 000000 _         |
277/ 000000 _         |
300/ 000000 _         |
301/ 000000 _         |
302/ 000000 _         |
303/ 000000 _         |
304/ 000000 _         |
305/ 000000 _         |
306/ 000000 _         |
307/ 000000 _         |
310/ 000000 _         |
311/ 000000 _         |
312/ 000000 _         |
313/ 000000 _         |
314/ 000000 _         |
315/ 000000 _         |
316/ 000000 _         |
317/ 000000 _         |
320/ 000000 _         |
321/ 000000 _         |
```

```
322/ 000000 _
323/ 000000 _
324/ 000000 _
325/ 000000 _
326/ 000000 _
327/ 000000 _
330/ 000000 _
331/ 000000 _
332/ 000000 _
333/ 000000 _
334/ 000000 _
335/ 000000 _
336/ 000000 _
337/ 000000 _
340/ 000000 _
341/ 000000 _
342/ 000000 _
343/ 000000 _
344/ 000000 _
345/ 000000 _
346/ 000000 _
347/ 000000 _
350/ 000000 _
351/ 000000 _
352/ 000000 _
353/ 000000 _
354/ 000000 _
355/ 000000 _      UDA for file "GEE"
356/ 000000 _
357/ 000000 _
360/ 000000 _
361/ 000000 _
362/ 000000 _
363/ 000000 _
364/ 000000 _
365/ 000000 _
366/ 000000 _
367/ 000000 _
370/ 000000 _
371/ 000000 _
372/ 000000 _
373/ 000000 _
374/ 000000 _
375/ 000000 _
376/ 000000 _
377/ 000000 _
400/ 000000 _
401/ 000000 _
402/ 000000 _
403/ 000000 _
404/ 000000 _
405/ 000000 _
406/ 000000 _
407/ 000000 _
```

```
410/ 000000 _          |
411/ 000000 _          |
412/ 000000 _          |
413/ 000000 _          |
414/ 000000 _          |
415/ 000000 _          |
416/ 000000 _          |
417/ 000000 _          |
420/ 000000 _          |
421/ 000000 _          |
422/ 000000 _          |
423/ 000000 _          |
424/ 000000 _          |
425/ 000000 _          |
426/ 000000 _          |
427/ 000000 _          |
430/ 000000 _          |
431/ 000000 _          |
432/ 000000 _          |
433/ 000000 _          |
434/ 000000 _          |
435/ 000000 _          |
436/ 000000 _          |
437/ 000000 _          |
440/ 000000 _          |
441/ 000000 _          |
442/ 000000 _          |
443/ 000000 _          |
444/ 000020 _          |
445/ 000000 _          |
446/ 000000 _          |
447/ 000000 _          |
450/ 000000 _          |
451/ 000000 _          |
452/ 000000 _          |
453/ 000000 _          |
454/ 000000 _          |
455/ 000000 _----------+
456/ 000020 _ 0 / 4              BKACL (type 4)
457/ 002503 _ 4 / "C
460/ 040514 _ AL
461/ 000037 _ <0>" / <37>        O W A R E
462/ 020000 _  8. / 0            BKEND (type 8)
463/ 002056 _  1 / 46.           BKFST (type 1)
464/ 000012 _----------+
465/ 177777 _          |
466/ 177777 _          |
467/ 000007 _          |
470/ 000000 _          |
471/ 000000 _          |
472/ 000001 _          |
473/ 000003 _          |
474/ 014046 _          |
475/ 060132 _          |
```

```
476/ 014046 _          |
477/ 060132 _          |
500/ 014046 _          |
501/ 060132 _          |
502/ 000000 _          |
503/ 000000 _          |
504/ 000000 _          |
505/ 000000 _          |
506/ 000000 _          |
507/ 000000 _          |
510/ 000000 _          |
511/ 000000 _          |
512/ 000000 _----------+
513/ 004005 _ 2 / 5            BKNAM (type 2)
514/ 051101 _ "RA
515/ 052123 _  TS
516/ 000030 _ <0>" / 6         BKBEG (type 6)
517/ 000020 _  0 / 4           BKACL (type 4)
520/ 002503 _  4 / "C
521/ 040514 _  AL
522/ 000037 _  <0>" / <37>     OWARE
523/ 020000 _  8. / 0          BKEND (type 8)
524/ 022000 _  9. / 0          BKEDP (type 9)
```

Thus ends the sample dump file.

CHAPTER 10 -- INITIALIZATION
(AOS/VS revision 5.00)

This chapter will discuss AOS/VS initialization, the various bootstraps available and the stand alone utlities needed to build the AOS/VS environment.

There are several programs which get the user started in the world of AOS/VS. These programs perform several unique functions. Before the user can get started in the AOS/VS environment he must run two very important programs. They are DFMTR and INSTL. These are stand alone or stand among programs. The programs are the same in operation whether they be stand alone or stand among. The DFMTR program creates the AOS/VS disk environment and INSTL creates the system disk environment. In addition to these programs there are three other programs which play important roles in the world of AOS/VS. They are TBOOT, DKBT, and SYSBOOT. Each of these programs allow for the use of a device as a media which is used to load programs to be run in memory. TBOOT reads programs from magtape. DKBT is the program which is installed on the system disk in blocks 0 and 1. SYSBOOT is the program which gets the full AOS/VS system functioning.

When the operator types BOOT 22 there is read from ROM in the computer a small program which loads in the first 1024 words from file 0 and does a JMP . at location 377. This JMP . gets overwritten by the data read from tape which then starts the real TBOOT program.

TBOOT

The first program a user or system engineer uses is TBOOT. TBOOT allows the user to load and execute a stand-alone program from magnetic tape.

- Go to location 0 and start program
- Force a JMP @376 into location 777
- When 777 is overwritten it will JMP to location 10
- At 10 IORST is done
- Get Device code of tape drive and adjust I/O driver for that device.
- Display message for file number
- Make sure tape is backspaced to file 0.
- Space forward the number of files entered at prompt
- Move TBOOT to high memory out of the way.
- Set up DCH mapping and init the map registers
- Start the load of the specified file into memory
- After loading in program rewind the tape
- Get start address from location 2 and jump to it.

DFMTR

DFMTR is the program which is used to format a disk to enable it to be used by AOS/VS. It is usually run stand-alone from the initial release tape. On the tape DFMTR is file 2.

- Reset stacks and clear error traps
- Display revision number and program name
- Ask for full or partial format
- Get description of LDU's
- If partial format read in DIB and check if REV 2 disk
  if not valid display inconsistent DIB and exit
- Print out map of LDU (i.e. size of disk)
- Get LDU unique ID.
- Get LDU name
- Build ACL for disk
- Run surface analysis to flag bad blocks
- Install DIB on each Physical Unit
- If PU is 1 write out 'funny FIB'
- Get bitmap address and init bitmap
- Make sure there are no bad blocks in bitmap area
- Write bitmap onto disk
- If system disk allocate bootstrap area (124. blocks)
- Get and set overlay area and size
- Get and set up Remap area on all disks
- Write out name block and Access Control Block
- Write out 'LOGICAL DISK CREATED'
- DFMTR is done

INSTL

INSTL is the program which installs the system bootstrap and the operating system in the ivisible area of the system disk. It is file 3 on the release tape and can be booted using TBOOT.

- o CLear stacks and set up error routines
- o Print out program name and revision #
- o Get the LDU device for drivers
- o Read in the DIB
- o If #1 in LDU set up DKBT driver info
- o Write out blocks 0 and 1 for DKBT
- o Get answer to install system bootstrap
- o If yes then get the file from somewhere (usually tape file 4)
- o Get answer to install a system
- o If yes then get the file from somewhere (usually tape file 5)
- o Rewrite DIB
- o Exit back to SCP-CLI

DKBT

DKBT is the program which gets written to blocks 0 and 1 of the system disk for the initial loading of the disk and its drivers to enable the loading of the SYSBOOT program.

- After having been loaded by the hardware ROMs DKBT starts at location 10.
- Issue an IORST and retrieve device code which is in AC0
- Set up for entry into location 400 in DKBT
- Read in DIB
- Start loading SYSBOOT into memory starting at disk address stored in the DIB
- Re-cal the disk
- Move part of SYSBOOT to 76000
- Move rest of SYSBOOT to proper place
- Enter SYSBOOT through its location 2 start address

SYSBOOT

SYSBOOT is the program which loads in the program to be run very similiar to TBOOT. It resides on disk and knows about standard disk file structure. It only knows about a hashframe size of 7 (so don't change the hashframe size of the root directory).

- Issue IORST and set up ECC error routine for memory
- Init the entire LDU by asking for each disk in the LDU
- Verify that the disks are valid devices
- Read in DIB to check for valid revision and FIXUP bit on
- Set up LDU table to reflect this disk
- Set up bad block table
- Get microcode file name and request if it should be loaded
- If it was the permanent name then don't ask just load it
- Resolve the file name on disk
- Execute the LCS instruction to load the microcode
- Size memory for SYSBOOT
- Get the system file name
- If user specified a file name resolve the path else create a pointer to the installed system
- Read in the system file and move it to high memory
- Write out the overlay file for the OS.
- Set up communications area for SINIT
- This communications area has the address of the overlay file and names of LDU's in system
- Move part of system into lower memory and start the SINIT code

At this point in time the system has been booted into memory and now some specific modules take over to process the sizing of memory and building the system databases out of free memory. The following will follow the flow through the modules involved with starting AOS/VS into its running state. The modules are SINIT, SINIT1 and MLDUI. There is also a process started as PID 3 which was built during the VSGEN procedure (CLIBT).

SINIT

- Set up the stack base, frame pointer, stack limit, and stack pointer
- Set up for default IOC 0
- Check to see if system patched, if not issue a warning
- Set up PIT slice residue for system startup
- Determine microcode rev and physical memory size
- If not valid do not come up
- Get table left behind by SYSBOOT
- If system booted from magtape for debugger then get size of OVMIN and set it up as the size of the disk overlay file
- Display AOS/VS and revison number
- Clear last page of memory for disk world
- Set up maximum number of IOC's to support (currently 2)
- Set up CBASE.W for pointing to the CME's
- Initialize the CME's and free memory block chain (FC1024)
- Get memory for the system wide shared page header hash table
- Set up system 'map'
- Set up SBR table
- Set up page tables for the first 32 mB memory
- Set hi level page tables to point to low level page tables
- Write protect the resident executeable code of system
- Execute protect page 0
- Flag overlays to prevent flushing to the page file

- Turn on the ATU
- Init IOC status register clear all error flags
- Enable data channel mapping
- Clear reference and modified bits for all physical pages
- Set up sytem overlay queue
- Allocate logical swap area
- Reserve memory for the swapfile VCME descriptor page(s)
- Allocate system DLS's (Dynamic Logical Slots)
- Set up referenced bit matrix
- If system call counting enabled (SCCNT <> 0) set up counter table
- Set up PTBL for CLIBT
- Set up PIDBT
- Init interrupt vector table
- Allocate data channel map slots
- Allocate 8. system buffers
- Turn on interrupts
- Go to SINIT1 code
- Initialize master LDU (JSR MLDUI)
- Get Date and Time from the operator
- Initialize the RTC
- Initialize interface to SCP
- Initialize the UPSC if avaiable
- INIT floating point unit
- Get answer to default specs
- Set up system buffer cache
- Set up :PER directory

- Create Generic file names
- Set up priority ranges
- Create unit entries in unit table (UNITB)
- Create :PROC and PIF
- Create :SWAP and :PAGE
- Set up stack temps for proc'ing CLIBT
- Ask if this is initial load or not
- Create an IPC spool file so that PMGR can talk to CLIBT
- If initial load make CLIBT PID 2
- Start CLIBT as PID 3
- This is where SINIT1 ends and AOS/VS is really running

CLIBT

CLIBT is the initial process which is running and set up by system initialization code. It is run as PID 3. After it is run it disappears and leaves the PMGR running as PID 1 and OP:CLI as PID 2.

- Set up a stack for self use
- Enable I/O mode to use OP console
- Set default ACL
- Check flags sent from SINIT
- If initial load GO and do it
- Set ACL for :PROC to + E
- Set ACL for :PER to + RE
- Create PMGR proc packet and wait for it to start
- Create PID 2 process
- Enable ERCC interrupts
- Kill CLIBT

This is the end of the initialization programs. As can be seen from the brief descriptions above, starting AOS/VS to run is a considerable task. These routines are not very complex and are really some pieces of VS 'hacked' out to start the initial processes.