DataGeneral

Customer Documentation

AOS/VS System Concepts

AOS/VS System Concepts

093-000335-01

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000335

Revision 01, February 1986

NOTICE

AOS/VS System Concepts
093-000335-01

Revision History:	Effective with:
Original Release - March 1983	
First Revision - February 1986	AOS/VS Rev. 7.00

A vertical bar or an asterisk in the margin of a page indicates substantive change or deletion, respectively, from the previous revision except for Chapters 6, 10, and 11, which contain all new information..

Preface

This manual describes Revision 7.00 of the AOS/VS Operating System. It supersedes the *Advanced Operating System/Virtual Storage (AOS/VS) Programmer's Manual, Volume 1, System Concepts.*

This manual is for experienced assembly language programmers. If you have not done any assembly language programming on the AOS/VS system, you might want to read the following manuals first:

- *Learning to Use Your AOS/VS System* (093–000031). In this manual, Chapters 13 and 14 introduce you to assembly language programming on the AOS/VS system.
- *Advanced Operating System/Virtual Storage (AOS/VS) Macroassembler (MASM) Reference Manual* (093–000242). This manual describes the AOS/VS assembly language and the Macroassembler utility.

In this manual, *AOS/VS System Concepts*, we divide our description into the following chapters:

- Chapter 1 introduces the AOS/VS system.
- Chapter 2 describes virtual memory concepts and how you can manage your use of memory on an AOS/VS system.
- Chapter 3 describes what a process is and how you can create and manage processes.
- Chapter 4 describes how to create and manage files.
- Chapter 5 describes how to perform input/output (I/O) to both files and devices.
- Chapter 6 describes how to create and manage windows, how to perform I/O to windows, and how to create graphics applications that use windows.
- Chapter 7 describes how to create and manage a multitasking environment.
- Chapter 8 describes how to perform interprocess communications using the AOS/VS interprocess communications (IPC) facility.
- Chapter 9 describes how to create and manage process connections.
- Chapter 10 describes how to initialize physical processors and manage a multiprocessor system.
- Chapter 11 describes how to create process classes and create scheduling arrangements, called *logical processors*, using these process classes.
- Chapter 12 describes how to manage certain AOS/VS resources.
- Chapter 13 describes how to support user devices.

- Chapter 14 describes how to support binary synchronous communications applications.
- Chapter 15 describes how to manage 16–bit processes.
- Appendix A describes the format of the SYSLOG file, from which you can extract information on system use.

Some features of AOS/VS may change from revision to revision. Please see the latest AOS/VS Release Notice for information about functional changes and enhancements. You will find this Release Notice in the utilities directory (:UTIL) on your system tape.

Related Manuals

Within this manual, we refer to the following related manuals:

System Call Dictionary (AOS/VS and AOS/DVS) (093-000241).

How to Generate and Run AOS/VS (093-000243).

Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS) (093-000122).

Using CLASP (Class Assignment and Scheduling Package) (093-000422).

SWAT® Debugger User's Manual (093-000258).

ECLIPSE® 32-Bit Principles of Operation (014-000704).

In addition to the "Principles of Operation" manual listed above, you may find the following machine-specific supplements of interest:

ECLIPSE MV/2000™ DC System Principles of Operation (014-001203).

ECLIPSE MV/4000® System Principles of Operation (014-001226).

ECLIPSE MV/8000® Principles of Operation (014-001227).

ECLIPSE MV/10000™ System Principles of Operation (014-001228).

ECLIPSE MV/20000™ System Principles of Operation (014-001169).

ECLIPSE MV/4000® SC and Data General DS Systems Functional Characteristics (014-001066).

Reader, Please Note

In this manual, we use the following conventions:

Symbol	Meaning
< >	Angle brackets — indicate the paraphrase of an argument or statement, which you have to supply.
*	One asterisk — indicates multiplication. For example, 2*3 means 2 multiplied by 3.
**	Two asterisks — indicate exponentiation. For example, 2**3 means 2 raised to the 3rd power.

Unless the text specifies a specific radix, *we give all memory addresses as octal values, and all other numbers as decimal values.*

Finally, in the examples, we use

This typeface to show CLI and source file entries.

```
This typeface for data values returned by AOS/VS.
```

Contacting Data General

- If you have comments on this manual, please use the prepaid Remarks Form that appears after the Index. We want to know what you like and dislike about this manual.
- If you need additional manuals, please use the enclosed TIPS order form (USA only) or contact your Data General Sales representative.

End of Preface

Contents

Chapter 1 - Introduction to AOS/VS

Chapter 2 - Managing Memory

Chapter 3 - Creating, Managing, and Terminating Processes

Chapter 4 - Creating and Managing Files

Chapter 5 - Performing Input/Output (I/O)

Chapter 6 - Windowing

Chapter 7 - Initiating and Managing Tasks

Chapter 8 - Using the Interprocess Communications (IPC) Facility

Chapter 9 - Creating and Managing Process Connections

Chapter 10 - Managing a Multiprocessor Environment

Chapter 11 - Creating and Managing a Class Scheduling Environment

Chapter 12 - Using and Managing AOS/VS System Resources

Chapter 13 - Supporting User Devices

Chapter 14 - Supporting Binary Synchronous Communications (BSC)

Chapter 15 - Managing 16-Bit Processes

Appendix A - System Log Record Format

Tables

Table

Illustrations

Figure

Chapter 1
Introduction to AOS/VS

The Advanced Operating System/Virtual Storage (AOS/VS) is a 32–bit, demand–paged, virtual-memory operating system that runs on Data General's ECLIPSE® MV/Family of machines.

Because of its 32–bit addressing capability, its multiuser and multiprogramming support, and its compatibility with existing Data General software, AOS/VS is suited to both commercial and scientific applications. Specifically, AOS/VS provides you with the following:

- A logical address space of up to 2048 megabytes per process.
- Virtual memory management.
- Sophisticated process-protection schemes.
- Support for a multiprocessor environment.
- The option of creating process classes and scheduling processes by class.
- Compatibility with both the Advanced Operating System/Distributed Virtual System (AOS/DVS) and the Advanced Operating System (AOS).
- Support for concurrent 16– and 32–bit programs.
- A wide range of system and applications utilities.
- High-level language support.

What Is Virtual Memory?

Virtual memory allows you to run programs that are larger than the physical memory configuration of your system. With virtual memory, AOS/VS can move the active portions of a program from disk to memory while the program is executing. Then, when the system needs more memory, AOS/VS returns the inactive portions of the program to disk. This act of moving portions of the program in and out of memory is what we call *demand paging.*

An executing program is a *process.* The part of 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 the demands of the process change. AOS/VS determines the working set size by examining the number of pages that the process currently needs, as well as the process's 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. Consequently, to run your system as efficiently as possible, you should reduce the number of page faults. One way to reduce the number of page faults is to write your code in modules that cluster the instructions and data together as closely as possible. The fewer page faults your process causes, the smaller and more stable is its working set. Still, some page faults are unavoidable.

The Ring Structure of AOS/VS Memory

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. Because of the independence of the segments, AOS/VS can use each segment for a different function. This makes your virtual memory system very efficient and reliable.

Segment Protection

Each segment is protected by a ring that AOS/VS permanently binds to that segment. Ring 0 (the innermost ring) protects Segment 0, Ring 1 protects Segment 1, and so on through Ring 7 (the outermost ring), which protects Segment 7.

These rings prevent segments from interfering with one another; if a program that is executing in one segment needs to change or access the contents of another segment, it must observe strict protocols established by the rings. (The system observes these protocols without your knowledge.)

The Ring/Segment Hierarchy

AOS/VS arranges the eight segments and their rings 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 a single large logical address space, *context switching* — the transfer of control from one process to another — is often unnecessary. In fact, system calls and calls to routines that are in another segment become subroutine calls. While system calls require some participation by AOS/VS for their execution, a process does not have to switch contexts when it issues a system call. This allows you to avoid the additional processing overhead that context switching requires.

Ordinarily, a segment can only change or access the contents of segments that have segment and ring numbers higher than its own segment and ring number. For example, the rings will not allow a process running 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.

Subroutine Calls and Segment Protection

With a subroutine call, a segment having a segment number that is higher than or equal to that of the segment in which the subroutine resides can access that subroutine. In this case, the ring that protects the segment containing the subroutine 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. 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.

For information on the hardware instructions that allow you to define gates and reference code in the outer rings, see the "Principles of Operation" manual for your computer.

The Advantages of Using Inner Rings

The AOS/VS system allows you to write multitasked programs that will execute in more than one user ring (the user rings are Rings 4 through 7). Use of the inner rings gives you the following advantages:

- 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*. Local servers are servers that share the same logical address space as their customers. You can load a local server into the inner rings of a process.

  Local servers are faster than *global servers,* which must run as separate processes. Local servers do not need to use the interprocess communications (IPC) facility system calls or the ?MFBC and ?MTBC system calls to move data between customer and server. 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 rings to implement local servers, the server becomes part of the logical address space of the process that uses it, and 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 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).

Multiprocessor Support

The AOS/VS system supports multiprocessor hardware configurations. In a multiprocessor environment, important system processes, such as the AOS/VS kernal, run on an initial *mother* processor that the system manager brings up at VSGEN. Other processes can run on both the mother and *child processors;* you bring up a child processor *after* system initialization. To bring up a child processor, you can either use the CLI INITILIZE command, or you can use the system calls that we describe later in this book.

Process Scheduling Options

In addition to a standard scheduling arrangement, the AOS/VS system allows you to create other, special scheduling arrangements.

Under standard scheduling, AOS/VS schedules processes to run based on the priorities that the processes receive when you create them.

To create other scheduling arrangements, you must first create *classes,* into which you can group the processes on the system. You can then allocate the amount of processor time that each class can receive. For example, during the working day, you might want to use a scheduling arrangement that favors interactive processes. You could allocate up

to 75 percent of the available processor time to a class of interactive processes, and the remaining 25 percent to a class of batch processes. At night, you might use a different scheduling arrangement: up to 80 percent of the processor time for batch processes, and 20 percent for interactive processes.

AOS/VS supports class scheduling in both the uni- and multiprocessor environments. In fact, in the multiprocessor environment, AOS/VS allows you to use a different scheduling arrangement *for each physical processor,* should you choose to do so.

AOS/DVS Compatibility

AOS/VS and AOS/DVS are compatible; for most applications, you need only relink AOS/DVS-written programs to run them under AOS/VS. However, some AOS/DVS programs may require you to reassemble or recompile them before they can run under AOS/VS.

AOS Compatibility

AOS/VS and AOS are also compatible. Not only can you can run both 32–bit and 16–bit programs concurrently under AOS/VS, but, usually, you only need to relink AOS-written programs to run them under AOS/VS. However, some programs may require you to reassemble or recompile them before they can run under AOS/VS.

Your AOS/VS system's compatibility with AOS also extends to the file structure, magnetic-tape formats, and peripheral devices. You can transport disk files and tapes developed under AOS to AOS/VS without rewriting them. In addition, 16–bit device drivers written under AOS/VS can coexist with their 32–bit counterparts.

AOS/VS also supports overlays for 16–bit programs. However, AOS/VS *does not* support overlays for 32–bit programs. 32–bit programs do not need overlays because they take advantage of the AOS/VS system's virtual memory scheme, which allows 32–bit programs to exceed the size of the system's physical memory.

Certain system databases, such as task control blocks (TCBs) and the user status table (UST) are in the system's address space. Consequently, AOS-written programs that manipulate these databases without using task system calls need modification. AOS/VS does provide each program with a copy of the program's UST, but for reading purposes only. The AOS/VS system calls use 32-bit packets. If your AOS assembly language program uses the appropriate mnemonics for the packet offsets, you need only reassemble them with the new 16-bit parameter file (PARU.16) and relink them to run under AOS/VS.

System Calls

AOS/VS supports a wide variety of *system calls.* System calls are command macros that call on predefined system routines. There are various categories of system calls, which allow you to do the following:

- Manage the logical address space.
- Create and manage processes.
- Establish interprocess communications.
- Create and maintain disk files and directories.

- Perform file input and output.
- Create and manage windows.
- Create and manage a multitasking environment.
- Manage the system's physical processors.
- Create and manage process classes and the special scheduling arrangements that use these classes.
- Define and access user devices.
- Establish binary synchronous communications.
- Establish customer/server connections between processes.
- Perform input and output in blocks, rather than in records or lines.

This manual groups the system calls into functional categories, with a chapter that describes each category. The individual system call descriptions are arranged alphabetically in Chapter 13.

End of Chapter

Chapter 2
Managing Memory

The system calls that you use to manage memory are

?ESFF	Flush shared file memory pages to disk.
?FLUSH	Flush contents of a shared page to disk.
?GMEM	Return the current number of undedicated pages.
?GSHPT	List the current size of the shared partition.
?LMAP	Map a lower ring.
?MEM	List the current unshared memory parameters.
?MEMI	Change the number of unshared memory pages.
?PMTPF	Permit access to an open, protected shared file.
?RPAGE	Release a shared page and decrement its use count.
?SCLOSE	Close a shared file.
?SOPEN	Open a shared file.
?SOPPF	Open a protected shared file.
?SPAGE	Read a shared page and increment its use count.
?SSHPT	Establish a new shared partition size.
?VALAD	Validate a logical address.
?VALIDATE	Validate an area of memory.

This chapter describes how AOS/VS organizes memory. It also describes how a process — an executing set of instructions and system calls — can manage its own memory.

To understand this chapter, you must be familiar with the following terms and what they mean to AOS/VS:

- Logical context.

  Logical context refers to the total pages available to you, including shared, unshared, and unused pages.

- Logical address space.

  Logical address space is the entire range of locations that a process can address. A process's user-visible logical address space can be up to 512 megabytes for each user ring (we describe "user rings" and the AOS/VS memory structure below).

- Shared page.

  A shared page is a memory-resident page in your logical address space that is accessible to more than one process. Shared pages are usually write-protected to prevent overwriting. (See the "Shared Pages" section in this chapter for more information.)

- Unshared page.

  An unshared page is a page in your logical address space that only one process can access. Unshared pages cannot be write-protected.

- Unused page.

  An unused page is a page in your logical address space that is neither shared nor unshared. We describe the relationships between shared, unshared, and unused pages later in this chapter under the heading "User Context."

- Working set.

  Working set is the subset of a process's logical address space that resides in memory. The working set of a process changes in size and content as the process references pages, and then stops referencing them.

Ring Structure

The AOS/VS system divides its logical address space into eight 512–megabyte units called segments. Although strict protocols connect these segments, they are independent of one another. This allows AOS/VS to use each segment for a different function.

A hardware ring protects each segment (see Figure 2–1). Ring 0, the innermost ring, protects Segment 0, Ring 1 protects Segment 1, and so on through Ring 7, the outermost ring, which protects Segment 7.

The rings prevent segments from interfering with one another. If a program that is executing in one segment needs to change or access the contents of another segment, the program must observe strict protocols that the rings establish. (The system observes these protocols without your knowledge.)

AOS/VS arranges the eight segments and rings hierarchically. Segment 0 has the greatest ability to change or access the contents of other segments, while Segment 7 has the least. Similarly, Ring 0 gives Segment 0 the greatest protection from interference by other segments, while 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 and 5 are for both user programs and certain optional software supplied by Data General. Segments 6 and 7 are for user programs alone.

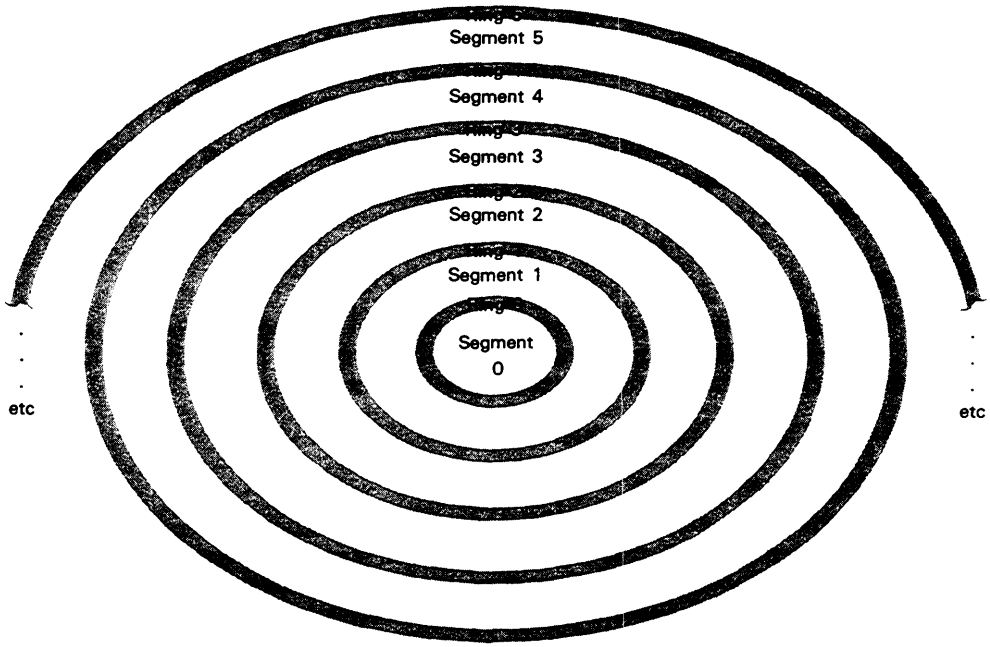


Figure 2–1. Segments and Their Protection Rings

Subroutine Calls

Because the user programs and the AOS/VS operating system share a single logical address space, *context switching* — the transfer of control from one process to another — is often unnecessary. In fact, system calls and calls to routines that are in another segment become subroutine calls. While system calls require some participation by AOS/VS for their execution, a process does not have to switch contexts when it issues a system call.

Ordinarily, a segment can only change or access the contents of segments that have a segment and ring number 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 that has a segment number 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 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. The only restriction on subroutine returns is that they must originate from a segment that has a number lower than or equal to the target segment.

For information on the hardware instructions that allow you to define gates and reference code in the inner rings, see the "Principles of Operation" manual for your Data General MV/Family computer.

User Rings

Ring 7 is the default user ring. However, you can load a program file into one of the other user rings (4 through 6) by issuing the ?RINGLD system call. AOS/VS also allows you to write programs that execute in more than one user ring.

By using the inner user rings, you can

- Improve software performance

  You can use the inner user rings to create local servers. Local servers, which you load into the inner rings of a process, are more efficient than global servers. Global servers must use system calls to move data between customer and server. Local servers, which reside in the same logical address space as their customers, can use instead the more efficient MV-series hardware instructions to perform the same tasks.

- Improve accounting.

  When you create local servers in the inner user rings, the local server becomes part of the logical address space of the process that uses it — the local server is not a separate process. A local server's use of resources is accounted for by AOS/VS as part of the resources that the customer's process uses.

- Expand the 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 all four user rings).

Demand Paging

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

The pages outside the working set make up the process's virtual address space. In Chapter 3, Figure 3–2 shows the working sets and virtual address space of several processes.

Prepaging at Fault Time Option

By default, when a page fault occurs — 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. We call this option *prepaging at fault time*. Prepaging is the process of adding unreferenced virtual pages to a working set.

The prepaging option is useful when

- A program includes large array-like data structures for which the virtual addresses exceed the amount of main memory available.
- The algorithm that processes a large data structure references the entire structure or parts of it sequentially.
- The area in which you want prepaging to occur is in unshared or unused memory.

If your program has such characteristics and you understand its page referencing patterns, prepaging 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 prepaging option, your system manager must set the prepaging parameter during the VSGEN dialog. The prepaging parameter specifies the maximum number of pages that you can add to the working set for each page fault. If the system manager has set the parameter to 0 or 1, then prepaging is turned off system-wide.

If prepaging is enabled, you must use the SPRED utility to edit the preamble of your program file to 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.

For more information on the prepaging option and the VSGEN dialog, see *How to Generate and Run AOS/VS*.

Program Load Option

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

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

- Is short.
- Runs briefly.
- Frequently references a large unshared area.

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

To load your program initially

- Your system manager must have previously enabled the initial program load option during the VSGEN dialog. The system manager enables this option by indicating the number of pages that a process can have at initial load time.
- You must specify the address range of the area that you want to load in the preamble of your program. To specify the address range, use the SPRED utility to edit the preamble of your program file.

For more information of the initial program load option, see *How to Generate and Run AOS/VS*.

Variable Swapfiles

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

When heavy memory contention occurs, the system picks a process to swap out to disk. The system swaps the process out in a *swap file*. Each process has its own swap file in the SWAP directory.

By default, swap files have a fixed size. The fixed size of the swap file can be a disadvantage for those processes whose working sets exceed the size of the swap file. To swap such a process out to disk, the system must break the process's working set up into several swap files. When the system later swaps this process back into memory, the process must incur a series of page faults to restore the process's working set to memory. For processes with large working sets, this paging can be costly.

To avoid breaking up large working sets into several swap files, you can set up a system to allow swap files that vary in size from process to process.

To enable the use of variable swap files

- The system manager must enable the use of variable swap files and specify a default and a maximum swap file size during the the VSGEN dialog.
- The system manager must give those users who run programs with large working sets the privilege of changing their swap file size.

- The privileged user must specify a size for the swap file that is equal to the program's working set size by editing the preamble of the program files with the SPRED utility.

For more information on enabling variable swap files and the VSGEN dialog, see *How to Generate and Run AOS/VS*.

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 that multiple users can read and (possibly) modify.

The system monitors the use of shared pages. If a shared page is not used, that page remains in memory only as long the demand for memory is low. If the demand for memory exceeds the amount of memory that is available, the system releases the unused shared page and places it on a least recently used (LRU) chain.

An LRU chain is a list of released shared pages that the system arranges in least recently used order. Any process can reuse the shared pages on the LRU chain. (See Figure 2–2.) You can conserve memory by using shared pages, because they allow more than one process to use the same re-entrant code or data. Also, shared pages reduce disk I/O, because AOS/VS does not immediately swap them to disk when a process releases them. Instead, it retains shared pages in a cache-like collection in memory for other processes to use.

The ?SPAGE system call reads one or more contiguous pages of a disk file into the shared area of the caller's logical address space. If the ?SPAGE system call tries to read pages that are beyond the disk file's end-of-file (EOF), AOS/VS writes zeros into those pages, allocates them to the disk file, and then reads those zeroed pages into shared area.

The ?RPAGE system call releases one or more shared pages from the caller's logical address space, but may retain them in memory. If you have modified a shared page and you want to release and update that page immediately, you must issue either a ?FLUSH system call, a modified version of the ?RPAGE system call, or the ?ESFF system call. Each of these system calls provides a way to write 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 on.
- 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.

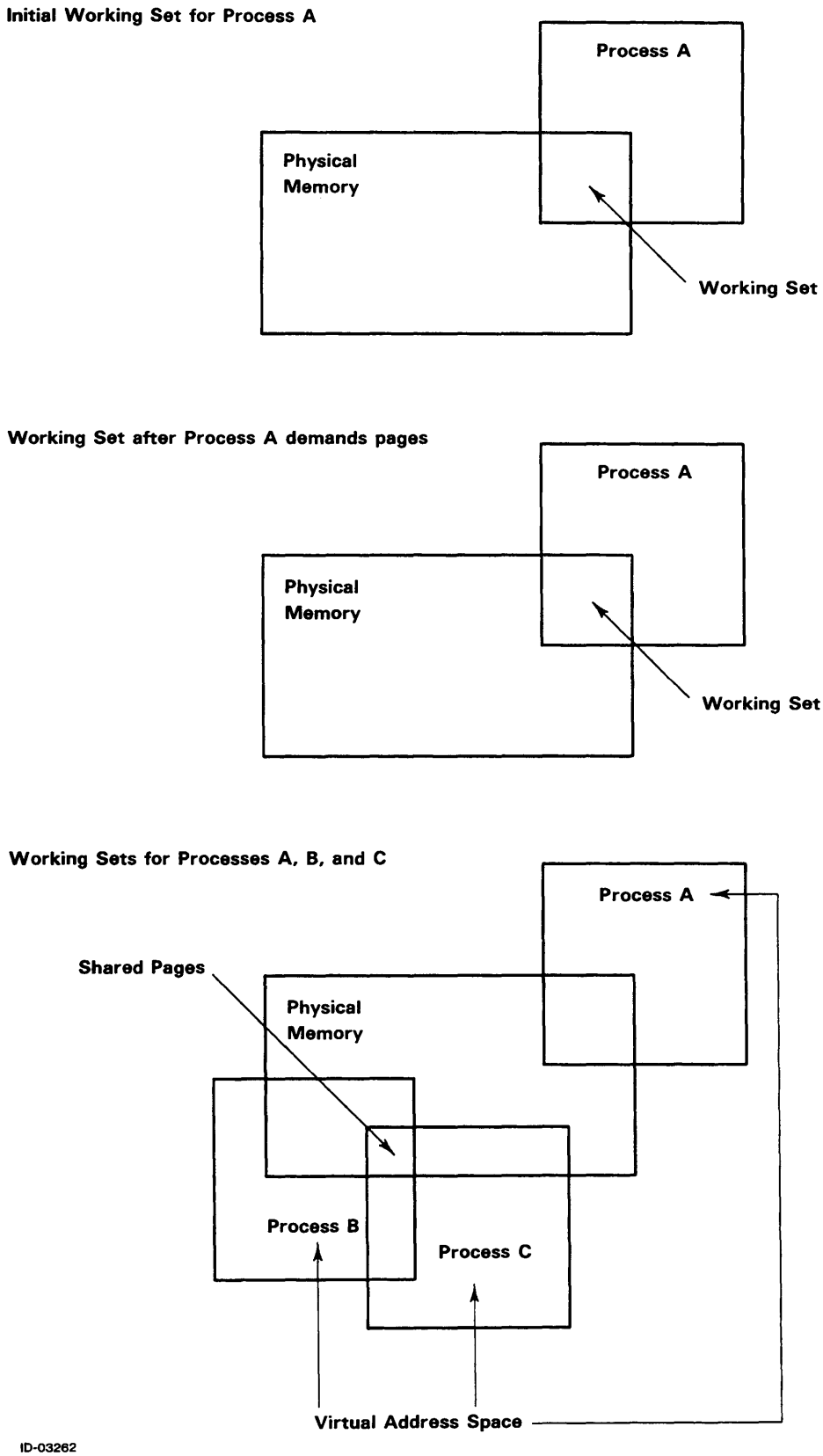


Figure 2–2. Working Sets in Memory

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 normal relocatable (NREL) 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 statements .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 the .PART pseudo–op. This pseudo–op allows you to define a variety of attributes (characteristics) for the partition, including whether it is part of 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 process that uses the program file.

For information on using the ?OPEN system call for page sharing, see Chapter 5.

Protected Shared Files

A set of common local 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 of 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 calls that open the same shared file. For more information on the difference between first and subsequent opens, see the individual system call description of the ?SOPPF system call in the *System Call Dictionary (AOS/VS and AOS/DVS)*.

The segment image that uses the ?SOPPF system call to open a protected shared file for the first time is 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 other segments can then issue a somewhat different form of the ?SOPPF system call to open the file.

Only the first opener of a protected shared file can issue a ?PMTPF system call against that file. To complete this system call successfully, the connection between the PID/ring tandem that issues the ?PMTPF system call (the server) and the PID/ring tandem of the target (the customer) must be valid.

The ?PMTPF caller also informs AOS/VS of the type of file access privileges that it wants to pass to another segment image. The caller can only pass on those priviliges that it already has. 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 another ?PMTPF with fewer or no access privileges.

Coordinated Shared-File Update

Periodically, inner-ring servers may need to record the checkpoint state of a set of shared memory pages — this may be critical for recovery from system failure.

The ?ESFF system call helps record the checkpoint state of shared memory by flushing to disk all modified pages associated with a specified shared file, no matter where they are in system memory. AOS/VS tries to flush all modified shared pages, even if it encounters an I/O error while it is flushing the pages.

The ?ESFF system call makes only one pass through the pages in a shared file. Should another process (or other tasks within the same process) concurrently update the shared file, the checkpoint state will be uncertain.

Validating Memory

In some situations, particularly when developing inner ring servers, you can avoid those user traps caused by referencing an invalid area of memory by first validating your access privileges to that memory area. To validate a logical address, you can use the ?VALAD system call. To validate a specific range of addresses, you can use the ?VALIDATE system call. For example, if a ring 7 caller provides a ring 4 address to a ring 4 server, you could use the ?VALIDATE system call to indicate that the caller's ring is 7. As a result, the call would return a validation error to protect the server.

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.

Managing Your Memory Context

Your unshared area starts at the first word of the logical address space in the current ring, and expands 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. When a process issues the ?MEMI system call to allocate pages to the unshared area, AOS/VS zeros those pages before allocating them.

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 2–3 shows the relationship among the unshared, unused, and shared areas in a typical user context.

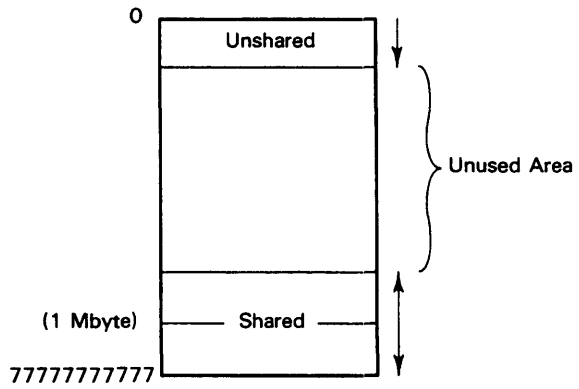


Figure 2–3. Memory Context

Table 2–1 lists the system calls that you can use to manage a process's logical context.

Table 2–1. Context-Management System Calls

System Call	Function
?ESFF	Flushes shared file memory pages to disk (shared pages only).
?FLUSH	Flushes contents of a shared page to disk.
?GSHPT	Lists shared-partition information for this context (shared pages only).
?MEM	Lists the maximum number of unshared pages available, the number of unshared pages used, and the highest currently used unshared address in Ring 7 (unshared pages only).
?MEMI	Increases or decreases the number of unshared pages in Ring 7 (unshared pages only).
?PMTPF	Permits access to an open, protected shared file (shared pages only).
?RPAGE	Releases a shared page (shared pages only).
?SCLOSE	Closes a shared file (shared pages only).
?SOPEN	Opens a file for shared access (shared pages only).
?SOPPF	Opens a protected shared file (shared pages only).
?SPAGE	Reads a shared page (shared pages only).
?SSHPT	Establishes a new shared-partition size (shared pages only).

End of Chapter

Chapter 3
Creating, Managing, and Terminating Processes

You can use the following system calls to create, manage, and terminate processes:

?AWIRE	Change wiring characteristics of Agent portion of a resident process.
?BLKPR	Block a process.
?BRKFL	Terminate a process and create a break file.
?CHAIN	Pass control to a new program.
?CTYPE	Change a process's type.
?DADID	Get the PID of a process's father.
?ENBRK	Enable a break file.
?EXPO	Set, clear, or examine execute-protection status.
?GBIAS	Get the system's current bias factors.
?GLIST	Get a process's search list.
?GPID	Get a list of active PIDs on a remote host.
?GUPID	Translate a host ID and PID into a VPID.
?GUNM	Get the username of a process.
?HNAME	Return a host ID or host name.
?IHIST	Start a histogram for a 16–bit process.
?KHIST	Terminate a histogram.
?LOCALITY	Change a process's user locality.
?MDUMP	Dump the memory image from a specified ring to a file.
?PCLASS	Get a process's class ID, and user and program localities.
?PIDS	Return information on local PIDs.
?PNAME	Get a process name.
?PRIPR	Change the priority of a process.
?PROC	Create a process.
?PSTAT	Return status information on a process.
?RESCHED	Schedule another process for execution.
?RETURN	Terminate the calling process and transfer control back to its father.
?RINGLD	Load a program file into a specified ring.
?RNGPR	Return the .PR filename for a ring.
?RNGST	Stop lower rings from being loaded.
?RUNTM	Get runtime statistics on a process.
?SBIAS	Set the system's bias factors.
?SONS	Return a list of son processes.
?SUPROC	Enter, leave, or get status of Superprocess Mode.
?SUSER	Enter, leave, or return status of Superuser Mode.
?SYSPRV	Enter, leave, or get status of System Manager Mode, Superprocess Mode, or Superuser Mode.
?TERM	Terminate a process.
?TPID	Translate a VPID into a host ID and PID.
?UBLPR	Unblock a process.
?UNWIRE	Unwire pages previously wired.
?WHIST	Start a histogram for a 32–bit process.
?WIRE	Wire pages to the working set.

This chapter describes processes and how they use memory. It also describes how AOS/VS schedules processes for processor time. The chapter then describes how to create, manage, and terminate processes.

What Is a Process?

A *process* is an executing set of instructions. A process can contain both instructions that you have written and instructions that the operating system has provided as a resource. The file that contains these instructions is a *program.*

Each process can contain one or more *tasks*, which execute *asynchronously* (that is, at different times). You can design your program so that several tasks execute a single re-entrant sequence of instructions, or so that each task executes a different instruction path. The AOS/VS system always gives processor time to the highest priority *ready task* within the highest priority *ready process.* We describe process priority later in this chapter. For more information on tasks, see Chapter 7.

Processes and Virtual Memory

As we described in Chapter 2, AOS/VS is a virtual memory, demand-paged system. Virtual memory means that memory is a composite of physical (computer) memory and disk memory. Demand-paged means that AOS/VS adds a page to each process's working set of pages on process demand. AOS/VS releases unused pages as processes require more memory.

Each process starts with a certain number of pages of virtual memory — its initial *working set.* The working set is a subset of the process's total *logical address space.* When the process needs more pages (perhaps to execute a routine that isn't in memory), a *page fault* occurs; AOS/VS then adds an additional page to the process working set. The theoretical limit on the number of pages in a process's working set exceeds 1,000,000 — providing a limit of 2 gigabytes on any process's logical address space. (See Figure 3–1.)

There can be many processes running simultaneously. Memory contention occurs when all currently active processes (including the AOS/VS system and its peripheral manager) desire a working set larger than the computer's physical memory. Memory contention can occur much of the time.

In *light* memory contention, AOS/VS resolves the situation by removing *inactive pages* from processes and storing their images in its PAGE directory. The processes remain in physical memory, but with fewer pages in their working sets. Later, if demanded, the system restores these pages to the working sets. This is called *paging.*

In *heavy* memory contention, AOS/VS removes whole *processes* (selecting blocked processes first), and stores their images in its SWAP directory. AOS/VS removes their entire working sets from memory. Later, AOS/VS restores the working sets to physical memory and the processes can run again. This is called *swapping.*

If there is no memory contention, no paging or swapping occurs. But if there is contention, AOS/VS may page or swap processes to disk on the basis of their process types and priorities.

Getting control of a job processor is a two-phase operation. Before a process can do it, its working set must be in physical memory.

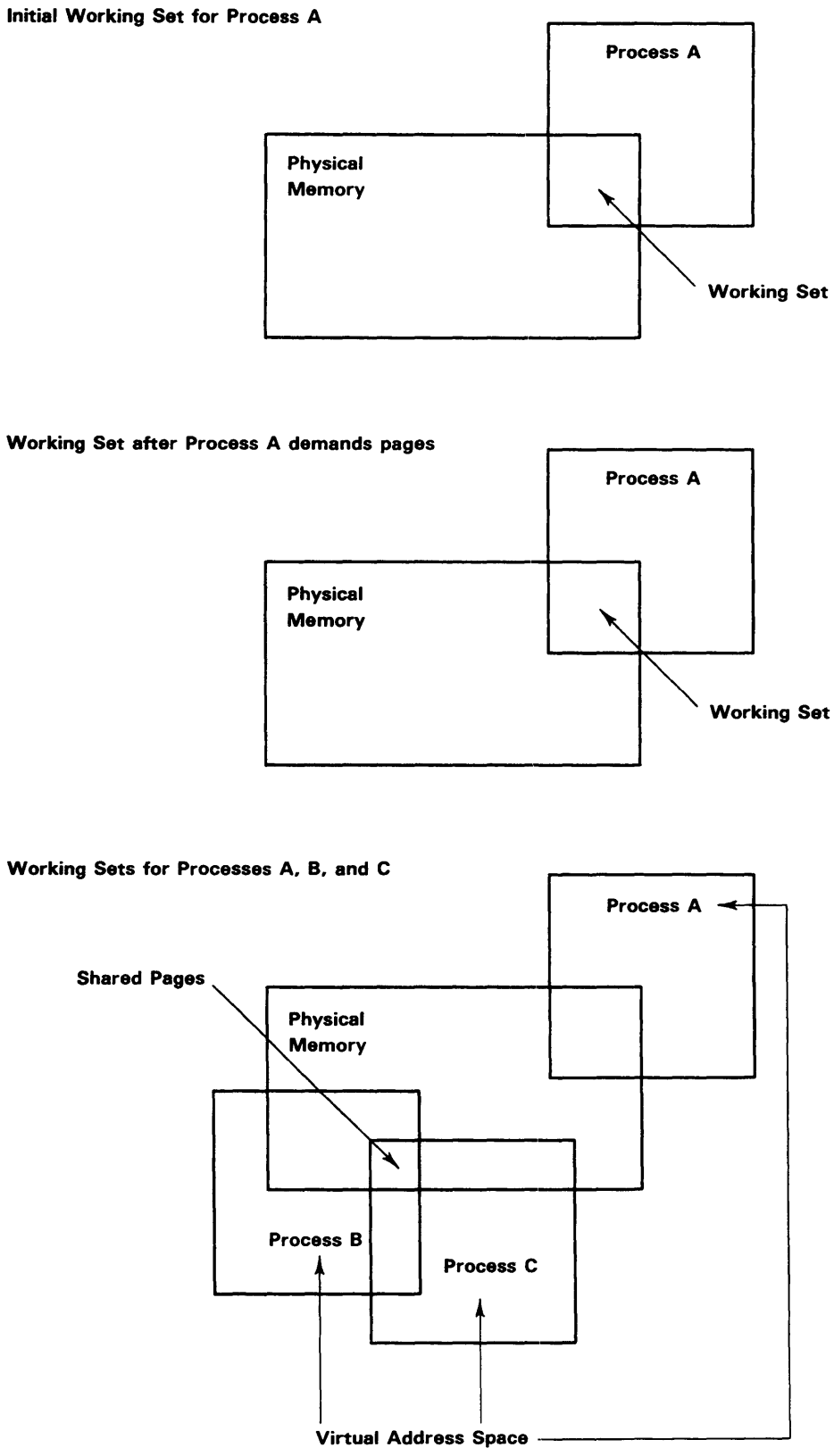


Figure 3–1. Working Sets in Memory

Processes and Physical Memory

AOS/VS gives processes physical memory according to *type*; scheduling characteristics, such as process class and priority, are secondary. The process types are

- Resident.
- Pre-emptible.
- Swappable.

The AOS/VS system scheduler allocates memory for each type as follows:

Process Type	How It Gets Physical Memory
Resident	Gets memory on demand and keeps it. The scheduler retains the process initial working set in memory; the system may page the process, but may not swap it.
Pre-emptible	Gets memory if the memory is not needed by a resident process. The scheduler swaps a pre-emptible process if • A resident or higher priority pre-emptible process requires memory; or • The process becomes blocked and *any* other process requires memory.
Swappable	Gets memory if the memory is not required by a resident or unblocked pre-emptible process. The scheduler can swap the process as needed during memory contention.

By default, all user processes, including the batch processes that the system creates for users, are swappable.

Adjusting the Size of the Working Set

When you create a process, you can specify the size of its initial working set. You can also adjust the size of its working set once a process has gained memory.

By default, when a process first gains memory, the system allocates enough memory for the initial working set to include both page 0 (the first 2 kilobytes of the logical address space) and the program counter (PC) page. However, when you create a process (?PROC system call), you can set both a minimum and a maximum working set size. This allows you to set a minimum working set size that exceeds the initial default working set size. It also allows you to limit the amount of memory that the process can use.

The process's *type* also affects the size of the working set. For example, a resident process always has a larger initial working set than a pre-emptible or swappable process. When you create a resident process or change a process into a resident process (?CTYPE system call), the system automatically binds those pages to the working set that contain the Agent portion of the process. (The Agent is that part of the AOS/VS system that pre-processes system calls and serves as an interface to the operating system.)

You can also remove, or *unwire*, all Agent pages (except those needed for user device support) from the resident process by using the ?AWIRE system call. By unwiring the Agent pages from a resident process, you make more memory available and improve system performance; however, the resident process may be less efficient.

You can wire and and unwire memory pages to the working set of any process by using the ?WIRE and ?UNWIRE system calls. The ?WIRE system call wires pages to the working set. The ?UNWIRE system call releases previously wired pages. You should be careful, however, not to wire too many pages to the working set of a resident process. Should memory contention occur, the system will not be able to swap these pages to disk. This reduces the amount of memory available and degrades system performance.

If the system manager has enabled the pre-paging option in the VSGEN dialog, you can also specify the number of pages that the system adds to the working set when a page fault occurs. For more information on the pre-paging option, see Chapter 2.

Process Scheduling

The AOS/VS system schedules processes for processor time in one of two ways: *standard scheduling* (the default) and *class scheduling.*

Standard Scheduling

Standard scheduling works as follows: when a job processor becomes free, the highest priority, ready process gains control of that processor. This process will then use the job processor for a full *subslice* period of 32 milliseconds, unless one of the following events takes place:

- The process encounters a blocking event.
- Another higher priority process becomes ready.

Process Priority

The AOS/VS system determines the relative priority of processes based on their priority numbers. These priority numbers range from 1, the highest priority, to 511, the lowest priority.

The AOS/VS system divides these process numbers into three ranges, or "groups", between which there are no gaps or overlaps. In the VSGEN dialog, the system manager specifies the range of priority numbers in each group. The ranges for these groups are

- Group 1 — high priority — default priority numbers 1 through 255.

  Includes resident and pre-emtible processes with priority 1, 2, or 3. Also includes processes with priorities 4 through n, where n is the VSGEN "lowest priority for group 1" that the system manager specifies.

- Group 2 — medium priority — default priority numbers 256 through 258.

  Includes priorities n + 1 through m, where m is the VSGEN "lowest priority for group 2" that the system manager specifies.

- Group 3 — low priority — default priority numbers 259 through 511.

  Includes priorities m + 1 through 511, the lowest priority.

The group into which a process's priority falls determines the type of scheduling that the system uses for that process. The system uses two types of scheduling: *round-robin* and *heuristic.*

With round-robin scheduling, the system gives processes of equal priority an equal amount of processor time, *without regard for their behaviour.* The system schedules both Group 1 and Group 3 processes on a round-robin basis.

With heuristic scheduling, the system *schedules processes according to their behavior*, favoring interactive processes over processor-intensive, noninteractive processes. The system schedules Group 2 processes heuristically.

For example, based on a Group 2 process's prior behavior, the system will expect that process to have a certain number of blocking events within a certain time. If the process has *more* blocking events within that time, the system will increase the process's priority, giving it a better chance at getting processor time. However, if the process has *fewer* blocking events, it is starting to monopolize the processor; the system then reduces that process's priority so as to give more interactive processes a better chance at getting processor time.

Changing Process Priority — the ?PRIPR System Call

You can change the priority of a process by issuing the ?PRIPR system call. You can issue the ?PRIPR system call from either the process itself, or from another process. However, if you issue the ?PRIPR system call from another process, that process *must* be in Superprocess mode (we describe Superprocess mode later in this chapter).

Changing Process Priority — by Process Type

A process's type can affect its priority. While a resident or pre-emptible process can have any of the priorities 1 through 511, a swappable process *cannot* have any of the priorities of a Group 1 process. Consequently, if you change a Group 1 resident or premptible process into a swappable process, the AOS/VS system will lower the actual priority of that process (see the "System Mapping of Priority Numbers" section that follows).

By default, when you create a process, that process is of the same type and priority as its father process (the process from which you create it). However, in the ?PROC system call, you can specify that the new process be of a different type from that of its father process. You can also change the process type of an existing process by issuing the ?CTYPE system call.

System Mapping of Priority Numbers

To maintain compatibility with the AOS (16–bit) operating system, the AOS/VS system "maps" the priority numbers of swappable processes: The actual priority of the swappable process, and the priority that the AOS/VS system displays in response to the CLI PRIORITY command, may be different. The system uses the "actual", undisplayed priority to schedule the process for processor time.

Given that the boundries of the three scheduling groups are

Group 1 Priorities 1 through n.

Group 2 Priorities n + 1 through m.

Group 3 Priorities m + 1 through 511.

the AOS/VS system maps priority numbers as shown in Table 3–1.

Table 3–1. Priority Mapping — Resident/Pre-emptible to Swappable

Original Priority Before Change (Resident/Pre-emptible)	Displayed Priority After Change (Swappable)	Actual Priority After Change (Swappable)
1 through 3	1 through 3	n + 1 through n + 3 *
4 through n	2	n + 1 through n + 3 *
n + 1 through n + 3	1 through 3	n + 1 through n + 3
n + 4 through m	n + 4 through m	n + 4 through m
m + 1 through 511	m + 1 through 511	m + 1 through 511

* Compatible with AOS priority mapping.

When you change a swappable process to a resident or pre-emptible process, the AOS/VS system gives the resident or pre-emptible process the swappable process's *displayed* priority, not its actual priority. (See Table 3–2.)

Table 3–2. Priority — Changing from Swappable to Resident/Pre-emptible

Actual Priority of Swappable Process	Displayed Priority of Swappable Process	Resident/Pre-emptible Priorty After Change
n + 1 through n + 3	1 through 3	1 through 3 *
4 through n	4 through n	4 through n
n + 4 through 511	n + 4 through 511	n + 4 through 511

* Compatible with AOS priority mapping.

Bias Factors for Interactive and Compute-Bound Processes

Under standard scheduling, AOS/VS generally keeps interactive processes in memory longer than noninteractive, processor-intensive processes. However, you can adjust the way in which standard scheduling does this by setting the system's *bias factors*. The system's bias factors determine both the *minimum* and *maximum* number of noninteractive, compute-bound processes that AOS/VS will try to keep in memory.

To set the system's bias factors, issue the ?SBIAS system call. To return the system's current bias factors, issue the ?GBIAS system call.

Class Scheduling

A *class* is a set of processes that gets special scheduling treatment. Usually, this treatment involves the allocation of a percentage of processor time. A *logical processor*, which is a scheduling arrangement for one or more classes, determines the amount of processor time that each class can use.

Each class has a name and one or more user and program localities, called *locality pairs*, that identify it. When class scheduling is enabled, a process will run in a specific class if its locality pair matches one of those that the system manager has defined for that class.

When the system manager creates a user profile with PREDITOR, the system manager specifies both the user's default user locality, and any other user localities to which the user can assign a process. The selective preamble editor (SPRED) determines the process's program locality in the program file.

You can create process classes, and the logical processor(s) that schedule them, with either the system calls that we describe in Chapter 9, or, if you have it on your system, with the CLASP (Class Assignment and Scheduling Package) utility. For more information on the CLASP utility, see *Using CLASP (Class Assignment and Display Package)*.

Primary Versus Secondary Classes

A logical processor can allocate processor time for up to 16 classes. It can rank each class as either *primary* or *secondary*.

- Primary class — has a guaranteed percentage of processor time. Processes in a primary class can use up to their assigned percentage of proceesor time.
- Secondary class — no guaranteed percentage, but ranked in levels. After ready processes in the primary classes get time, secondary processes compete for time by level.

Under class scheduling, the highest priority, ready process gets control of the processor, *unless*

- The process belongs to a primary class that has used up its percentage.
- The process belongs to a secondary class, and a higher level secondary class or a primary class process is ready.

In either of these cases, the process can't get processor time in the current interval, regardless of its priority.

Specifying User Localities at Process Creation

By default, when you create a process, that process has the same user locality as its father process. If the father process has the privilege to change its user locality, the son process will also have this privilege. The son process can then change its user locality to any of those localities to which its father process is privileged to change.

In the ?PROC system call, you can specify that the process have a different user locality from that of its father process. By default, this locality must be one of those to which the father process is privileged to change.

You can also give the new process a set of user localities that it is privileged to use, which is different from that for which its father process is privileged to use. By default, the set of user localities for this process can contain *only* those localities to which the father process is privileged to change.

Changing the User Locality After Process Creation

Once a process exists, it can use the ?LOCALITY system call to change its user locality. By default, the new user locality can only be one of the user localities for which the process is privileged.

A process can also use the ?LOCALITY system call to change the locality of another process in its sub-tree (the target process must be a son, grandson, and so on). The calling process can only change the target process's locality to one for which the calling process is privileged. The calling process can also change the set of user localities for which the target process is privileged.

Getting Locality Information

To determine the current user locality of a process, and the other user localities to which it is privileged, you can use the ?PCLASS system call. By determining those user localities for which a process is privileged, the ?PCLASS system call helps you to avoid assigning a process to a user locality for which the process is not privileged.

Assigning Localities without Restriction — System Manager Mode

If a process has the system manager privilege — given by PREDITOR in the user profile — and has turned on System Manager mode with the ?SYSPRV system call, the process can

- Assign *any* user locality to a new process, and give that process privileges to *any* other user localities.
- Change the user locality of *any* existing process to *any* user locality.

We describe the ?SYSPRV system call in Chapter 9.

Rescheduling a Process

If an executing process cannot proceed, you can issue the ?RESCHED system call, which allows the calling process to give up control of the processor and forces AOS/VS to immediately schedule another process for execution.

Process Hierarchy

When you initialize the system, AOS/VS creates a process called the *system root*, from which *all* other processes proceed. From the system root, AOS/VS creates the system processes, such as the peripheral manager (PMGR), which manages character I/O. AOS/VS also creates at least one user process, called the initial (operator) process. The initial process can create subordinate processes, or sons, and assign them a process type and priority number.

AOS/VS organizes processes into a hierarchical tree structure, where processes on the lower levels are subordinate to their relatives on the higher levels. (See Figure 3–2.)

The system root is the highest process in the system hierarchy; every other process is a son of the system root. User processes are sons of the initial process.

Process Identification

A process name and a process identifier (PID) identify each process. When you create a process, you assign the process its name. At the same time, AOS/VS assigns the process a PID. The PID can be in the range from 1 through n, where n is the maximum number of processes that you specified at VSGEN (the default maximum is 255).

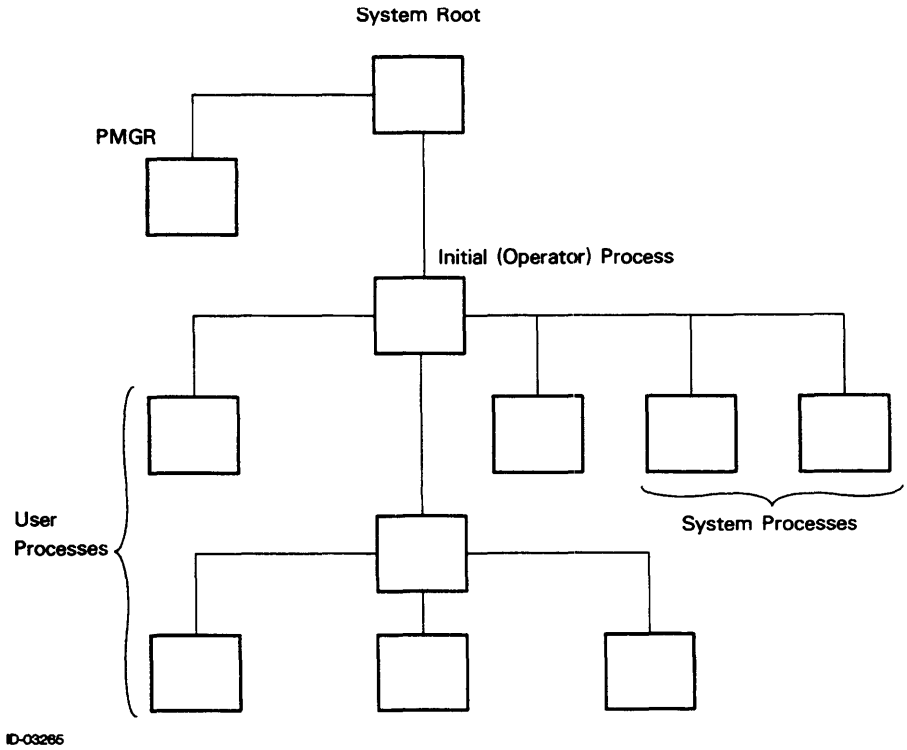


Figure 3–2. Process Hierarchy

Specifying a Process Name

A full process name is a character string that consists of a username and a simple process name, with a colon (:) between the two elements. Each element can contain up to 15 valid filename characters. The valid filename characters that you can include in the process name are

- Letters A through Z. (AOS/VS treats uppercase and lowercase letters the same.)
- Numbers 0 through 9.
- Period (.), dollar sign ($), question mark (?), and underscore (_).

AOS/VS uses the username part of the process name to determine the process's geneology and its file access privileges. By default, each son process has its father's username. A father process can assign its sons a different username only if the father was created (by issuing the ?PROC system call) with the privilege to do so.

You can use either a 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. (See Figure 3–3.)

Figure 3–3 shows a process with the full process name SAM:PROC2, where SAM is the username and PROC2 is the simple process name. If you issue a system call from SAM:PROC1 with the simple process name PROC2 as an input parameter, AOS/VS recognizes the target process as SAM:PROC2.

You cannot specify the same simple process name for processes that have the same username. If you do, AOS/VS returns error code ERPNU (process name already in use).

The Process Identifier (PID)

When you create a process, the AOS/VS system gives it a PID that is in the range appropriate to that process. The appropriate range for a process depends on both the maximum number of processes allowed on the system, and on whether you created the program that is executing prior to or under Revision 7.00 of AOS/VS.

Prior to Revision 7.00 of AOS/VS, up to 255 processes could run on the system at one time. Consequently, the process identifiers ranged from 1 to 255. However, as of Revision 7.00, you can allow the system to run a higher number of processes — up to 4096 — by so specifying in the VSGEN dialog. Consequently, the PIDs can range from 1 to whatever number you specify as the

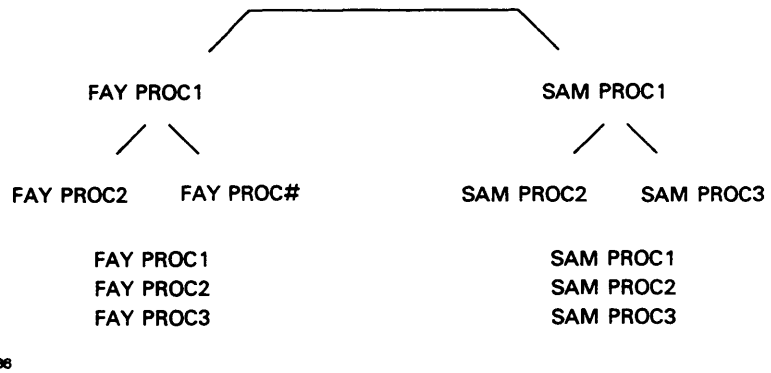


Figure 3–3. Process Names

maximum (up to 4096). Under Revision 7.00, the default number of processes that the system will run is 255.

If the system allows the default maximum of 255, programs written under previous revisions of the operating system will run without the need for alteration. But, if the system allows a maximum number of processes exceeding 255, you must determine if any program files written under previous revisions of the operating system have any PID-related restrictions (we describe PID-related restrictions in the next section).

After you have identified and corrected any PID-related restrictions, you must

1. Assemble and link the new program.
2. Use SPRED, the selective preample editor, to change the program's *PID-size type*. The program's PID-size type determines whether or not, and under what conditions, a program can run using a PID above 255.

We describe the PID-size types after we describe the types of PID-related restrictions that you might find in a program.

PID-Related Restrictions in Program Files

If a program issues a system call for which there is a PID-related restriction, you may have to change that program before you can run it successfully. The system calls with PID-related restrictions are ?PSTAT, ?IREC, and ?EXEC. You can check your program files for these system calls by either checking your source files directly, or by running the macro PIDCALL CHECK, shipped with AOS/VS Revision 7.00, against the program file.

If a program *does* issue one or more of these calls, check the context in which it issues the call. If the program issues the call in a context that isn't limited, you don't need to change the program.

?PSTAT System Call — The ?PSTAT call returns status information on processes, including a list of sons; *however,* ?PSTAT can list only sons with PIDs 1–255. ?PSTAT has no other PID-related limitation.

If a program uses the ?PSTAT system call, and relies on the son information returned in the ?PSTAT packet, you can add the ?SONS system call to the program. The ?SONS system call can return information on all son processes. The program must get the sons information from the ?SONS buffer, not from the ?PSTAT packet.

?IREC System Call — If the program uses the ?IREC system call to listen for termination or obituary messages from a connected process, the program must change the way in which it interprets the message. The format of the message in the ?IREC packet has changed. We describe the ?IREC system call, which you use for interprocess communication, in Chapter 8, and the management of process connections in Chapter 9.

?EXEC System Call — If the program uses the ?XFSTS function of the ?EXEC system call, which can return only PID numbers up to 255, you should replace the ?XFSTS function with the extended function, ?XFXTS. The ?XFXTS function can return *any* PID number, including those above 255.

Program and Process PID-Size Types

Under Revision 7.00 of AOS/VS, if you specify a maximum number of processes that exceeds 255 at VSGEN, then each program and process falls into one of three PID-size types. Table 3–3 describes these program and process PID-size types.

Table 3-3. Program and Process PID-Size Types

Program PID-Size Type	Process PID-Size Type
SmallPID-type program. A smallPID program can't run if PIDs 1 through 255 are in use. SmallPID is the PID-size type of all programs before AOS/VS Revision 7.00 (except the CLI and EXEC, which were hybrid in Revision 6.00). The Link program creates programs of SmallPID-size by default.	Type A process. A type A process has a PID between 1 and 255. It can't execute any program if PIDs 1 through 255 are in use. Error conditions may result if a process with a PID over 255 tries to communicate with a type A process. This is the PID-type of all processes before AOS/VS Revision 7.00 (except the CLI and EXEC).
Hybrid program. A hybrid program's program file has been edited with the SPRED editor and its PID-size type made hybrid. (You can tell SPRED to label *any* program as hybrid, but if the program has any small-PID limitation, the process may not be able to communicate with PIDs above 255. And, since AOS/VS thinks the process is a legitimate hybrid, it won't detect PID-range errors. A hybrid program can't run if PIDs 1 through 255 are in use. Most programs shipped with AOS/VS Revision 7.00 are hybrid programs.	Type B process. A type B process has a PID between 1 and 255. It can't run if PIDs 1 through 255 are in use, but can create and communicate with a process of any PID-size type. Most DG programs, including the CLI and EXEC, run as type B processes. By default, the CLI run for each user is a type B process; by default, a user CLI must run in the range of 1 through 255, but can execute any PID-type program. Most processes from programs supplied with AOS/VS are type B processes.
AnyPID program. An anyPID program's program file has been edited with the SPRED editor and its PID-size type made anyPID. (As with a hybrid program, you can tell SPRED to label any program as anyPID, but if the program has small-PID limitations, communication errors may occur and go undetected by AOS/VS.) An anyPID program can run at any PID up to the maximum specified at VSGEN. The system will run it above 255 if possible.	Type C process (if a PID above 255 is free). Type B process (if no PID above 255 is free). A type C process can execute any PID-size type program. But error conditions may arise after it executes a smallPID program (since the father process has a PID the son can't understand).

Because an anyPID program can run at any PID, it's the most flexible PID-size type. Its only disadvantage appears when it has a PID above 255, where type A processes may not be able to communicate with it. You can avoid this problem by making all smallPID programs into hybrid or anyPID programs

Changing the PID-Size Type

To change a program's PID-size type, run the SPRED program, edit choice 6, specify the PID-size type desired, apply changes (choice 8), and leave SPRED by typing **BYE**. To do this, you need write access to the program file.

(Remember, don't use SPRED to make a smallPID program into a hybrid or anyPID program until you've made sure that the program doesn't have any PID-related limitations. At least, run the PIDCALL_CHECK macro on it to check for any potentially limiting system calls; if the macro finds any, don't change the program PID-size type until you've determined whether or not those system calls actually limit the program.)

The Virtual Process Identifier (VPID)

The AOS/VS system uses the *virtual process identifier* (VPID) to identify a process on a remote host. A VPID consists of a host ID-PID combination.

Some system calls require a VPID as input. You can translate a host ID and PID into a VPID by issuing the ?GVPID system call. Conversely, you can break a VPID down into its component host ID and PID by issuing the ?TPID system call.

If, before you issue the ?GVPID system call, you need to get a host ID, you can issue the ?HNAME system call. The ?HNAME system call returns either the host ID or host name, as you specify.

Depending on your input specifications, the following system calls return the process name and/or PID of a target process:

- ?PNAME returns the full process name or PID of either the calling process or another target process.
- ?GLIST returns the process's search list (?GLIST can also return the search list of the program file executing under that process.
- ?GUNM returns the username associated with a specific simple process name or PID.
- ?DADID returns the PID of a father process of either the calling process or of another target process.

Creating a Process

To create a process, and to specify its privileges and characteristics, use the ?PROC system call. The ?PROC system call allows you to create the process on either a local or remote host.

Once you have used the ?PROC system call to create a process, that process continues to exist until

- The process *traps*, or encounters a hardware error. (See the "Process Trapping" section in this chapter.)
- The process terminates voluntarily. (See the descriptions of the ?TERM and the ?RETURN system calls in The *System Call Dictionary (AOS/VS and AOS/DVS)*).
- Another process terminates the process. (See the description of the ?TERM system call in The *System Call Dictionary (AOS/VS and AOS/DVS)*).
- The process's father terminates.

Specifying Process Privileges

Within the ?PROC packet, you can specify a number of privileges for a newly created process; for example, the right to create sons and to assign those sons minimum and maximum working-set parameters, and the right to override the usual file access controls. However, you cannot assign the new process any privileges that the calling process does not have.

Table 3–4 lists the bit masks in offset ?PPRV of the ?PROC packet that define process privileges.

Table 3–4. Process Privileges

Privilege	Meaning
?PVPC	The new process can create an unlimited number of sons.
?PVWS	The new process can create sons of a different program file type (that is, 16–bit or 32–bit program files).
?PVEX	The new process can remain unblocked while one of its sons executes.
?PVWM	The new process can define working-set parameters for its sons.
?PVPR	The new process can use the ?PRIPR system call to change its own priority or to assign its sons higher priorities than its own.
?PVTY	The new process can use the ?CTYPE system call to change its process type or to create sons of any process type.
?PVIP	The new process can issue the ?ISEND and ?IS.R primitive IPC system calls. (See Chapter 7 for information on IPC system calls.)
?PVUI	The new process can create sons that have usernames different from its own.
?PVDV	The new process can define and access user devices. (See Chapter 10 for information on devices.)
?PVSP	The new process can issue the ?SUPROC system call to turn on Superprocess mode. (See the "Superuser Mode" and "Superprocess Mode" sections in this chapter.)
?PVSU	The new process can issue the ?SUSER system call to turn on Superuser mode. (See the "Superuser Mode" and "Superprocess Mode" sections in this chapter.)

Process Creation Parameters

AOS/VS determines the number of offspring a process can create by checking its ?PROC packet for

- The ?PVPC privilege, which specifies that the new process can create an unlimited number of sons processes.

  This privilege overrides every other creation parameter in the ?PROC packet. When a process that does not have the ?PVPC privilege tries to create a son, AOS/VS performs the following steps to check the other creation parameters:

  1. Checks to see if the number of sons and their combined ?PPCR count exceed the calling process's ?PPCR value. If yes, AOS/VS signals an error. If no, AOS/VS performs Step 2.
  2. Checks to see if bit ?PVEX is set. If yes, AOS/VS allows the calling process to create the son. If no, AOS/VS performs Step 3.
  3. Checks to see if the calling process has the ?PVEX privilege. If yes, AOS/VS allows the calling process to create the son. If no, AOS/VS does not allow the calling process to create the son.

- The ?PVEX privilege, which specifies that the new process can remain unblocked while one of its sons executes.
- The presence of offset ?PPCR, which specifies the maximum number of offspring.

  Offset ?PPCR is a cumulative value. That is, if a process with a ?PPCR value of 10 creates 2 sons, each with a ?PPCR value of 4, the original process cannot create any other sons, because 2 sons plus 2*4 (8 potential grandsons) equals 10.
- The presence of the ?PFEX mask (within offset ?PLFG), which determines whether the new process blocks while its sons execute.

You can use ?PROC system calls in your program if you want to create son processes, which, in turn, can create other sons. Figure 3–4 shows a process tree of this kind: process A created processes B, C, and D; process B created process F; and process D created processes G and E.

Overriding Default Restrictions: Superuser and Superprocess Mode

By default, a process can issue certain system calls only against its subordinate processes, and can use only those files for which it has the necessary access privileges. However, if the system manager has given you the Superuser or Superprocess privilege in your user profile, you can override these restrictions by putting a process into Superuser or Superprocess mode.

You should restrict the use of Superuser and Superprocess privileges, because a process in Superuser mode can delete any file, and a process in Superprocess mode can terminate any process.

Superuser Mode

A process that is in Superuser mode can access any file, regardless of the file's ACL, and can also determine the access privileges of any process to any file.

When you create a process, you can assign the Superuser privilege to a process by setting the ?PVSU mask in the ?PPRV offset of the ?PROC packet. The process can then issue the ?SUSER system call to turn on Superuser mode.

A process that has the Superuser privilege can pass the privilege on to its sons. These son processes can in turn issue the ?SUSER system call to turn on Superuser mode. Once a process has turned on Superuser mode, it remains in that state until it issues a second, complimentary ?SUSER system call to turn it off.

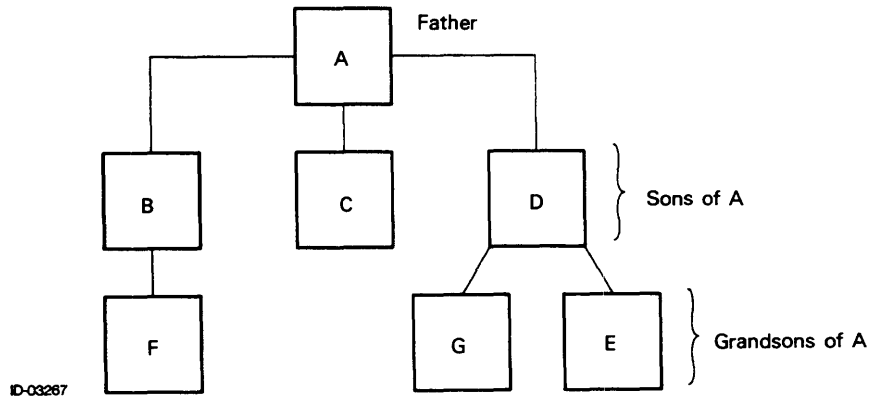


Figure 3–4. Sample Process Tree

Superprocess Mode

A process in Superprocess mode can change the state of *any* process, not just its subordinate processes, by issuing one of the following system calls:

?BLKPR Blocks a process.

?UBLPR Unblocks a process.

?BRKFL Terminates a process and creates a break file.

?CTYPE Changes a process's type.

?GTACP Gets a process's access control privileges.

?PRIPR Changes a process's priority.

?TERM Terminates a process.

(For more information on these system calls, see the individual system call descriptions in Chapter 15.)

When you create a process, you can assign the Superprocess privilege to that process by setting mask ?PVSP in offset ?PPRV of the ?PROC packet. You can then issue the ?SUPROC system call from that process to turn on Superprocess mode.

A process with the Superprocess privilege can also pass that privilege to its sons, although sons created with this privilege are not in Superprocess mode initially. A process remains in Superprocess mode until it issues a second. complementary ?SUPROC system call to turn off Superprocess mode.

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. (See the section, "Creating a Process", in this chapter for information on the process hierarchy.)
- 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. (See The *System Call Dictionary (AOS/VS and AOS/DVS)* for more information on the ?IREC and ?WDELAY system calls and see Chapter 7 for more information on tasks.)

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.

A resident process cannot be explicitly blocked.

Process Information

You can issue system calls that return information about the calling process, a target process, or about processes on remote hosts. These system calls can return the following information:

- Process name or PID number.
- PID numbers of son processes.
- Process's use of system resources.
- Class scheduling status.
- Local PID Information: maximum number of PIDs allowed, number of PIDs active, and so on.
- List of active PIDs on a remote host.

Process Name, PID Number, or Pathname

To get the process name or PID number of a process, issue the ?PNAME system call. Often, other system calls require this information as input.

To get the pathname of a process, issue the ?GLIST system call.

Use of System Resources

To get information on a process's use of system resources, you can issue the ?RUNTM, ?WHIST, or ?IHIST system call.

The ?RUNTM system call returns the following process information:

- The real time that has elapsed since process creation (in seconds, within the range 0 through (2**32)-1).
- The CPU time that the process used (in milliseconds).
- The number of blocks read or written.
- The page usage over a period of time (in page-seconds). AOS/VS calculates page-seconds by multiplying processor usage by main memory usage.

The ?WHIST system call generates a histogram for a 32–bit process (the ?IHIST system call generates a histogram for a 16–bit process). A histogram is a data array which provides a global view of processor activity. To issue the ?WHIST or ?IHIST system calls, a process must be resident. The calling process can activate only one histogram at a time. To terminate a histogram, the process must issue the ?KHIST system call.

Each histogram shows which address was being executed at each *tick* of the system's real-time clock. As a result, a histogram provides you with a pattern of activity for instructions.

AOS/VS updates the histogram statistics after each tick, or real-time clock pulse.

The ?WHIST system call does not zero out existing histograms in a data array. This allows you to stop a histogram and restart it without losing data; unless you want to aggregate data, you should explicitly reset the array to zero before you use it for another histogram.

The ?PSTAT system call returns internal statistics about a process and performance information about all programs that are currently executing.

Identity of Son Processes

The ?PSTAT system call also returns a list of those son processes that have PID numbers between 1 and 255. To return a list of all son processes, issue the ?SONS system call, which returns a list of *all* son processes.

Class Scheduling Status

To get the class ID and the user and program localities of a process, issue the ?PCLASS system call. The process for which you request this information can be either the calling process or target process.

Local PID Information

To get information on local PIDs, issue the ?PIDS system call. The ?PIDS system call returns

- The maximum number of PIDs that the system allows (as set in (the VSGEN dialog).
- The number of active PIDs currently on the system.
- The number of processes that currently have PIDs under 256.

Active Processes on a Remote Host

To get a list of the active processes on a remote host, issue the ?GPID system call.

Execute-Protection Status

To make it easier to find errors in your code, you may want to prevent your program from executing certain logical pages, such as pages that contain data. Therefore, AOS/VS provides execute protection. The ?EXPO system call allows you to set, clear, or examine a process's execute-protection status.

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 the write-protected shared area of its logical address space). The ?VALIDATE system call decreases the likelihood 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. (See Chapter 7 for more information on termination messages.)

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 TCBs) 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 the ?BRKFL system call.
- Type a CTRL–C CTRL–E sequence from the process terminal. (See Chapter 5 for a full description of terminal control characters and control sequences.)

To create a break file every time a process traps, set the ?PBRK bit in the ?PFLG offset 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 the program counter (PC)

Status Word	Contents
?BRTID	TID (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

The breakfile analysis program (BRAN) produces a report that displays information on the terminated process, such as the program type, memory use, the task that was active when the process terminated, and so on. For more information on running the Breakfile Analysis Program, see the BRAN entry in the *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS).*

Unless you specify another pathname, AOS/VS assigns the break file the default pathname

```
?pid.time.BRK
```

where

pid is the five-digit PID of the terminated process.

time is the time of the termination, in the form hours_minutes_seconds.

AOS/VS creates a break file only when the terminated process has Write or Append access to its working directory and the working directory has enough disk space for the break file.

Unlike the ?BRKFL system call, which terminates a process and creates a break file, the ?ENBRK system call 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 specify. The ?ENBRK system call allows AOS/VS to create a break file, it does not explicitly direct it to do so.

Transferring Process Control to the Debugger Utility

The ?DEBUG system call allows you to transfer control to the Debugger utility while your process is running. By including the ?DEBUG system call in your program, you can set up predefined breakpoints for testing purposes. Another way to call the Debugger utility is to choose the ?PFDB option in offset ?PFLG of the ?PROC packet.

You can use the ?DEBUG system call to examine or modify inner-ring user contexts. The user debugger does not base its protection logic upon the ring-maximization protection scheme. Instead, all access is based upon the ACLs of the inner-ring segment image.

To examine a user ring, the caller must have Read access to the segment image file. Also, the caller must have Write access to the segment image file to permit any modification (including setting breakpoints) of the user ring. (To set breakpoints in any user ring, you must always have Write access to Ring 7.)

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. The ?CHAIN system call transfers the following attributes to the new program:

- The username, process name, PID, terminal, 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 the 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:

1. Unloads all of the process's inner user rings.
2. Terminates all son processes that were previously created by ?PROC system calls issued from the inner user rings.
3. Breaks the connection, which, in turn, causes AOS/VS to revoke access privileges to protected shared files.

Loading Programs into Inner Rings

To load program files into a specific rings, 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 sytem call. (See Chapter 2 for more information on the ring structure.)

To cross from an outer ring to an inner ring, a program must have access to the proper ring 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 your gate entry points as .ENT (entry point). (See the GATE.ARRAY sample program in Chapter 5 for an example of using the .EXTG pseudo-op.) The *Principles of Operation ECLIPSE® 32–Bit Systems* manual explains how to reference gates and how to set up gate arrays.

Figure 3–5 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.

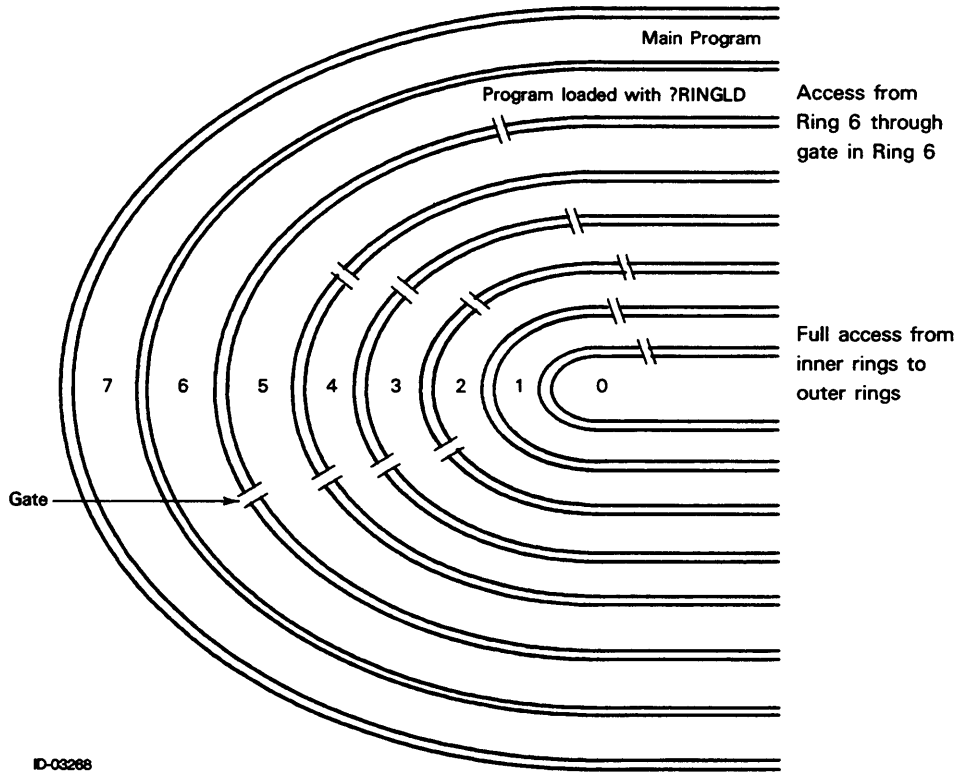


Figure 3–5. Ring Structure

Process and Memory Sample Programs

This chapter ends with examples that demonstrate the use of system calls to manage processes and memory.

Creating a Swappable Son Process: the SON Subroutine

The following subroutine, SON, creates a swappable son process. The son process runs program SPEAK.PR, which is an IPC sample program. (See Chapter 8.)

NOTE: To use the SON subroutine, you must have the Create Without Block privilege in your user profile.

```
                .TITLE  SON
                .ENT    SON
                .NREL                   ;Default partition 4.

;Get program name to ?PROC:
SON:    WSSVS   0                       ;Save return from XJSR.
        XLEFB   0,PRGNM*2               ;Byte pointer to the program
                                        ;name.
        XWSTA   0,PKT+?PSNM             ;Put in ?PROC packet.
        ?PROC   PKT                     ;Create process.
        WBR     ERROR                   ;Report error and quit.
        WRTN                            ;Return to caller.
```

```
PRGNM:  .TXT    "SPEAK.PR"

ERROR:  WLDAI   ?RFEC!?RFCF!?RFER,2     ;Error flags: Error code is in
                                        ;AC0 (?RFEC), message is in
;CLI format (?RFCF), and

                                        ;father should handle this as
                                        ;an error (?RFER).
        ?RETURN                         ;Return to CLI.
        WBR     ERROR                   ;Report error and quit.

;?PROC packet:

PKT:    .BLK    ?PLTH                   ;Allocate enough space for
                                        ;packet.
        .LOC    PKT+?PFLG
        .WORD   0                       ;Default process creation
                                        ;specifications. (See the
                                        ;description of ?PROC in
                                        ;Chapter 13.)

.LOC    PKT?PPRI
.WORD   -1                              ;Default priority of son
                                        ;process to same as father.

.LOC    PKT+?PSNM
.DWORD  PRGNM*2                         ;Byte pointer to pathname of
                                        ;program file for son to
                                        ;execute.

.LOC    PKT+?PIPC
.DWORD  -1                              ;No IPC message header to
                                        ;send to son (default is -1).

.LOC    PKT+?PNM
.DWORD  -1                              ;Default son's simple process
                                        ;name to ASCII representation
                                        ;of its PID.

.LOC    PKT+?PMEM
.DWORD  -1                              ;Default maximum number of
                                        ;son's logical pages to same
                                        ;as father.

.LOC    PKT+?PDIR
.DWORD  -1                              ;Default name of son's working

                                        ;directory to same as father

.LOC    PKT+?PCON
.DWORD  0                               ;Default name of son's
                                        ;@CONSOLE device to same as
                                        ;father
```

```
.LOC    PKT+?PCAL
.WORD   -1                      ;The default number of system
                                ;@lls son can issue
                                ;concurrently is two.

.LOC    PKT+?PWSS
.WORD   -1                      ;Default son's maximum working
                                ;set size to no limit.

.LOC    PKT+?PUNM               ;Byte pointer to son's
                                ;username.
.DWORD  -1                      ;Default son's username to
                                ;same as father.

.LOC    PKT+?PPRV               ;Son's privileges.
.WORD   ?PVIP                   ;Son an issue ?ISEND and
                                ;?IS.R.

.LOC    PKT+?PPCR
.WORD   0                       ;Son can create no sons.

.LOC    PKT+?PWMI
.WORD   -1                      ;Default son's minimum working
                                ;set size to no minimum.

.LOC    PKT+?PIPF
.DWORD                          ;Son has no @INPUT file.

.LOC    PKT+?POFP
.DWORD   0                      ;Son has no @OUTPUT file.

.LOC    PKT+?PLFP
.DWORD  0                       ;Son has no @LIST file.

.LOC     PKT+?PDFP
.DWORD  0                       ;Son has no @DATA file.

.LOC    PKT+?SMCH
.DWORD  -1                      ;Default maximum processor time
                                ;allotted for son to remainder
                                ;of father's time limit.

.LOC    PKT+?PLTH               ;End of packet.

.END    SON                     ;End of SON program.
```

Getting and Displaying Runtime Statistics: the RUNTIME Program

The following program, RUNTIME, gets its own runtime statistics and displays these statistics on the terminal.

First, RUNTIME opens the console, and then it issues the ?RUNTM system call, and finally, it converts the runtime statistics to ASCII decimal values and displays them on the terminal screen. Although RUNTIME gets its own runtime statistics, you can use it to get any process's runtime statistics by passing the process's filename.PR or the process's PID. To use RUNTIME as a subroutine, start with a proper save and end with a proper return.

```
                    .TITLE  RUNTIME
                    .ENT    RUNTIME,  CONVERT
                    .NREL

;Open terminal for I/O.

RUNTIME:?OPEN   CON                 ;Open terminal (CON) for I/O.
        WBR     ERROR               ;Report error and quit.

        ?WRITE  CON                 ;Display message on terminal.
        WBR     ERROR               ;Report error and quit.

;Call   ?RUNTM to get statistics.

LOOP:   WLDAI   -1,0                ;Check self.
        ?RUNTM  RPKT                ;Get statistics in RPKT.
        WBR     ERROR               ;Report error and quit.
        XWLDA   1,MSECS             ;Get time in milliseconds from
                                    ;RPKT.

        XLEFB   2,MSECMSG*2         ;Byte address of message that
                                    ;describes milliseconds
                                    ;elapsed.
        XJSR    CONVERT             ;Convert milliseconds elapsed
                                    ;to ASCII decimal and put
                                    ;converted value in
                                    ;milliseconds elapsed message.

        XLEFB   0,MSECMSG*2         ;Get byte pointer to
                                    ;milliseconds elapsed message.
        XWSTA   0,CON+?IBAD         ;Put milliseconds elapsed
                                    ;message in I/O packet.

        ?WRITE  CON                 ;Display milliseconds elapsed
                                    ;message on terminal.
        WBR     ERROR               ;Report error and quit.

        XWLDA   1,PSECS             ;Get page-seconds from RPKT.
        XLEFB   2,PSECMSG*2         ;Byte address of message that
                                    ;describes page-seconds
                                    ;elapsed.
```

```
        XJSR    CONVERT                 ;Convert page-seconds elapsed
                                        ;to ASCII decimal and put
                                        ;converted value in
                                        ;page-seconds elapsed message.
        XLEFB   0,PSECMSG*2             ;Get byte pointer to
                                        ;page-seconds elapsed message.

        XWSTA   0,CON+?IBAD             ;Put page-seconds elapsed
                                        ;message in I/O packet.

        ?WRITE  CON                     ;Display page-seconds elapsed
                                        ;message on terminal.
        WBR     ERROR                   ;Report error and quit.

;See if user wants to stop.

        XLEFB   0,BUF*2                 ;Get byte pointer to I/O
                                        ;buffer.
        XWSTA   0,CON+?IBAD             ;Put in I/O packet.

        ?READ   CON                     ;Look for terminator.
        WBR     ERROR                   ;Report error and quit.

        NLDAI   'ST',0                  ;Put ST in AC0.
        XNLDA   1,BUF                   ;Put first word of buffer in
                                        ;AC1.

        WSNE    0,1                     ;Skip next instruction if
                                        ;first word is not ST.
        WBR     BYE                     ;If first word is ST, go to
                                        ;BYE.
        WBR     LOOP                    ;If first word is not ST, do
                                        ;LOOP again.

;Error handler and return.

ERROR:  NLDAI   ?RFEC!?RFCF!?RFER,2     ;Error flags: Error code is
                                        ;in AC0 (?RFEC), message is in
                                        ;CLI format (?RFCF), and
                                        ;father should handle this
                                        ;as an error (?RFER).

BYE:    WSUB    2,2                     ;Good return flags.
        ?RETURN                         ;Return to father.
        WBR     ERROR                   ;?RETURN error return.

;Open and I/O packet for terminal.

CON:    .BLK    ?IBLT                   ;Allocate enough space for
                                        ;packet.

        .LOC    CON+?ISTI               ;File specifications.
        .WORD   ?ICRF!?RTDS!?OFIO       ;Change format to
                                        ;data-sensitive records and
                                        ;open for input and output.
```

```
        .LOC    CON+?IMRS
        .WORD   -1                      ;Default physical block size
                                        ;to 2 Kbytes.

        .LOC    CON+?IBAD
        .DWORD  ITEXT*2                 ;Set byte pointer to record
                                        ;I/O buffer.

        .LOC    CON+?IRCL
        .WORD   120.                    ;Record length is 120
                                        ;characters.

        .LOC    CON+?IFNP
        .DWORD  CONS*2                  ;Byte pointer to pathname.

        .LOC    CON+?IDEL               ;Delimiter table address.
        .DWORD  -1                      ;Use default delimiters: null,
                                        ;NEW LINE, form feed, and
                                        ;carriage return (default is
                                        ;-1).

        .LOC    CON+?IBLT               ;End of packet.

;Filename, start message, and buffer.  A .NOLOC 1follows.
CONS:   .TXT    "@CONSOLE"              ;Use generic name.

ITEXT:  .TXT    "I give runtime statistics on a process.
                 Type ST[NL] to return to father.<212><12>"
BUF:    .BLK    (BUF-CONS)*2)           ;Use number of bytes in
                                        ;message.
        .NOLOC  0                       ;Resume listing all.

;Messages to include converted statistics. .NOLOC here.

MSECMSG:.TXT    "       milliseconds elapsed.<12>"
PSECMSG:.TXT    "       page-seconds elapsed.  Type
       ST[NL]  to stop, type another character to loop.<212><12>"
        .NOLOC  0

;?RUNTM packet.

RPKT:   .BLK    ?GRLTH                  ;Allocate enough space for
                                        ;packet.
SECS:   .LOC    RPKT+?GRRH
        .DWORD  0                       ;AOS/VS returns elapsed time
                                        ;in seconds.

MSECS:  .LOC    RPKT+?GRCH
        .DWORD  0                       ;AOS/VS returns elapsed Processor
                                        ;time in milliseconds.
```

```
;See if user wants to stop.

        XLEFB   0,BUF*2                 ;Get byte pointer to I/O
                                        ;buffer.
        XWSTA   0,CON+?IBAD             ;Put in I/O packet.

        ?READ   CON                     ;Look for terminator.
        WBR     ERROR                   ;Report error and quit.

        NLDAI   'ST',0                  ;Put ST in AC0.
        XNLDA   1,BUF                   ;Put first word of buffer in
                                        ;AC1.

        WSNE     0,1                    ;Skip next if first word is
                                        ;not ST.

        WBR     BYE                     ;If first word is ST, go to
                                        ;BYE.
        WBR     LOOP                    ;If first word is not ST, do
                                        ;LOOP again.

;Error handler and return.

ERROR:  NLDAI   ?RFEC!?RFCF!?RFER,2     ;Error flags: Error code is
                                        ;in AC0 (?RFEC), message is in

                                        ;CLI format (?RFCF), and
                                        ;father should handle this
                                        ;as an error (?RFER).

BYE:    WSUB    2,2                     ;Good return flags.
        ?RETURN                         ;Return to father.
        WBR     ERROR                   ;?RETURN error return.

;Open and I/O packet for terminal.

CON:    .BLK    ?IBLT                   ;Allocate enough space for
                                        ;packet.

        .LOC    CON+?ISTI               ;File specifications.
        .WORD   ?ICRF!?RTDS!?OFIO       ;Change format to data-
                                        ;sensitive records and open
                                        ;for input and output.

        .LOC    CON+?IMRS
        .WORD   -1                      ;Default physical block size
                                        ;to 2 Kbytes.

        .LOC    CON+?IBAD
        .DWORD  ITEXT*2                 ;Byte pointer to record I/O
                                        ;buffer.

        .LOC    CON+?IRCL
        .WORD   120.                    ;Record length is 120
                                        ;characters.
```

```
        .LOC    CON+?IFNP
        .DWORD  CONS*2                  ;Byte pointer to pathname.

        .LOC    CON+?IDEL               ;Delimiter table address.
        .DWORD  -1                      ;Use default delimiters: null,
                                        ;NEW LINE, form feed, and
                                        ;carriage return (default is
                                        ;-1).

        .LOC    CON+?IBLT               ;End of packet.

;Filename, start message, and buffer.  A NOLOC 1 follows.
CONS:   .TXT    "@CONSOLE"              ;Use generic name.

ITEXT:  .TXT    "I give runtime statistics on a process.
                 Type ST[NL] to return to father.<212><12>"
BUF:    .BLK    (BUF-CONS)*2            ;Use number of bytes in
                                        ;message.
        .NOLOC  0                       ;Resume listing all.

;Messages to include converted statistics. .NOLOC here.

MSECMSG:.TXT    "       milliseconds elapsed.<12>"
PSECMSG:.TXT    "       page-seconds elapsed.  Type
        ST[NL] to stop, type another character to loop.<212><12>"
        .NOLOC  0

;?RUNTM packet.

RPKT:   .BLK    ?GRLTH                  ;Allocate enough space for
                                        ;packet.

SECS:   .LOC    RPKT+?GRRH
        .DWORD  0                       ;AOS/VS returns elapsed time
                                        ;in seconds.

MSECS:  .LOC    RPKT+?GRCH
        .DWORD  0                       ;AOS/VS returns elapsed Processor
                                        ;time in milliseconds.

IO:     .LOC    RPKT+?GRIH
        .DWORD  0                       ;AOS/VS returns number of
                                        ;blocks read or written.
PSECS:  .LOC    RPKT+?GRPH              ;Page usage over elapsed Processor
                                        ;time.
        .DWORD  0                       ;AOS/VS returns page usage
                                        ;over elapsed Processor time in
                                        ;page-seconds.

        .LOC    CON+?GRLTH              ;End of packet.
```

```
;CONVERT routine converts binary value into its decimal equivalent and
;puts it in a text string. AC1 contains the value and AC2 contains
;the byte address of the text message.

CONVERT:WSSVS   0                       ;Save return.
        WMOV    2,3                     ;Use AC3 to shift byte pointer.

        WADI    3,3                     ;Add integer 3 to byte address.
        NLDAI   10.,2                   ;Put 10 in AC2

DLOOP:  WSUB    0,0                     ;Zero AC0 (high-order portion
                                        ;of dividend). AC1 still
                                        ;contains low-order portion of
                                        ;dividend.
        WDIVS                           ;Divide by 10, put quotient in
                                        ;AC1, and put remainder in AC0.

        IORI    60,0                    ;OR in 60 for ASCII number.
        WSTB    3,0                     ;Store AC0 byte in byte
                                        ;address of AC3.

        WSBI    1,3                     ;Decrement byte address.
        MOV     1,1,SNR                 ;Is quotient 0?
        WRTN                            ;If quotient is 0, return to
                                        ;caller.
        WBR     DLOOP                   ;If quotient is not 0, do
                                        ;another digit.

        .END    RUNTIME                 ;End of RUNTIME program.
```

Loading a Program into an Inner Ring: the RINGLOAD Program

The following program, RINGLOAD, loads program INRING into an inner ring. Then, RINGLOAD uses an LCALL instruction to call INRING. RINGLOAD assumes that file INRING.PR, which was linked from INRING and GATE.ARRAY exists. (GATE.ARRAY is at the end of this section.)

```
                    .TITLE  RINGLOAD
                    .ENT    RINGLOAD
                    .NREL

;Open terminal for I/O and issue ?RINGLD to load program into inner
;ring.

RINGLOAD: ?OPEN CON                     ;Open CON (terminal) for I/O.
        WBR     ERROR                   ;?OPEN error return.

        ?WRITE  CON                     ;Display message on terminal.
        WBR     ERROR                   ;?WRITE error return.

        XLEFB   0,PNAME*2               ;Get byte pointer to INRING
                                        ;name.

        ?RINGLD                         ;Load INRING.
        WBR     ERROR                   ;?RINGLD error return.

        LCALL   INRING,0,0              ;Call INRING and set the index
                                        ;and argument count to 0.
        WBR     INERROR                 ;Report INRING error.

;Back from INRING. Depart with message for CLI.

        XLEFB   0,MES2*2                ;Get byte pointer to farewell
                                        ;message.

        XWSTA   0,CON+?IBAD             ;Put in I/O packet.

        ?WRITE  CON                     ;Display message on terminal.
        WBR     ERROR                   ;?WRITE error return.

        WSUB    2,2                     ;Set return flags for normal
                                        ;return.
        WBR     BYE                     ;Done. Give message and
                                        ;depart.

;Inner-ring program and current program error handlers.

INERROR: LLDFB  0,INMES                 ;Get inner-ring program error
                                        ;message.
        LWSTA   0,CON+?IBAD             ;Put in I/O packet.

        ?WRITE  CON                     ;Display message on terminal.
        WBR     ERROR                   ;?WRITE error message.
```

```
ERROR:  WLDAI   ?RFEC!?RFCF!?RFER,2      ;Error flags: Error code is in
                                         ;ACO (?RFEC), message is in
                                         ;CLI format (?RFCF), and
                                         ;father should handle this as
                                         ;an error (?RFER).

BYE:    ?RETURN                          ;Return to CLI.
        WBR     ERROR                    ;?RETURN error return.

;Definition of inner-ring program ring bracket and gate.  AOS/VS uses
;this, instead of the program name (INRING) to access gate and
;inner-ring program messages.  A .NOLOC 1 follows.

        INRING = 5S3+0                   ;Ring 5 + first gate.

PNAME:  .TXT    "INRING.PR"              ;Program name is INRING.

MES1:   .TXT    "I'm RINGLOAD.  I am about to ?RINGLOAD program
                 INRING.<12>"

MES2:   .TXT    "<212>I'm RINGLOAD.  I'm back from INRING, and I'm
                 terminating.<12>"

INMES:  .TXT    "<212>ERROR IN INNER-RING PROGRAM.<12>"
        .NOLOC  0                        ;Resume listing all.

;Open and I/O packet. (You need this packet for I/O.)

CON:    .BLK    ?IBLT                    ;Allocate enough space for
                                         ;packet.

        .LOC    CON+?ISTI                ;File specifications.
        .WORD   ?ICRF!?RTDS!?OFIO        ;Change format to
                                         ;data-sensitive records and
                                         ;open for input and output.

        .LOC    CON+?IMRS
        .WORD   -1                       ;Default physical block size
                                         ;to 2 Kbytes.

        .LOC    CON+?IBAD
        .DWORD  MES1*2                   ;Byte pointer to record I/O
                                         ;buffer.

        .LOC    CON+?IRCL
        .WORD   120.                     ;Record length is 120.
                                         ;characters.

        .LOC    CONS+?IFNP
        .DWORD  CONS*2                   ;Byte pointer to pathname.
```

```
        .LOC    CON+?IDEL               ;Delimiter table address.
        .DWORD  -1                      ;Use default delimiters: null,
                                        ;NEW LINE, form feed, and
                                        ;marriage return (default is
                                        ;-1).

        .LOC    CON+?IBLT               ;End of packet.

CONS:   .TXT    "@CONSOLE"              ;Use generic name.

        .END    RINGLOAD                ;End of RINGLOAD program.
```

The INRING Program

Program RINGLOAD loads the following program, INRING, into Ring 5. Then, RINGLOAD uses an LCALL instruction to call INRING.

INRING saves the return address, opens the terminal, writes messages, and invokes the Debugger (so you can explore the inner ring). To return to RINGLOAD in Ring 7, type ESC R.

Except for the I/O packet, all code in INRING is shared.

You must link INRING with GATE.ARRAY (the last program in this section). Depending on the LCALL name definition in GATE.ARRAY, the link that the gate defines in GATE.ARRAY, and the Link switch that you use, you can execute INRING in Rings 4, 5, or 6. In this case, RINGLOAD defined the LCALL name as Gate 5 (5S3), GATE.ARRAY defined Gate 5, and the Link command line was X LINK/RING=5 INRING GATE.ARRAY.

```
                .TITLE  INRING
                .ENT    INRING
                .NREL   1
                   .EXTL   GATE.ARRAY

;Define a two-word pointer to the gate array.

        .LOC    34                      ;Locations 34 and 35.
        .DWORD  GATE.ARRAY              ;Pointer.

;Save the return, open the terminal, write the message, and enter the
;debugger.

INRING: WSAVR   0                       ;Save frame (ACs, PC in AC3).
        ?OPEN   CON                     ;Open terminal (CON) for I/O.
        WBR     ERTN                    ;?OPEN error return.

        ?WRITE  CON                     ;Display message from Ring 5
                                        ;on terminal.
        WBR     ERTN                    ;Report error and quit.

        ?DEBUG                          ;Enter debugger.
        WBR     ERTN                    ;Report error and quit.

        LLEFB   0,MES2*2                ;Get byte pointer to return
                                        ;message.
        LWSTA   0,CON+?IBAD             ;Put in I/O packet.
```

```
        ?WRITE  CON                     ;Display another message on
                                        ;terminal.
        WBR     ERTN                    ;?WRITE error return.

;Done.  Ready for good return to caller.

        LDAFP   3                       ;Get frame pointer in AC3.
        XWISZ   0,3                     ;Increment return address for
                                        ;normal return to process
                                        ;issuing ?LCALL.
        WRTN                            ;Return to caller.

;INRING error handler.  Returns error to outer-ring caller, not CLI.

ERTN:   LDAFP   3                       ;Get frame pointer in AC3.
        LWSTA   0,?OACO,3               ;Put error code (ACO) in saved
                                        ;frame's ACO.
        WRTN                            ;Return to LCALL process's
                                        ;error return.

;Text messages.  .NOLOC 1 follows.

MES1:   .TXT    "I'm INRING. I'm in the inner ring. I'm about to debug.
                 Type ESC R to proceed.<212><12>"

MES2:   .TXT    "<212><212>From INRING.  I'm about to WRTN.<12>"

        .NOLOC  0                       ;Resume listing all.

        .NREL                           ;Use unshared code for packet,
                                        ;because program and AOS/VS
                                        ;write into it.

;Open I/O packet for @CONSOLE.

CON:    .BLK    ?IBLT                   ;Allocate enough space for
                                        ;packet.

        .LOC    CON+?ISTI               ;File specifications.
        .WORD   ?ICRF!?RTDS!?IFIO       ;Change format to data-
                                        ;sensitive records and open
                                        ;for input and output.

        .LOC    CON+?IMRS
        .WORD   -1                      ;Default physical block size
                                        ;to 2 Kbytes.

        .LOC CON+?IBAD
        .DWORD  MES1*2                  ;Byte pointer to record I/O
                                        ;buffer.

        .LOC    CON+?IRCL
        .WORD   120.                    ;Record length is 120
                                        ;characters.
```

```
        .LOC    CON+?IFNP
        .DWORD  CONS*2                  ;Byte pointer to pathname.

        .LOC    CON+?IDEL               ;Delimiter table address.
        .DWORD  -1                      ;Use default delimiters: null,
                                        ;NEW LINE, form feed, and
                                        ;marriage return (default is
                                        ;-1).
```

The GATE.ARRAY Program

The following program, GATE.ARRAY, defines the gate array. Generally, you must define the gate array in a separate module. A gate array module must contain .EXTG PROG-ENTRY-NAME, where PROG-ENTRY-NAME is the start entry name in the program that will be accessed through the gate. Also, you must link the gate array module with the inner-ring program. In this case, the Link command line is

```
X LINK/RING=5 INRING GATE.ARRAY
```

```
                        .TITLE  GATE.ARRAY
                        .EXTG   INRING
                        .ENT    GATE.ARRAY

        .NREL   1                       ;Shared code for general use.
        .ENABLE ABS

GATE.ARRAY:     .DWORD  1               ;Gate array, one gate.
                .DWORD  (RING7-         ;LINK will determine the
RING5)!INRING
                                        ;address. A program in any
                                        ;ring can access the gate.

        RING7 = 7S3                     ;Bits 1 3 specify Gate 7.
        RING5 = 5S3                     ;Bits 1 3 specify Gate 5.

        .END    GATE.ARRAY              ;End of GATE.ARRAY program.
```

End of Chapter

Chapter 4
Creating and Managing Files

You can use the following system calls to create and manage files:

?CGNAM	Get a complete pathname from a channel number.
?CPMAX	Set maximum size for a control point directory (CPD).
?CREATE	Create a file or directory.
?DACL	Set, clear, or examine a default access control list.
?DELETE	Delete a file entry.
?DIR	Change the working directory.
?FSTAT	Get file status information.
?GACL	Get a file entry's access control list.
?GLINK	Get the contents of a link entry.
?GLIST	Get the contents of a search list.
?GNAME	Get a complete pathname.
?GRNAME	Return the complete pathname of a generic file.
?GNFN	List a particular directory's entries.
?GTACP	Get access control privileges for specific file and username.
?INIT	Initialize a logical disk.
?RECREATE	Recreate a file.
?RELEASE	Release an initialized logical disk (LD).
?RENAME	Rename a file.
?SACL	Set a new access control list.
?SATR	Set or remove attributes for a file or directory.
?SLIST	Set the search list.

This chapter describes the AOS/VS file structure and the system calls that you use to create and maintain files and directories.

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 filenames.

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 terminals, you can use strictly for file I/O.

Disk File Structures

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 4 (four disk blocks per element), or whatever file-element size you selected during the system-generation procedure. (Refer to the *How to Generate and Run AOS/VS* manual for more information on the system-generation procedure.) You can also specify a file-element size when you create a file.

AOS/VS always rounds file-element sizes to the next higher multiple of the default file-element size. For example, if you create a file with a file-element size of 5 and the default file-element size is 4, AOS/VS rounds the file-element size to 8.

AOS/VS allocates disk space to a file based on its file-element size. For example, a file with a file-element size of 4 *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 4 may consist of a number of *scattered* four-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 level. Figure 4–1 shows typical growth stages for a file with a file-element size of 4.

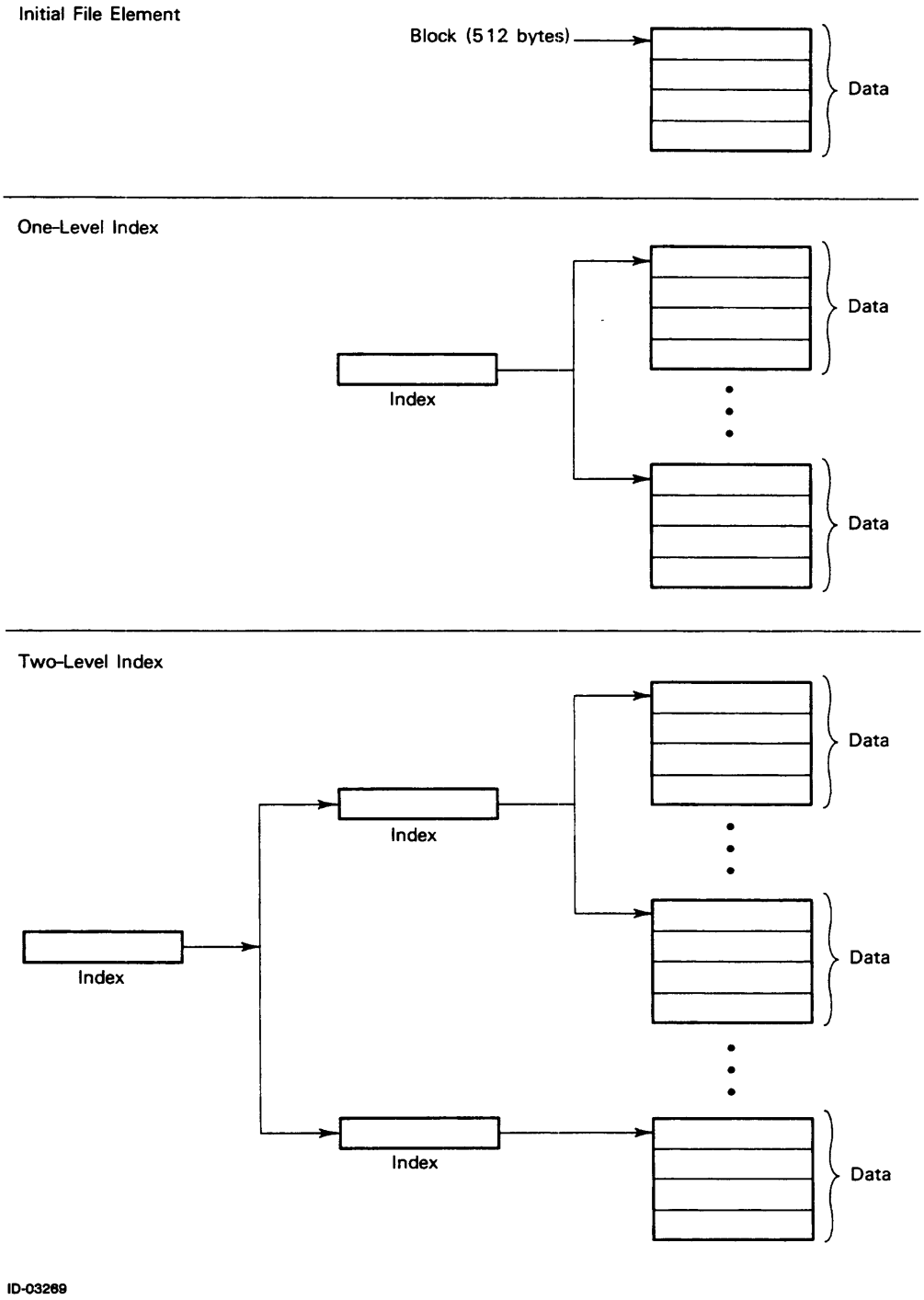


Figure 4–1. File Growth Stages

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 8 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/I source files, or a directory called UPD to contain all user profiles. The AOS/VS filename conventions also apply to directory names.

AOS/VS organizes directories into a hierarchical tree structure similar to the process tree structure. (See Figure 4–2.) The initial directory, called the root, is superior to all others in the hierarchy. A colon (:) represents the root.

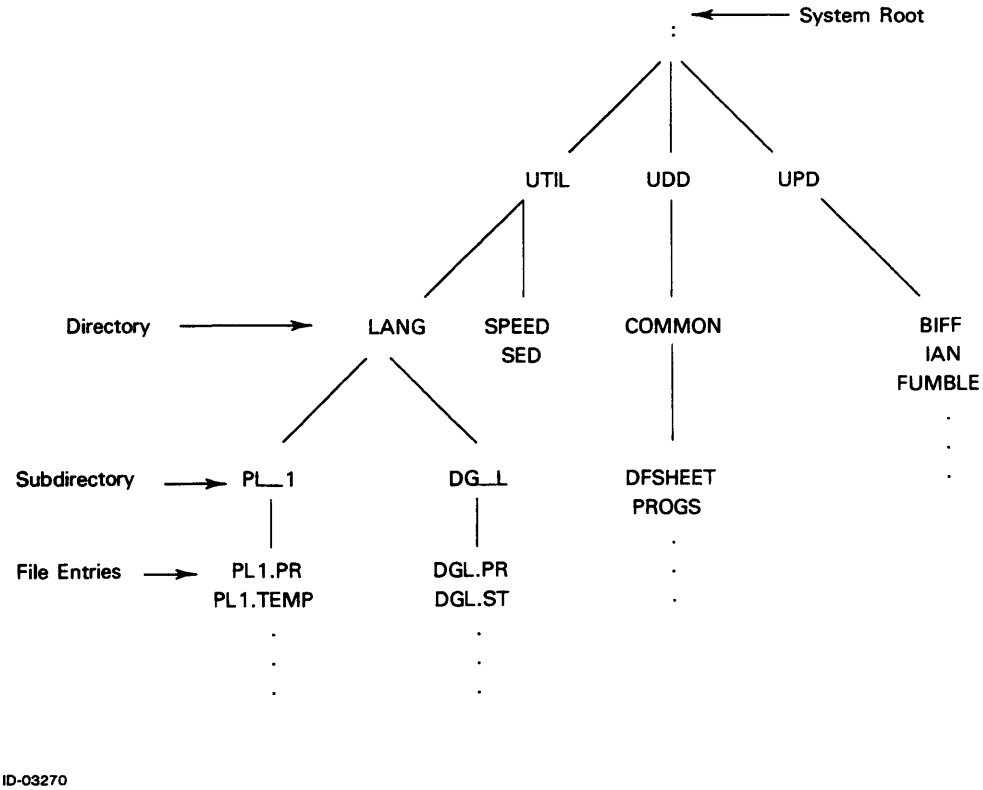


Figure 4–2. Sample Directory Tree

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 process that created the file and the file's local port number. AOS/VS recognizes 256 different types of directory entries, numbered from 0 through 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.

File Types

A file's characteristics and function determine its file type. Table 4–1 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 4-1 indicates, there are three types of program files:

- ?FPRV files, which are developed under AOS/VS.
- ?FPRG files, which are developed under AOS.
- ?FUN files, which are developed under DG/UX™ and MV/UX™ operating systems.

Table 4–1. File Types

Mnemonic	Type	Special Attributes/Restrictions
?FUDF	User Data File	Usually applies to source or object files.
?FTXT	Text File	Should contain ASCII text.
?FPIP	Pipe File	A transient file for use in transferring data between files.
?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 DG/UX and MV/UX operating systems.
?FDIR	Disk Directory	None.
?FCPD	Control Point Directory	(See the "Disk Space Control" section 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 user profile editor (PREDITOR) and EXEC.
?FSDF	System Data File	None.

Table 4-1. File Types

Mnemonic	Type	Special Attributes/Restrictions
?FIPC	IPC Port Entry	(See Chapter 8.)
?FMTF	Magnetic Tape File	None.
?FGFN	Generic Filename	Refers to the generic filenames; that is, @OUTPUT, @LIST, @DATA, etc.
?FGLT	Generic Labeled Tape	None.
?FGLM	Generic Labeled Media	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.)
?FMCU	Multiprocessor Communications Unit	Cannot create with the ?CREATE system call.
?FMTU	Magnetic Tape Unit	Device for accessing 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 the ?CREATE system call.
?FPLA	Plotter	Cannot create with the ?CREATE system call.
?FCON	Terminal (console), hard copy or video display.	Cannot create with the ?CREATE system call.
?FSYN	Synchronous Communications Line	Cannot create with the ?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 reassemble or recompile an AOS program file to execute it under AOS/VS.) If you try to execute an ?FPRG program file under AOS/VS, it returns the ERIFT (illegal file type) error code.

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 access 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, or as many as 31, ASCII characters. The legal filename characters are

- Uppercase and lowercase alphabetic characters (a–z, A–Z).
- Numerals 0 through 9.
- Period (.).
- Dollar sign ($).
- Question mark (?).
- Underscore (_).

AOS/VS treats uppercase and lowercase alphabetic characters as the same.

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 4–2 lists the filename conventions used by AOS/VS and its utilities.

Table 4–2. Filename Conventions

File	Filenames End In
Assembly language source files	.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 or compile them to produce object files. One or more linked object modules and/or library files make up an executable 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 subordinate to directory UDD, which is subordinate to the system root, represented by the colon (:).

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 4–3 lists the valid pathname prefixes.

Table 4–3. Valid Pathname Prefixes

Prefix	Meaning
:	Start at the system root directory.
=	Start at the current working directory.
^	(Caret or SHIFT–6) Move up to the immediately superior directory. (You can use more than one caret in a pathname.)
@	Start at the peripheral directory (:PER).

The peripheral directory (:PER), which is subordinate to the root, contains the names of generic filenames that refer to classes of I/O devices, and the names of system devices. (See Chapter 5 for more information on generic filenames and the peripheral directory.)

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 both UDD 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 4–3 illustrates the use of pathname strings for a sample directory structure.

Many system calls require pathnames as arguments. When you give a pathname as an argument, you must terminate the pathname with a null <000> byte (unless otherwise indicated in the *AOS/VS and DVS System Call Dictionary*).

Similarly, when the system passes pathnames to your programs, it too terminates the pathname with a null byte. Consequently, when you use a system call that returns a pathname (or a simple filename), you must allow sufficient buffer space to hold both the pathname *and* the null byte that terminates it.

The ?GNAME and ?CGNAM system calls both return a file's complete pathname, starting with the root. However, 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. (We describe channels in Chapter 5.)

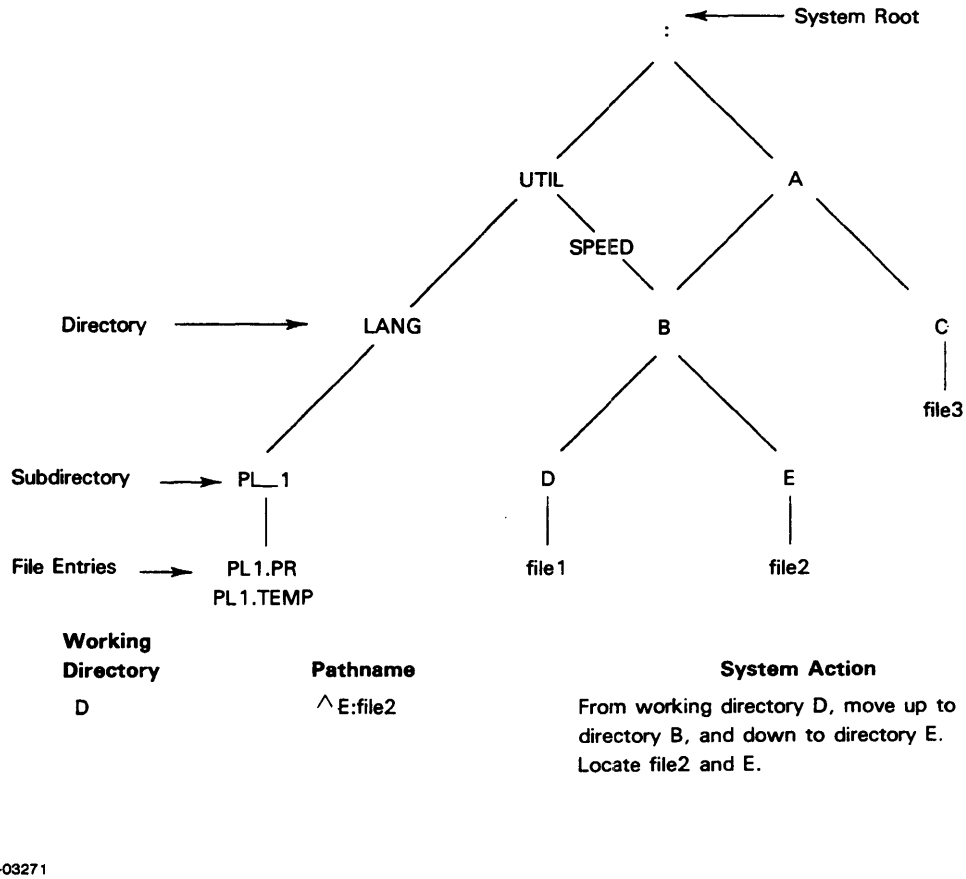


Figure 4–3. Directory Structure

Assuming that the directory structure is the one shown in Figure 4–3, 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. (See Chapter 5 for more information on generic files.)

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 4–3, 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.

NOTE: Link entries work differently as input to the system calls ?CREATE and ?DELETE. The next section discusses these 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 the 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 in 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 want 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 4–4 lists the access privileges and their meaning for directories and all other file types.

Table 4–4. File Access Privileges

Privilege	Nondirectory Files —Specific Privileges	Directories —Specific Privileges
Owner Access	• Read and change the file's ACL. • Read the filestatus and permanence attribute of the file. • Set the permanence attribute of the file.	• Read and change the ACL of the directory. • Initialize an LDU if you have owner access to the LDU's root directory. Rename or delete the directory.
	• Get a complete pathname of the file. • Rename or delete the file. • Create a user data area (UDA) for the file and read or write to it.	
Write Access	• Modify the data in the file. • Read the filestatus and permanence attribute of the file • Get a complete pathname of the file. • Create a User Data Area (UDA) for this file and write to it.	• Create, delete, and rename the directory's files. • Read and change each file's ACL. • Read and set the permanence attribute of the directory's files. • Initialize and release an LDU in the directory.
Append Access	• Read the filestatus and permanence attribute of the file. • Get a complete pathname of the file.	• Add files to the directory. • Initialize an LDU in the directory.

(continues)

Table 4-4. File Access Privileges

Privilege	Nondirectory Files —Specific Privileges	Directories —Specific Priveleges
Read Access		
	• Examine the data in the file	• List the name, filestatus, and permanence attribute of each file in the directory.
	• Read the filestatus and permanence attribute of the file.	• Use this directory as a working directory.
	• Get a complete pathname of the file.	• Read each file's ACL.
	• Read a user data area for the file.	• Get the contents of a link entry in the directory.
Execute Access		
	• Execute the file.	• Name the directory in a pathname (this is essential if you want to name the directory or refer to it.)
	• Read the filestatus and permanence attribute of the file.	
	• Get a complete pathname of this file.	• Make the directory your working directory.
		• Resolve a pathname using a link in the directory.

(concluded)

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 the "Logical Disks" section 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. For more information about reading and writing to files, see Chapter 5.

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 4–5 lists the valid ACL templates and the character combinations that they represent.

Table 4-5. Valid ACL Templates

Template	Meaning
+	Matches any character string. For example, the ACL username specification PA+ matches any character string that begins with PA. (For example, it matches 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, and PA_B, but not PA.M.)
*	Matches any single character except 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. Consequently, 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 Permanence Attribute

Any user with Owner access can easily delete a directory or file. Therefore, AOS/VS provides the permanence attribute for additional protection.

The permanence attribute prevents users from deleting a directory or file, regardless of its ACL. The ?SATR system call sets the permanence 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 permanence attribute.

If you set the permanence 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 resides entirely within a single LD.

Each LD is a collection of disk space that contains a directory tree structure. Each LD has a single directory called the local root, which is the foundation for constructing this 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 (:).

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.
- Execute access to each disk unit in the LD.
- Write or Append access to the target directory.

The ?INIT system call grafts the LD's local root to a specified directory (see Figure 4–4).

The disk structure within each LD can have up to eight directory levels, excluding the local root (directory level zero within that LD). You can graft an LD into any directory in another LD, with the exception of the lowest directory level (level 8). The maximum directory level attainable is limited only by the maximum length of a pathname under AOS/VS, which is 256 characters.

Releasing a Logical Disk

An LD remains initialized until you release it. If you want to remove an LD's component volumes from the disk drives so that you could mount other volumes, you would have to release that LD. To release an LD, you issue the ?RELEASE system call.

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 the following two additional variables:

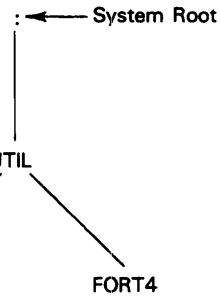
- Current space (CS), which is the amount of space currently allocated.
- Maximum space (MS), which is the maximum amount of space available in the directory. CS is the current number of disk blocks occupied by the CPD and its subordinate files, except for files in a subordinate LD. When you create a CPD, AOS/VS initializes CS to 0. MS is the maximum number of disk blocks available to the CPD and all its subordinate files, except for files in an subordinate 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.

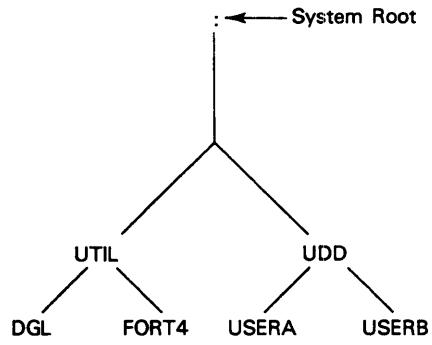
Figure 4–5 shows a simple directory structure with two CPDs.

Assume that the LD root and directory CP1 in Figure 4–5 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.



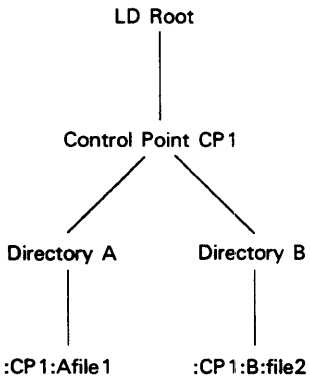
Assume that the LD to be initialized is LD ALPHA. ALPHA's local root, directory UDD, contains two inferior directories: USERS, and USERB. If you issue the ?INIT system call for ALPHA, and you specify 0 in AC1, AOS/VS grafts ALPHA to the system root, and the directory tree becomes

Master LD (after ?INIT)



ID-03272

Figure 4–4. Initializing a Logical Disk



ID-03273

Figure 4–5. Control Point Directories (CPD s)

When you create a CPD, AOS/VS does not initially check its MS against those of the other CPDs in the directory structure. In fact, AOS/VS permits oversubscription, as long as the directory structure'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 Creation and Management Sample Programs

The following program, FILCREA, opens the terminal and asks you for the name of the file you want to create. Then, if the file already exists, FILCREA deletes the file and recreates it for you.

```
                    .TITLE  FILCREA
                    .ENT    FILCREA
                    .NREL

FILCREA:?OPEN   CON                 ;Open CON (terminal) for I/O.
        WBR     ERROR               ;Error out.

        ?WRITE  CON                 ;Write message.
        WBR     ERROR               ;Quit.
        XLEFB   0,BUF*2             ;Get byte pointer to buffer.
        XWSTA   0,CON+?IBAD         ;Put in I/O packet.

        ?READ   CON                 ;Read filename.
        WBR     ERROR               ;Quit.

CREATE: ?CREATE CPKT                ;Create file (AC0 still
                                    ;contains byte pointer to
                                    ;filename.)
        WBR     TEST                ;Try to handle the error.

        ?WRITE  CON                 ;Echo the filename.
        WBR     ERROR               ;Quit.

        XLEFB   0,TMES*2            ;Get byte pointer to
                                    ;confirmation message.
        XWSTA   0,CON+?IBAD         ;Put in I/O packet.
        ?WRITE  CON                 ;Display confirmation message
                                    ;on terminal.
        WBR     ERROR               ;Quit.

        WSUB    2,2                 ;Good return flags.
        ?RETURN                     ;Return to the CLI.
        WBR     ERROR               ;?RETURN error return.


;Here we deal with errors from ?CREATE.

TEST:   WLDAI   ERNAE,2             ;Is the error code
        WSEQ    2,0                 ;"filename already exists"?
        WBR     ERROR               ;No. Report error and quit.

;File already exists. Delete it and start again.
```

```
        XLEFB   0,BUF*2                 ;Get byte pointer to buffer.
        ?DELETE                         ;Delete file.
        WBR     ERROR                   ;NOW what?
        WBR     CREATE                  ;Resume processing.


;All errors except those from ?CREATE come here.  We just return with
;an error code.

ERROR:  WLDAI   ?RFEC!?RFCF!?RFER       ;Error flags: Error code is
                                        ;in ACO (?RFEC), message is
                                        ;in CLI format (?RFCF), and
                                        ;caller should handle this as
                                        ;an error (?RFER).
        ?RETURN                         ;Return to the CLI.
        WBR     ERROR                   ;?RETURN error return.

;?CREATE packet.

CPKT:   .BLK    ?CLTH                   ;Allocate enough space for
                                        ;packet.

        .LOC    CPKT+?CFTYP             ;Record type in left byte and
        .WORD   ?ORDS*400!?FUDF         ;data type in right byte.

        .LOC    CPKT+?CCPS              ;File control parameters.
        .WORD   0                       ;Ignore.

        .LOC    CPKT+?CTIM              ;Address of time block.
        .DWORD  -1                      ;Set all values to current
                                        ;time (default is -1).

        .LOC    CPKT+?CACP              ;Set up byte pointer to ACL.
        .DWORD  ACL*2

        .LOC    CPKT+?CDEH              ;Reserved
        .WORD   0                       ;Set to 0.

        .LOC    CPKT+?CDEL              ;File element size.
        .WORD   -1                      ;Set to default.

        .LOC    CPKT+?CMIL              ;Maximum number of index
        .WORD   -1                      ;levels. Default.

        .LOC    CPKT+?CLTH              ;End of packet.

ACL:    .TXT "Username<0><?FACO!?FACW!?FACR><0>"        ;Set ACL to OWR.

;File already exists. Delete it and start again.

        XLEFB   0,BUF*2                 ;Get byte pointer to buffer.
        ?DELETE                         ;Delete file.
        WBR     ERROR                   ;NOW what?
        WBR     CREATE                  ;Resume processing.
```

```
;All errors except those from ?CREATE come here.  We just return with
;an error code.

ERROR:  WLDAI   ?RFEC!?RFCF!?RFER       ;Error flags: Error code is
                                        ;in AC0 (?RFEC), message is
                                        ;in CLI format (?RFCF), and
                                        ;caller should handle this as
                                        ;an error (?RFER).
        ?RETURN                         ;Return to the CLI.
        WBR     ERROR                   ;?RETURN error return.

;?CREATE packet.

CPKT:   .BLK    ?CLTH                   ;Allocate enough space for
                                        ;packet.

        .LOC    CPKT+?CFTYP             ;Record type in left byte and
        .WORD   ?ORDS*400!?FUDF         ;data type in right byte.

        .LOC    CPKT+?CCPS              ;File control parameters.
        .WORD   0                       ;Ignore.

        .LOC    CPKT+?CTIM              ;Address of time block.
        .DWORD  -1                      ;Set all values to current
                                        ;time (default is -1).

        .LOC    CPKT+?CACP              ;Set up byte pointer to ACL.
        .DWORD  ACL*2

        .LOC    CPKT+?CDEH              ;Reserved
        .WORD   0                       ;Set to 0.

        .LOC    CPKT+?CDEL              ;File element size.
        .WORD   -1                      ;Set to default.

        .LOC    CPKT+?CMIL              ;Maximum number of index
        .WORD   -1                      ;levels. Default.

        .LOC    CPKT+?CLTH              ;End of packet.

ACL:    .TXT "Username<0><?FAC0!?FACW!?FACR><0>"        ;Set ACL to OWR.

;Open an I/O packet for the terminal.

CON:    .BLK    ?IBLT                   ;Allocate enough space for
                                        ;packet.

        .LOC    CON+?ISTI               ;File specifications.
        .WORD   ?ICRF!?RTDS!?OFIO       ;Change format to data-sensitive
                                        ;records and open for input and
                                        ;output.

        .LOC    CON+?IMRS               ;Physical block size (in
```

```
                                    ;bytes).
        .WORD   -1                  ;Default to 2 Kbytes.

        .LOC    CON+?IBAD           ;Set byte pointer to record I/O
        .DWORD  ITEXT*2             ;buffer.

        .LOC    CON+?IRCL
        .WORD   120.                ;Record length is 120
                                    ;characters.

        .LOC    CON+?IFNP           ;Set byte pointer to pathname.
        .DWORD  CONS*2

        .LOC    CON+?IDEL           ;Delimiter table address.
        .DWORD  -1                  ;Use default delimiters: null,
                                    ;NEW LINE, form feed, and
                                    ;carriage return.

        .LOC    CON+?IBLT           ;End of packet.

;Filename, message, and buffer.

CONS:   .TXT "@CONSOLE"             ;Use generic name.

ITEXT:  .TXT "Type filename of file you want to create.  "

BUF:    .BLK    50.                 ;Allocate enough space for
                                    ;buffer.
TMES:   .TXT "created with ACL of WSR.<12>"

        .NOLOC  0

        .END    FILCREA             ;End of FILCREA program.
```

End of Chapter

Chapter 5
Performing Input/Output (I/O)

The system calls that you use to perform Input/Output are

?ALLOCATE	Allocate disk blocks.
?ASSIGN	Assign a device to a process for record I/O.
?BLKIO	Perform block I/O.
?CLOSE	Close a file previously opened for record I/O.
?CRUDA	Create a user data area (UDA).
?DEASSIGN	Deassign a character device.
?FLOCK	Lock or return the lock status of a file element.
?FUNLOCK	Unlock a file element.
?GCHR	Get the characteristics of a character device.
?GCLOSE	Close a file previously opened for block I/O.
?GDLM	Get a delimiter table.
?GECHR	Get extended characteristics of character device.
?GOPEN	Open a file for block I/O.
?GPOS	Get the file pointer position.
?GTRUNCATE	Truncate a disk file (block I/O).
?INTWT	Define a terminal interrupt task.
?KINTR	Enable interrupts to virtual terminals.
?KIOFF	Disable interrupts to virtual terminals.
?KION	Reenable interrupts to virtual terminals.
?KWAIT	Define and enable an interrupt task for virtual terminals.
?LABEL	Create a label for a magnetic tape.
?ODIS	Disable terminal interrupts.
?OEBL	Enable console interrupts.
?OPEN	Open a device for record I/O.
?PRDB/?PWRB	Perform physical block I/O.
?RDB/?WRB	Perform block I/O.
?RDUDA	Read a user data area (UDA).
?READ	Read a record for record I/O.
?RELEASE	Release an initialized logical disk (LD).
?SCHR	Set the characteristics of a character device.
?SDLM	Set delimiter table.
?SECHR	Set extended characteristics of a character device.
?SEND	Send a message to an operator.
?SPOS	Set the position of the file pointer.
?STOM	Set the time-out value for a device.
?TRUNCATE	Truncate a disk file or magnetic tape file (record I/O).
?UPDATE	Flush file descriptor information.
?WRITE	Write a record for record I/O.
?WRUDA	Write a user data area (UDA).

This chapter describes how to perform file I/O. It describes how AOS/VS stores and accesses files, and the steps that you take to perform file I/O. The chapter then describes the various types of I/O — record, block, and physical block I/O — and it describes how to perform I/O on various types of files and devices.

File Structure Under the AOS/VS Operating System

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 AOS/VS stores your files. Blocks vary in size from device to device. Consequently, 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 associate physical blocks on a device with a certain 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 ?LOCHN mnemonic represents the lowest possible channel number and the ?HICHN mnemonic 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

The sequence of operations, and the system calls that you use to perform these operations for record and block I/O, are as follows.

Operation	Record I/O Call	Block I/O Call
1. Open the file.	?OPEN	?GOPEN
2. Read/Write to the file.	?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 only certain system calls. 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 0.

The creation option in the ?OPEN packet allows you to simultaneously create and open certain file types. Table 5–1 lists the file types you can create with the creation option. If you accept the default file type when you select the creation option, AOS/VS will create the new file as a user data file (type ?FUDF). You can 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 reocrd in the file simultaneously.

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. However, the ?UPDATE system call does not write a file's data to disk, just its file descriptor information. This file descriptor information includes the file's UDA.

Table 5-1. 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 accept the default value for only the following parameters: hash frame size, maximum number of index levels, and ACL.
?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.
?FPIP	Pipe File	A pipe file is a transient, byte-oriented file that you can use to transfer data between processes.

File Pointers

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.

Record I/O

Record I/O is the process of reading or writing to files that exist on a device, in logical groupings called *records*. There are four types of records: *dynamic-length, fixed-length, data-sensitive,* and *variable-length* records.

Dynamic-Length Records

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 Records

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 Records

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. AOS/VS then 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 ERLTL (line too long) error code 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 Records

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. Each record in a file can be different length.

Block I/O

Block I/O is the process of reading or writing to files that exist on a device, in physical units called blocks. The sizes of these blocks vary from device to device.

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 zeros 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 maximum size.

To perform block I/O on a file, you must know the number of blocks that 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. (For a description of the block I/O packet structure, see the description of the ?RDB/?WRB and ?BLKIO system calls in the *AOS/VS and AOS/DVS System Call Dictionary*.)

The ?RDB/?WRB and ?BLKIO calls are similar, however, the ?BLKIO system call includes additional functionality that allows it to read 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. (For examples of how this feature works, see the description of ?BLKIO in the *AOS/VS and AOS/DVS System Call Dictionary*.)

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.

Reuse of Disk Blocks

AOS/VS prevents processes from reading any data that might remain on a previously used — but currently unused — disk block.

When a process issues the ?ALLOCATE system call, AOS/VS writes zeros into every block that it has not previously allocated; a file will never contain spurious data from a file that had previously used one of its blocks.

Similarly, if a process issues a ?RDB system call to an open disk file and specifies that it wants to read a block that AOS/VS has not allocated to that file, AOS/VS will return a block of zeros. The specified block remains unallocated.

Physical Block I/O

AOS/VS supports physical block I/O for disks and tapes. 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 the I/O retries that AOS/VS makes when it encounters 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.

Returning Lock Status

You can also use the ?FLOCK system call to return the lock status of a file or a file element. If you set the ?FTCK parameter in the ?FLTYP offset of the packet, the ?FLOCK system call will return the following status information:

- The type of lock (exclusive or shared).
- The 32–bit number identifying the locked element.
- The PID of the process that has locked the element.

If the element is not currently locked, the ?FLOCK system call will return a 0 in the fields for the lock type and PID.

Unlocking Files and File Elements

To unlock a file or the elements in a file, the *locking process* must issue the ?FUNLOCK system call. The process can unlock either

- Those elements for which it gives the 32–bit identification number of the first element in a range of elements.
- *All* elements that it has locked through the channel that it specifies in the ?FLCHN offset of the packet.

Only the process that has locked the file or file elements can issue the ?FUNLOCK system call that releases them from their locks. Should a process terminate without having explicitly released its locks, the AOS/VS lock manager will release all locks that it granted to that process.

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; when you use @ as a filename prefix, AOS/VS recognizes the filename as either a device name or a generic filename (we describe generic files in the next section). Table 5–2 gives a complete list of the AOS/VS devices and their device names.

Table 5-2. AOS/VS Devices and Device Names

Name	Device
IOP	Asynchronous line multiplexor (IAC, MCP1, or in an ATI).
@CON0	System Console.
@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.
@LPB, @LPB1 through @LPB7	Data channel line printers 0 through 17.
@LMT	Labeled magnetic tape.
@LFD	Labeled floppy diskette.
@MCA and @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.

Generic File Names

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 5–3 lists the six generic filenames and the files that they represent. Like device names, generic filenames require the @ prefix.

Table 5–3. Generic Filenames

Filename	Refers To
@CONSOLE	Any interactive device associated with a process (usually a user terminal).
@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.

For an interactive process, your termninal 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 the @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

MYDIR is the current working directory.

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. (For more information on the ?PROC packet, see the description of the ?PROC system call in the *AOS/VS and AOS/DVS System Call Dictionary*.)

Usually, AOS/VS copies the data it reads from the @INPUT file to the @OUTPUT file. However, if @INPUT and @OUTPUT are both terminals, then the @INPUT function echoes data to the @OUTPUT terminal. 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 MCA protocol that Data General's AOS, RDOS, and RTOS operating systems use.

Each MCA enables two or more physical processors to communicate across a data channel. Hardware links connect the MCA units. A single MCA can connect a single physical processor to as many as 14 other processors. By adding a second MCA (MCA1), you can connect another 15 processors.

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 from 0 through 15

The link number indicates which remote processor you are communicating with when your local processor is linked to more than one remote processor.

Character Devices

Character devices are devices that perform I/O in bytes. CRT and hard-copy terminals are typical character devices.

Character 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. These device characteristics include

- The line length of the output.
- Whether the device is ANSI standard or non–ANSI standard.
- Whether the device echoes characters.
- Whether the device uses hardware tab stops or 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 terminal control character. (See the "Terminal Format Control" section in this chapter for a description of the terminal 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 five-word buffer in AC2 when you issue the ?SCHR system call or the ?SECHR system call. Usually, you will 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 terminal lines. They also control characteristics such as baud rates for Intelligent Asynchronous Controllers (IACs). For more information on these extended characteristics, see the description of the ?SECHR system call in the *AOS/VS and AOS/DVS System Call Dictionary*.

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 terminal without opening and closing the terminal. This means that you can pass messages from real-time processes without terminals 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. The following sections describe the operating procedures for each modem type. Table 5–4 lists the flags used in modem operation.

Table 5–4. 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 terminal 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 operating sequence for an auto-answer modem is as follows:

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. If DSR lapses during the I/O sequence, or if CD lapses for more than 5 seconds, the I/O call terminates with an error return.

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 characteristics word. (See the descriptions of the ?GCHR and ?SCHR system calls in the *AOS/VS and AOS/DVS System Call Dictionary.*) Like the ?CMRI characteristic, you can only set the ?CMOD characteristic during the system-generation procedure, unless you are PID 2.

AOS/VS uses the monitor ring indicator to detect incoming calls to a non-auto-answer modem. If you select the ring-indicator 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 carrier signal within 5 seconds of the modem interrupt, it posts a disconnect against the line. When this occurs, you must close the modem-controlled device and re-open it.

The operating sequence for non-auto-answer modems with the monitor ring indicator option is as follows:

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 the ring-indicator characteristic (?CMRI) during the system-generation procedure.

3. When a remote user places a call, the MV/Family 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.

Card Readers

AOS/VS also recognizes card readers as character devices. The operating sequence for card readers is as follows:

1. When you open a card reader, AOS/VS starts it for input. The card reader then reads ahead as many cards as will fit in its ring buffer. AOS/VS does not restart the card reader until there is room in the ring buffer for an entire card.

2. If the card reader is in Text mode when you issue the ?READ system call, AOS/VS performs the Hollerith-to-ASCII conversion. If the card reader is not in Text mode, AOS/VS does not convert Hollerith code to ASCII code.

3. If AOS/VS encounters a non-Hollerith card when you issue the ?READ system call, it returns the file read error code ERFIL.

4. When AOS/VS reads a card that has all rows punched in column 1, it returns an end-of-file condition.

5. AOS/VS assumes that all cards are at most 81 columns long. Because it does not check column length on input, mark-sense card readers are compatible with AOS/VS.

6. If the card reader is in Binary mode, you can set the packed characteristic for its input. This allows you to pack four 12–bit columns into three 16–bit words. Without the packing option, AOS/VS right-justifies the 12 bits in the buffer and uses the 4 upper bits for the following octal status codes:

100000	End of file.
040000	Hopper empty or stack full.
020000	Pick fail.
010000	Read error.

7. If you set the trailing blanks characteristic (?CTSP), AOS/VS retains all trailing blanks on the cards. If you omit this characteristic, AOS/VS discards all trailing blanks and writes a NEW LINE character after the last character on each card — this allows you to fit more cards into the ring buffer.

8. The no NEW LINE characteristic (?CNNL) directs AOS/VS to ignore all NEW LINE characters in each card.

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 can issue the ?ASSIGN system call only against a file that is not 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 0).

Device assignment works somewhat differently for user terminals. All son processes can share their father's terminal, even if the terminal was specifically assigned to the father. However, only the most recently created son can actually control the terminal by 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 terminal.

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 UDA for the file. The ?CRUDA system call creates a UDA. The ?RDUDA and ?WRUDA system calls read and write UDA information, 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 filename of the UDA that you want to create.

   This file can contain format specifications or, if you want, it can be empty.

2. Execute FCU. (Refer to the *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)* for more information on FCU.)

3. Move the newly created UDAs to the :UTIL: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 the 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. (Refer to the *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)* for more information on the CLI switches.)

Terminal Format Control

Several control characters and control sequences allow you to control the output that displays on your terminal.

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. This causes AOS/VS to pass the control character to your program. Table 5–5 lists the control characters and what they do.

Table 5-5. 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 terminal 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 terminal, 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 terminal.
CTRL–T	Reserved for future use by Data General. (Currently, these control sequences do nothing. However, if you CTRL–V precede either one with 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 terminal 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 5-6 lists the control sequences and what they do.

Table 5-6. Control Sequences and Their Functions

Control Sequence	Function
CTRL–C CTRL–A	Generates a terminal interrupt (provided you used the ?INTWT system call to define a terminal interrupt task — we describe the ?INTWT system call in the following section).
CTRL–C CTRL–B	Generates a terminal interrupt and aborts the current process.
CTRL–C CTRL–C	Echoes the characters ∧C ∧C on the terminal, and empties your type-ahead buffer. (This is useful when you want to revoke a command you have typed ahead.)
CTRL–C CTRL–D	Reserved for use by Data General.
CTRL–C CTRL–E	Generates a terminal interrupt, aborts the current process, and creates a break file.
CRTL–C CTRL–F through CTRL–C CTRL–Z	Reserved for use by Data General.

Defining, Enabling, and Disabling Terminal Interrupts

Before you can use a CRTL–C CTRL–A sequence as a terminal interrupt, you must define an interrupt processing task. To define an interrupt processing task, you issue an ?INTWT system call. The ?INTWT system call defines a task that monitors the terminal keyboard for CTRL–C CTRL–A sequences. When AOS/VS detects a CTRL–C CTRL–A sequence, it readies the interrupt processing task that you defined and passes control to the ?INTWT system call's normal return. AOS/VS reenables terminal interrupts only when you reissue either another ?INTWT system call, or when you issue ?OEBL system call.

By default, AOS/VS enables terminal interrupts when a program begins to execute. You can issue the ?ODIS system call to override this default. You can also use the ?ODIS system call to disable an interrupt that you previously enabled by issuing an ?OEBL, ?INTWT, or ?CHAIN system call.

Defining, Enabling, and Disabling Interrupts to Virtual Terminals

You can define, enable, and disable interrupts to virtual terminals. To enable virtual terminals to handle interrupts as if they were real terminals, issue the ?KINTR system call. Should you want the virtual terminal's process to handle interrupts through an interrupt task that you specify, issue the ?KWAIT system call.

To disable interrupts to the virtual terminal's process, issue the ?KIOFF system call; to reenable interrupts to the virtual terminal's process, issue the ?KION system call.

Using IPC Files as Communications Devices

In addition to the interprocess communications (IPC) procedures that we describe in Chapter 8, you can use IPC files as a communications devices and perform I/O against them. When you perform I/O against an IPC file, AOS/VS buffers the IPC messages in first-in/first-out (FIFO) order. The sequence of operations for using an IPC file as a communications device is as follows:

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.)* For more information on the global ?IREC option, see Chapter 8.

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 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.

When 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 first ?OPEN system call) owns the file.

Transferring Data through a Pipe File

You can also transfer data through a *pipe file*. A pipe file is a special kind of disk file that allows you to transfer data through a first in, first out (FIFO) queue of bytes that more than one process can access. To write to the pipe file, a process appends bytes to the tail of the queue. To read from a pipe file, a process removes bytes from the head of the queue. A pipe is the conceptual opposite of a stack, which operates on the last in, first out (LIFO) principle.

In the Figure 5–1, let's say that Process A has placed the byte groups A1, A2, and A3 successively into a pipe and that Process B has removed byte group A1 from the pipe.

Most commonly, a pipe is two-ended: a least one process has opened the pipe for reading, and at least one process has opened the pipe for writing. However, a pipe can become one-ended under the following conditions:

- All processes that have opened the pipe for reading have closed it, while at least one process currently has the pipe open for writing.
- All processes that have opened the pipe for writing have closed it, while at least one process currently has the pipe open for reading.

Because a pipe is a vehicle for interprocess communications — it joins the output stream of a process to the input stream of another process — it is a transient entity. Once all of the processes that are reading or writing to a pipe have closed their respective ends, AOS/VS deletes the pipe.

Boundry Conditions in Pipes

While the the number of bytes that a pipe contains can vary, a pipe has a fixed length and it can hold only so many bytes. When a pipe becomes empty or full, a boundry condition can exist. The boundry conditions, and the default actions that the AOS/VS system takes, are as follows.

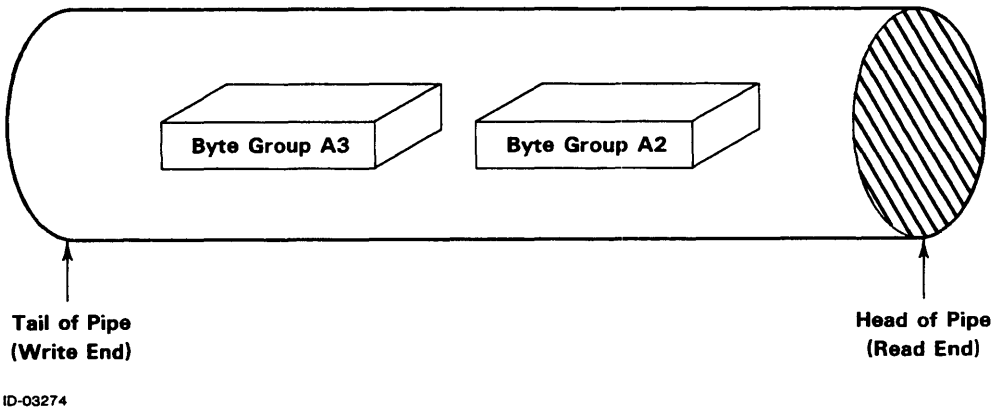


Figure 5-1. Diagram of a Pipe File.

Boundry Condition	Result
Attempt to read from an empty, one-ended pipe.	AOS/VS returns error code EREOF (end of file), provided that you have opened the write end of the pipe at least once.
Attempt to write to a full, one-ended pipe.	AOS/VS returns error code ERPFL (pipe is full), provided that you have opened the read end of the pipe at least once.
Attempt to read from an empty, two-ended pipe.	Because at least one other process has opened the write end of the pipe, AOS/VS pends the reading process until another process writes into the pipe. If all of the writing processes close their end of the pipe without writing into it, AOS/VS will return error code EREOF as above.
Attempt to write to an full, two-ended pipe.	Because at least one process has opened the read end of a pipe, AOS/VS pends the writing process until another process reads from the pipe. If all of the reading processes close the pipe without having read from it, AOS/VS will return error code ERPFL as above.

Creating a Pipe

You can create a pipe with either the ?CREATE or ?OPEN system calls. You can also create a pipe with the CLI command

```
CREATE/TYPE=PIP filename
```

If you create a pipe with either the CREATE CLI command or the ?CREATE system call, the length of the pipe will be two pages (4096. bytes) and the pending action of processes using that pipe will be as we described previously.

If you supply the pipe extension packet, you can create the pipe with the ?OPEN system call. In the ?OPEN system call, you can specify both the length and the pending action of the processes that use the pipe.

Specifying the Pipe Length

In the ?OPEN system call, you can specify a pipe length of 1 to 16 pages (from 2048. to 32768. bytes). If you specify a byte length that is not divisible by 2048. (one page), the system rounds the pipe length up to the nearest page.

Specifying the Pending Action

In the ?PIPD offset of the extension packet, you can specify one of the following pending actions.

Offset Value	Pending Action Specified
?PALW	A process *always* pends when trying to read from an empty pipe or write to a full pipe, regardless of whether the pipe is one- or two-ended.
?PNVR	A process *never* pends when trying to read an empty pipe or write to a full pipe, regardless of whether the pipe is one- or two-ended. AOS/VS returns error code EREOF if trying to read, error code ERPFL if trying to write.
?PTWO	*The default* — a process pends when trying to read an empty pipe or write to a full pipe, but only when the pipe is two-ended. When the pipe is one-ended, AOS/VS returns error code EREOF when trying to read, error code ERPFL if trying to write.

Opening a Pipe for I/O

To open an existing pipe for I/O, you must use the ?OPEN system call.

The ?OPEN system call opens only one end of the pipe. To open the *head* (read end) of the pipe, you set the input bit (?OFIN) of the ?OPEN packet. To open the *tail* (write end) of the pipe, set the output bit (?OFOT) of the pipe.

If you try to open the pipe for both reading and writing (?OFIO), or for shared I/O (?SHOP), the system returns the error code ERIOO (illegal open option for file type). The system ignores all other extensions to the ?OPEN packet (with the exception of the pipe and field translation extensions).

Reading and Writing to Pipes

To read and write to a pipe, you use the ?READ and ?WRITE system calls, respectively. You need only include the pipe extension to the ?READ and ?WRITE packets when you want to specify the pending action of the processes reading or writing to the pipe. With the exception of the pipe and field translation extensions, the system ignores all other extensions to the ?READ and ?WRITE packets.

A pipe is byte-oriented; if you try to read or write to a pipe with record type ?RTVB (variable block, variable record), the system will return error code ERRFM (illegal record format).

Closing or Deleting a Pipe

To close or delete a pipe, you can use the ?CLOSE or ?DELETE system calls, respectively. These system calls do not require any pipe-specific arguments.

Controlling Access to a Pipe

A pipe is a disk file; you can control acces to the pipe through the pipe's ACL (access control list). To open a pipe, a process must have *both* Read and Write access to the pipe.

You can change a pipe's ACLs by issuing either of the following system calls:

?SACL	Resets the pipe file's ACL.
?RENAME	Changes the pipe file's name and, optionally, resets the pipe file's ACL.

These system calls also do not require any pipe-specific arguments.

Invalid System Calls

Because a pipe is a transient, byte-oriented file, the system returns the error code ERIFT (illegal file type) if you issue any of the following system calls against a pipe:

?ALLOCATE	Allocate disk blocks.
?ESFF	Flush shared pages to disk.
?GCLOSE	Close a file open for block I/O.
?GOPEN	Open a file for block I/O.
?GPOS	Get file position.
?GTRUNCATE	Truncate a file for block I/O.
?PRDB/?PWRB	Perform physical block I/O.
?RDB/?WRB	Perform block I/O.
?RECREATE	Recreate a file.
?RPAGE	Release a shared page.
?SATR	Set or remove the permanence attribute for a file.
?SCLOSE	Close a shared file.
?SOPEN	Open a shared file.
?SOPPF	Open a protected shared file.
?SPAGE	Read in a shared page.
?TRUNCATE	Truncate a file opened for record I/O.

In addition, issuing the ?SPOS system call (set file position) results in a no–op.

Performing I/O to Labeled Magnetic Tapes

A labeled magnetic tape contains both user data and information about that data — the latter in the form of system and user labels. Labeled magnetic tapes provide the following advantages over unlabeled magnetic tapes:

- ANSI-standard and IBM formats, which enable you to use a labeled magnetic tape on another operating system.
- A naming facility, so you can reference your tape file by name rather than by tape number.
- Volume identifiers (volids), so that a logical file can span several physical tape reels.
- Detailed information about when and how much I/O is actually performed for a particular device.

You can use either the CLI LABEL utility or the ?LABEL system call to create labels for a magnetic tape. After you complete the labeling procedures, you can create files on the tape.

Labeling Formats

AOS/VS supports two primary labeling formats: ANSI format (Levels 1, 2, or 3), which uses the ASCII character set, and IBM format (Level 1 or 2), which uses the EBCDIC character set. This allows you to select a format and labeling level suitable for use on another operating system. The formats and levels differ in the number of files allowed in a volume set, the allowable record types, the types of labels, and the contents of the labels. Table 5–7 defines the number of files and record types allowed for each label format and level.

Table 5–7. Label Formats and Levels: Files per Volume Set, Record Types

	Specification	Format	Level
No. of Files	Single file, single volume	ANSI IBM	1, 2, 3 1, 2
	Single file, multiple volume	ANSI IBM	1, 2, 3 1, 2
	Multiple file, single volume	ANSI IBM	2, 3 1, 2
	Multiple file, multiple volume	ANSI IBM	2, 3 1, 2
Record Types	Fixed-length	ANSI IBM	1, 2, 3 1, 2
	Variable-length	ANSI IBM	3 1, 2
	Variable-length spanning blocks	IBM	2
	Undefined-length	IBM	1, 2
	Data-sensitive	N/A	N/A
	Dynamic	N/A	N/A

If you do not set any flags in offset ?IRES of the ?OPEN packet, AOS/VS assumes that you want to use labeled tapes in AOS format. However, if you want to use labeled tapes in ANSI or IBM format, you must set one of the following flags in offset ?IRES:

- Set ?OANS to use labeled tapes in ANSI format.
- Set ?OIBM to use labeled tapes in IBM format.

AOS/VS does not write the data in EBCDIC. To do this, you must select the field translation packet when you issue the ?READ or the ?WRITE system call.

The labeling level that you select should be compatible with the label support of the operating system on which you will use the tape. ANSI Level 3 and IBM Level 2 are the default levels. However, you can select a lower level within the ?LABEL system call packet, or within the ?OPEN system call packet extension for labeled tapes.

If you select a lower level before writing to the tape, AOS/VS will record less information about your data in the labels. If you select a lower level before reading from the tape, AOS/VS ignores some of the information in the labels. Because AOS/VS can read a tape to a lower level than you specify (for example, AOS/VS can read an ANSI Level 1 tape even if you specify it to be an ANSI Level 3 tape), you should default to the highest level.

Label Types

There are four types of labels:

- Volume labels

  These labels identify the volume (reel) of magnetic tape; they occur only at the start of each volume.

- File header labels

  These labels identify the file and its characteristics; they occur before every file on a labeled tape. If the file spans volumes, each file section starts with file header labels.

- End-of-file labels

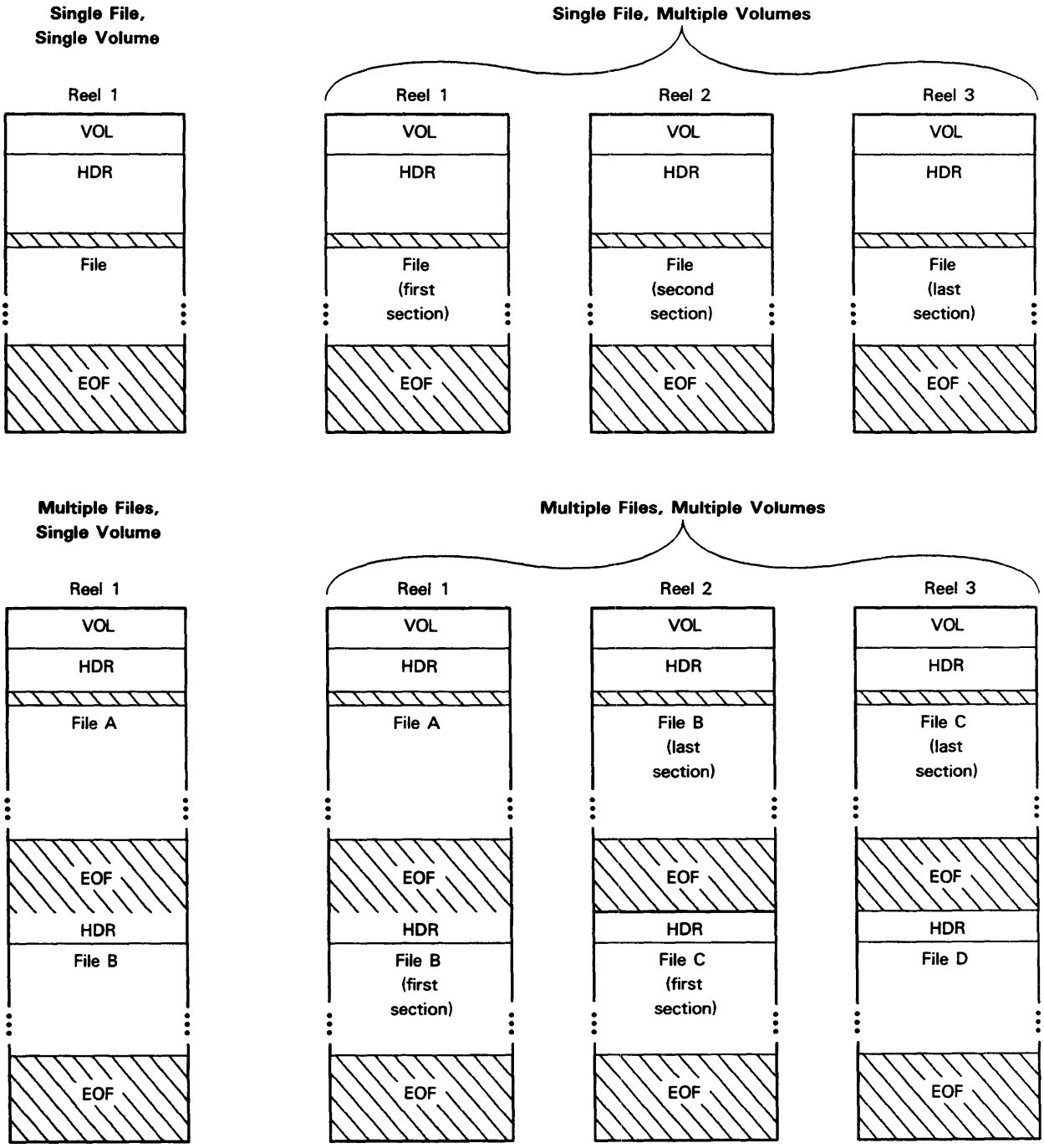
  These labels identify the file and its characteristics; they occur after every file on a labeled tape.

- End-of-volume labels

  These labels identify the file and its characteristics; these occur at the end of a volume of tape to indicate that the file spans volumes.

Figure 5–2 shows how AOS/VS writes labels and data to a labeled tape.

Each type of label contains one or more individual labels. Some labels are necessary and must be present, or AOS/VS returns an error. Other labels are used if present, but are not required, and some are permitted but are not used. ("Permitted" labels do not cause errors; AOS/VS ignores the information in them.) Table 5–8 lists the different types of labels for the various formats and levels.



KEY:

VIOL	Volume labels
HDR	Header labels
EOF	End-of-file labels
EOV	End-of-volume labels
\\\	Tape mark; separates data (Two consecutive tape marks represent an end-of-tape mark.)

ID-03275

Figure 5-2. Labels and Data on a Labeled Magnetic Tape

Table 5–8. Types of Labels

Labels	ANSI(1)	ANSI(2)	ANSI(3)	IBM(1)	IBM(2)
Volume Labels:					
VOL1 (Volume 1)	N	N	N	N	N
UVL1–9 (User Volumes 1 – 9)	P	P	P	P	P
File Header Labels:					
HDR1 (Header 1)	N	N	N	N	N
HDR2 (Header 2)	P	P	U	P	U
HDR3–9 (Headers	P	P	P	P	P
UHL1–9 (User Headers 1 –9)	P	U	U	U	U
End-of-File Labels:					
EOF1 (End of File 1)	N	N	N	N	N
EOF2 (End of File 2)	P	P	U	P	U
EOF3–9 (End of Files 3 – 9)	P	P	P	P	P
UTL1–9 (User Trailers 1 – 9)	P	U	U	U	U
End-of-Volume Labels:					
EOV1* (End of Volume 1)	N	N	N	N	N
EOV2* (End of Volume 2)	P	P	U	P	U
EOV3–9* (End of Vols 3 – 9)	P	P	P	P	P
UTL1–9 (User Trailers 1 – 9)	P	U	U	U	U

KEY:
- N Necessary
- U Used if present, but not required
- P Permitted, but not used
- * End-of-volume labels are necessary only if the file spans reels

Volume Labels

As Table 5–9 indicates, each labeled tape volume must begin with a volume 1 label (VOL1) of 80 bytes (characters). Table 5–10 lists the required contents of the VOL1 labels. The system supplies the characters in quotation marks (for example, "VOL1").

The *volid*, or tape volume identifier, must consist of up to six characters from the following character set:

- Alphabetic characters A through Z, uppercase only
- Numerals 0 through 9
- Special characters ! ‘ % ( ) * + , - . / ; < > = ?

The volid is part of the pathname you use to refer to a labeled tape file.

The Access field, which is used for ANSI tapes, defines the users allowed to access the tape. You must use a blank space character (ASCII 40) in this field. Otherwise, AOS/VS does not allow access to the volume. The space character allows anyone access to the volume.

Table 5-9. Contents of VOL1 Volume Labels

Byte Position	ANSI(1)	ANSI(2)	ANSI(3)	IBM(1)	IBM(2)
01–04	"VOL1"	"VOL1"	"VOL1"	"VOL1"	"VOL1"
05–10	Volid	Volid	Volid	Volid	Volid
11	Access	Access	Access	"0"	"0"
12–37	Blank	Blank	Blank	Blank	Blank
38–41	Owner name	Owner name	Owner name	Blank	Blank
42–51	Owner name	Owner name	Owner name	Owner name	Owner
52–79	Blank	Blank	Blank	Blank	Blank
80	Version number	Version number	Version number	Blank	Blank

The optional Owner Name field identifies the owner of the volume. AOS/VS ignores this field when you reference a file on the volume. The default value for this field is a blank space.

The Version Number field specifies the ANSI label format (version) you want for labeled tape processing. This field must contain 1, 2, or 3 if you intend to read the tape. AOS/VS uses version number 3 when you write to the tape.

If you use the ANSI label format, you can follow the VOL1 label with as many as nine optional user volume labels (UVLs) to record additional data about all files on the volume. Note that you cannot use UVLs for tapes that are in IBM format.

Each UVL can contain up to 76 bytes of data. Bytes 1 through 3 contain the character string "UVL", which AOS/VS supplies. AOS/VS numbers UVLs consecutively from 1 through 9. Byte 4 contains the label number.

Table 5–10 lists the contents of a UVL.

Table 5-10. Contents of User Volume Labels (UVLs)

Byte Position	ANSI(1)	ANSI(2)	ANSI(3)
01–03	"UVL"	"UVL"	"UVL"
04	Label number	Label number	Label number
05–80	User data	User data	User data

Header 1 Labels

A Header 1 (HDR1) label of 80 bytes must follow the VOL1 label, regardless of the tape's format or labeling level. Table 5–11 describes the contents of HDR1 labels. AOS/VS supplies the characters in quotation marks (for example, "HDR1").

AOS/VS assigns a sequence number to each file on a labeled tape volume set. If the file spans volumes, AOS/VS divides the file into sections and assigns each section a section number. AOS/VS uses the File Section Number and File Sequence Number fields, and a third field, Block Count, for error detection, as follows:

- File Section Number.

  The File Section Number indicates which section of the file AOS/VS is currently processing. AOS/VS checks this field to to see if the volume contains the proper file section so that AOS/VS can process the file section in the correct order.

- File Sequence Number.

  The File Sequence Number indicates the order of the files in the volume set. An incorrect sequence number means that you have mounted the wrong volume.

- Block Count.

  The block count indicates the number of blocks written to the file; if the block number on the end-of-file (EOF) or end-of-volume (EOV) label is not the number actually read, a block may have been skipped.

Table 5-11. Contents of HDR1 File Header Labels

Byte Position	ANSI(1)	ANSI(2)	ANSI(3)	IBM(1)	IBM(2)
01–04	"HDR1"	"HDR1"	"HDR1"	"HDR1"	"HDR1"
05–21	Filename	Filename	Filename	Filename	Filename
22–27	File ID set	File ID set	File ID set	File ID set	File ID set
28–31	File section number	File section number	File section number	File section number	File section number
32–35	File sequence number	File sequence number	File sequence number	File sequence number	File sequence number
36–39	"0001"	"0001"	"0001"	Blank*	Blank*
40–41	"00"	"00"	"00"	Blank*	Blank*
42–47	"00000"	"00000"	Creation date	Creation date	Creation date
48–53	Expiration date	Expiration date	Expiration date	Expiration date	Expiration date
54	" " (blank space character)	Access	Access	Access	Access
55–60	Block count	Block count	Block count	Block count	Block count
61–73	System ID	System ID	System ID	System ID	System ID
74–80	Blank	Blank	Blank	Blank	Blank

* For IBM levels 1 and 2, Bytes 36 through 41 contain information that AOS/VS does not use during processing.

The Expiration Date field prevents AOS/VS from overwriting the data on a labeled tape before the specified date. The default expiration date is 90 days after the tape's creation date.

The Access field (like the Access field in the VOL1 label) defines the users allowed to access the tape. For ANSI format, the default for this field is a blank space character (ASCII 40). For IBM format, the default is 0. This gives all users access to the data. Be sure to use the proper default value. If you use another character in this field, AOS/VS assumes that additional access privileges are required, and will not allow access to the tape.

AOS/VS checks the following fields to see that the file on the tape matches the file you requested for I/O.

- File Set Identifier.

  The File Set Identifier field identifies the file set. (A file set is a group of files that occupies one or more volumes.) AOS/VS checks the File Set Identifier to see that the newly mounted volume belongs to the file set. By default, the File Set Identifier is the volid (volume identifier) or the first volume in the file set.

- Filename.

  The Filename field identifies the file you want to process. There is no default value for this field.

- Generation Number.

  The Generation Number field indicates the file's generation. (The default generation number is 0001.) A file can appear on a tape more than once, if each occurrence has a different generation number. This is useful for recording changes to a file.

- Version Number.

  The Version Number field indicates which version of a certain file generation you are referencing. (The default version number is 00.) Only one version of a file's generation can appear on a tape.

Header 2 Labels

AOS/VS allows additional header labels (HDR2 and HDR3 through 9), but these are not required. In fact, AOS/VS uses only Header 2 labels (HDR2), if present, or the ANSI Level 3 and IBM Level 2 formats; it ignores header 3 through 9 (HDR3–9) for all formats and levels.

If you do use HDR2 labels, they must contain the information shown in Table 5–12. AOS/VS enters the characters in quotation marks (for example, "HDR2").

Table 5-12. Contents of HDR2 File Header Labels

Byte Position	ANSI(3)	IBM(2)
01–04	"HDR2"	"HDR2"
05	Record type	Record type
06–10	Block length	Block length
11–15	Record length	Record length
16–38	Blank	Blank*
39	Blank	Block attribute
40–50	Blank	Blank*
51–52	Buffer offset	Blank*
53–80	Blank	Blank*

* For IBM Level 2, Bytes 40 through 80 contain information that AOS/VS does not use during processing.

The HDR2 labels describe the record type, record length, and block length of the data. The ?OPEN packet conveys this information to AOS/VS. The fields on the HDR2 labels are as follows:

- Record Format.

  The Record Format field is dynamic, fixed-length, data-sensitive, or variable length. The record format field must match the specification in offset ?ISTI of the ?OPEN packet. You cannot default this value if you intend to write to the tape. If you intend to read the tape and there is no HDR2 label, the record type defaults to fixed-length.

- Block Attribute.

  The Block Attribute field states whether the records are blocked (several records per physical block), unblocked (only one record per block), or spanned (a record occupies two or more consecutive blocks). AOS/VS writes all records in blocked format. (You can specify spanned for variable-length records with the special variable-block record type, ?RTVB.)

- Block Length.

  The Block Length field states the maximum length of each physical block on the tape; offset ?IMRS in the ?OPEN packet governs this value. If you choose the ?IMRS default (–1), AOS/VS uses 2048 bytes as the block length when it is writing, and the value of the HDR2 field when it is reading.

- Record Length.

  The Record Length field states the maximum length of each record; offset ?IRCL in the ?OPEN packet conveys this value. If you choose the ?IRCL default (–1), AOS/VS uses 210 as the record length when it is writing, and this value in the HDR2 field when it is reading.

- Buffer Offset.

  The Buffer Offset field states the number of non-data bytes at the start of each physical block. AOS/VS ignores this field.

User Header and User Trailer Labels

In addition to file header labels and file trailer labels (end-of-file, end-of-volume), you can define user header and user trailer labels to supply further information about a labeled tape file. AOS/VS reads or writes these labels via the ?OPEN packet extension for labeled tapes. AOS/VS does not record these user-defined labels in the system labels.

Table 5–13 defines the contents of user header and user trailer labels. Notice that these labels have the same format as UVLs, except that bytes 1 through 3 contain the required strings "UHL" or "UTL", as appropriate.

Table 5–13. Contents of UHL and UTL User Labels

Byte Position	ANSI(2)	ANSI(3)	IBM(1)	IBM(2)
01–03	"UHL" or "UTL"	"UHL" or 'UTL"	"UHL" or 'UTL"	"UHL" or 'UTL"
04	Label number	Label number	Label number	Label number
05–80	User data	User data	User data	User data

End-of-Volume 1, End-of-File 1 Labels

End-of-volume 1 (EOV1) and end-of-file 1 (EOF1) labels have the same format as HDR1 labels, except that bytes 1 through 4 contain either "EOV1" or "EOF1", as appropriate. (See Table 5–11, shown earlier, for the format.)

End-of-Volume 2, End-of-File 2 Labels

End-of-volume 2 (EOV2) and end-of-file 2 (EOF2) labels have the same format as HDR2 labels, except that bytes 1 through 4 contain either "EOV2" or "EOF2", as appropriate. (See Table 5–12, shown earlier, for the format.)

File I/O on Labeled Magnetic Tapes

To use labeled tapes for file I/O, you must be logged on under the EXEC utility, either in batch or at a terminal. You cannot issue I/O system calls against a labeled tape from the system console, because the operator process (PID 2) is not a son of EXEC. The OP username must mount all labeled tapes, and the CLI command CONTROL @EXEC OPERATOR ON must be in effect. This command signals EXEC that the operator is available to mount the tapes.

There are two ways to mount a labeled tape: explicitly, by issuing the CLI MOUNT command; or implicitly, by issuing the ?OPEN system call. The CLI MOUNT command syntax is

```
MOUNT/VOLID=volid linkname operator-message
```

where

linkname	is the name of the link entry associated with the tape's filename.
operator-message	is a text string, which usually instructs the operator to mount the tape.
volid	is the 6–character volume identifier (See "Volume Labels" for the volid character set).

The CLI MOUNT command creates links for both labeled and unlabeled tapes. When you issue the CLI MOUNT command against a labeled tape, EXEC passes the message string to the operator and creates a link entry for the filename in your initial working directory.

The link resolves to @LMT:volid when you open, read, write, or close that tape volume. Note that EXEC creates the link entry in your initial working directory, not in the directory from which you issued the CLI MOUNT command.

When you perform primitive I/O or issue CLI commands against the labeled tape volume, you can substitute the tape's filename for @LMT:volid. After you read or write to a tape file that you opened with the CLI MOUNT command, use the CLI DISMOUNT command to tell the operator to remove the tape from the tape drive.

You can also mount a labeled tape with the CLI DUMP command, or any CLI command that accesses @LMT:volid. When you use this method, EXEC checks to see if the tape is already mounted. If it is not, EXEC directs the operator to mount it. The syntax of the CLI DUMP command is

```
DUMP @LMT:volid:filename
```

Each time you issue the CLI DUMP command, EXEC directs the operator to mount, and then dismount the tape. Thus, the CLI MOUNT command is usually the more efficient method.

If you mount the labeled tape with the ?OPEN system call, offset ?IFNP points to the name of the tape volume, which must be in the following form:

```
@LMT:volid:filename
```

AOS/VS does not create a link when you use this method, but it does tell the operator to mount the labeled tape volume specified in the pathname. When you close that tape file with the ?CLOSE system call, AOS/VS directs the operator to dismount the labeled tape volume.

Mounting a labeled tape explicitly with the CLI MOUNT command is the most efficient way to perform I/O on more than one labeled tape file, because AOS/VS does not need to rewind and reposition the tape for each I/O sequence or direct the operator to mount and dismount the tape for each ?OPEN and ?CLOSE system call. However, the ?OPEN system call is useful because it gives you the option of creating and opening the tape file at the same time.

When you read or write to a labeled tape, refer to the tape by one of the following pathnames

```
@LMT:volid:filename
```

where

@LMT is the generic filename for a labeled tape.

volid is the volume identifier number.

filename is the name of the file you wish to access.

— OR —

```
:UDD:username:linkname:filename
```

where

UDD is the name of the user directory.

username is your username.

linkname is the name of the link entry created by EXEC when the tape was mounted.

filename is the name of the tape file.

You do not need to cite a specific tape unit number for either of these formats. Use the second format if your current working directory is not :UDD:username.

EXEC creates the LMT entry and assigns it file type ?FGLT, the file type for labeled tapes. The filename you choose must consist of at least 1 and not more than 17 characters from the same character set you used for volid.

Because not all characters in this set conform to the character set for filenames, you cannot pass all labeled tape filenames through the CLI. (Instead, you must write your own programs, using the I/O system calls, to perform I/O on these labeled tapes.)

File I/O on Unlabeled Magnetic Tapes

To use a magnetic tape unit, you must first open it. To do this, specify the number of the tape unit and the position of the file on the tape (its file number) in the following form:

```
@MTBx:y
```

where

x is the number of the tape unit.

y is the file number.

Magnetic tape files are numbered sequentially, starting with 0. Thus, the pathname @MTB0:2 specifies the third file on tape unit 0. If you do not specify a file number, AOS/VS automatically opens the first file (file 0) on the tape.

If you use block I/O system calls to access a magnetic tape, you can specify the file number after you issue ?RDB or ?WRB system calls against the tape.

If you issue the CLI command MOUNT to signal the operator to mount a magnetic tape, use the linkname you used in the MOUNT command when you perform I/O against the file. For example, if you issue the following CLI command, you would be using the linkname TAPE1 to open, read, write, or close that file

```
MOUNT TAPE1 operator_message
```

In this case, AOS/VS would find TAPE1 in your initial working directory.

File I/O Sample Programs

The following program, RITE, opens the terminal and the disk file FILE. Then, RITE asks you to type lines of text at your terminal keyboard, and writes each line to FILE. When you type RD, RITE reads the lines back from FILE and displays them on your terminal screen.

RITE uses ?OPEN, ?READ, ?WRITE, and ?SPOS system calls.

```
                    .TITLE  RITE
                    .ENT    RITE
                    .NREL

;Open terminal (@CONSOLE) and file for input and output.

RITE:   ?OPEN   CON                 ;Open terminal (CON) for I/O.
        WBR     ERROR               ;Report error and quit.
        ?OPEN   FILE                ;Open or create disk file
                                    ;named FILE.
        WBR     ERROR               ;Quit.

;Write greeting and put byte pointer to I/O buffer in packet.

        ?WRITE  CON                 ;Display message on terminal.
        WBR     ERROR               ;?WRITE error return.
        XLEFB   0,BUF*2             ;Get byte pointer to I/O
                                    ;buffer.
        XWSTA   0,CON+?IBAD         ;Put in I/O packet.
```

```
;Read line, check for terminator, and then write to file.
        NLDAI   'RD',0                  ;Put RD terminator in AC0.

LOOP:   ?READ   CON                     ;Read a line.
        WBR     ERROR                   ;Quit.

        XNLDA   1,BUF                   ;Get first word of buffer.
        WSNE    0,1                     ;Did user type RD?
        WBR     SPOS                    ;Yes. Do ?SPOS.

        ?WRITE  FILE                    ;No. Write line to FILE.
        WBR     ERROR                   ;Quit.
        WBR     LOOP                    ;Get next line from user.

;Set position at beginning of file.

SPOS:   NLDAI   0,1                     ;Get 0 in AC1.
        XWSTA   1,FILE+?IRNH            ;Put in record number word.
        XNLDA   2,FILE+?ISTI            ;Get file's specifications.
        WMOV    2,0                     ;Save old specifications in
                                        ;AC0.
        WIORI   ?IPST,2                 ;Add ?IPST specification.
        XNSTA   2,FILE+?ISTI            ;Put in file specifications.

        ?SPOS   FILE                    ;Position at beginning of
                                        ;FILE.
        WBR     ERROR                   ;Quit.
        XNSTA   0,FILE+?ISTI            ;Restore old specifications.

;Read lines back from FILE and display on terminal.

LOOP1:  ?READ   FILE                    ;Read from FILE into buffer.
        WBR     EOF                     ;Try to handle the error.

        ?WRITE  CON                     ;Display line on terminal.
        WBR     ERROR                   ;Quit.

        WBR     LOOP1                   ;Read/write another line.

EOF:    NLDAI   EREOF,2                 ;Error code for end-of-file
                                        ;(EOF) is EREOF.
        WSEQ    0,2                     ;Was it an EOF?
        WBR     ERROR                   ;No. Quit.

;Close the file.

CLOSE:  ?CLOSE  CON                     ;Close terminal.
        WBR     ERROR                   ;Quit.

        ?CLOSE  FILE                    ;Close FILE.
        WBR     ERROR                   ;Quit.
        WSUB    2,2                     ;Set flags for normal return.
        WBR     BYE                     ;Take good return.
```

```
;Process error and/or return here.

ERROR:  WLDAI   ?RFEC!?RFCF!?RFER,2     ;Error flags: Error code is
                                        ;in AC0 (?RFEC), message is in
                                        ;CLI format (?RFCF), and
                                        ;caller should handle this as
                                        ;an error (?RFER).

BYE:    ?RETURN                         ;Return to CLI.
        WBR     ERROR                   ;?RETURN error return

;Open and I/O packet for terminal.

CON:    .BLK    ?IBLT                   ;Allocate enough space for
                                        ;packet.

        .LOC    CON+?ISTI               ;File specifications.
        .WORD   ?ICRF!?RTDS!?OFIO       ;Change format to
                                        ;data-sensitive records and
                                        ;open for input and output.

        .LOC    CON+?IMRS
        .WORD   -1                      ;Default physical block size
                                        ;to 2 Kbytes.

        .LOC    CON+?IBAD               ;Set byte pointer to record
        .DWORD  ITEXT*2                 ;I/O buffer.

        .LOC    CON+?IRCL
        .WORD   120.                    ;Record length is 120
                                        ;characters.

        .LOC    CON+?IFNP               ;Set byte pointer to pathname.
        .DWORD  CONS*2

        .LOC    CON+?IDEL               ;Delimiter table address.
        .DWORD  -1                      ;Use default delimiters: null,
                                        ;NEW LINE, form feed, and
                                        ;carriage return (default is
                                        ;-1).
        .LOC    CON+?IBLT               ;End of packet.

;Filename, buffer, and messages.

CONS:   .TXT "@CONSOLE"                 ;Use generic name.

BUF:    .BLK 60.                        ;Allocate enough space for
                                        ;buffer.

ITEXT:  .TXT "I write lines to file FILE.  Type RD[NL] to read lines
             back and stop.<12>"
        .NOLOC 0                        ;Resume listing all.

;?OPEN and I/O packet for FILE. You can omit those entries that you
;want to set to 0.
```

```
FILE:   .BLK    ?IBLT                   ;Allocate enough space for
                                        ;packet.

        .LOC    FILE+?ICH
        .WORD   0                       ;AOS/VS assigns channel
                                        ;number.

        .LOC    FILE+?ISTI                      ;File specifications.
        .WORD   ?OFCR!?OFCE!?ICRF!?RTDS!?OFIO   ;Delete file and then
                                                ;recreate it (?OFCR!?OFCE),
                                                ;change format (?ICRF) to
                                                ;data-sensitive records
                                                ;(?RDTS), and open for input
                                                ;and output (?OFIO).

        .LOC    FILE+?ISTO
        .WORD   0                       ;Default to ?FUDF, user data
                                        ;file.

        .LOC    FILE+?IMRS
        .WORD   -1                      ;Default physical block size
                                        ;to 2 Kbytes.

        .LOC    FILE+?IBAD              ;Byte pointer to record I/O
        .DWORD  BUF*2                   ;buffer.

.LOC    FILE+?IRES                      ;Density mode (for magnetic
                                        ;tapes only).
        .WORD   0                       ;Default it.

        .LOC    FILE+?IRCL
        .WORD   120.                    ;Record length is 120
                                        ;characters.

        .LOC    FILE+?IRLR              ;Number of bytes transferred.
        .WORD   0                       ;Only ?READ and ?WRITE use
                                        ;this.

        .LOC    FILE+?IRNW              ;Reserved.
        .WORD   0                       ;Set to 0.

        .LOC    FILE+?IRNH              ;Record number.
        .DWORD  0                       ;Only ?READ and ?WRITE use
                                        ;this.

        .LOC    FILE+?IFNP              ;Set byte pointer to pathname.
        .DWORD  FNAME*2

        .LOC    FILE+?IDEL              ;Delimiter table address.
        .DWORD  -1                      ;Use default delimiters: null,
                                        ;NEW LINE, form feed, and
                                        ;carriage return (default is
                                        ;-1).
```

```
        .LOC    FILE+?IBLT              ;End of packet.

FNAME:  .TXT    "FILE"                  ;Disk filename.

        .END    RITE                    ;End of RITE program.
```

Block I/O Sample Program

The block I/O sample program, DLIST lists all filenames in a directory and prints them on the line printer. DLIST uses the CLI ?GTMES mechanism (see Chapter 11) to get the directory name as well as using ?GOPEN to open the directory. Also, DLIST uses the ?OPEN, ?READ/?WRITE, ?GNFN, and ?SEND system calls. To execute DLIST, type the following.

```
X program_name directory_name
```

```
                .TITLE  DLIST
                .ENT    DLIST
                .NREL

;Get the directory name, open it, and open the line printer queue.

DLIST:  ?GTMES  CLIMSG                  ;Get directory name.
        WBR     ERROR                   ;?GTMES error return.
        LLEFB   0,DIRNAME*2             ;Get byte pointer to directory
                                        ;name.
        NLDAI   -1,1                    ;Specify that AOS/VS assign
                                        ;channel number for ?GOPEN.

        ?GOPEN  DIR                     ;Open the directory.
        WBR     ERROR                   ;?GOPEN error return.

        ?OPEN   LINEP                   ;Open the line printer queue.

        WBR     ERROR                   ;?OPEN error return.

;Use ?GNFN to get next name and write to line printer.

        XNLDA   1,DIR+?ICH              ;Keep channel number in AC1.

NEXT:   ?GNFN   GNAME                   ;Put filename in ?GNFN buffer.
        WBR     EOF                     ;?GNFN error return.

        ?WRITE  LINEP                   ;Send contents of ?GNFN
                                        ;buffer (filename) to line
                                        ;printer as output.
        WBR     ERROR                   ;?WRITE error return.
        XLEFB   2,NL*2                  ;Get address of NEW LINE
                                        ;character.
        XWSTA   2,LINEP+?IBAD           ;Put address of NEW LINE
                                        ;character in line printer
                                        ;buffer.
        ?WRITE  LINEP                   ;Send contents of buffer
                                        ;(address of NEW LINE
                                        ;character) to line printer
                                        ;as output.
```

```
        WBR     ERROR                   ;?WRITE error return.
        XLEFB   2,FNAME*2               ;Get byte pointer to filename
                                        ;buffer.
        XWSTA   2,LINEP+?IBAD           ;Restore buffer address.
        WBR     NEXT                    ;Get another filename.

EOF:    NLDAI   EREOF,2                 ;Is error code EREOF
                                        ;(end-of-file)?
        WSEQ    0,2                     ;Yes. Skip this instruction.
        WBR     ERROR                   ;No. Try to handle the error.

;Finished with filenames.  Get ?SEND parameters and issue ?SEND.

        XLEFB   0,CONS*2                ;Set byte pointer to terminal name.
        XLEFB   1,TMSG*2                ;Set byte pointer to ?SEND message.
        WLDAI   (CLIMSG-TMSG)*2!1S22,2  ;Message length and byte
                                        ;pointer flag.

        ?SEND                           ;Send message to terminal.
        WBR     ERROR                   ;?SEND error return.
        WSUB    2,2                     ;Done. Set flags for ?SEND
                                        ;normal return.
        WBR     BYE                     ;Goodbye.

ERROR:  NLDAI   ?RFEC!?RFCF!?RFER,2     ;Error flags: Error code is in
                                        ;AC0 (?RFEC), message is in
                                        ;CLI format (?RFCF), and
                                        ;should handle this as an
                                        ;error (?RFER).

BYE:    ?RETURN                         ;Return to CLI.
        WBR     ERROR                   ;?RETURN error return

NL:     .TXT    "<12>"                  ;Put each name on a new line.

;?SEND terminal name and message.  A .NOLOC 1 follows.

CONS:   .TXT    "@CONSOLE"              ;Use generic name.

TMSG:   .TXT    "All filenames written to line printer. Bye."
        .NOLOC  0                       ;Resume listing all.

;?GTMES packet to get directory name from CLI.

CLIMSG: .BLK    ?GTLN                   ;Allocate enough space for
                                        ;packet.

        .LOC    CLIMSG+?GREQ            ;Request type.
        .WORD   ?GARG                   ;Put argument in ?GRES only.

        .LOC    CLIMSG+?GNUM
        .WORD   1                       ;Argument 1 is directory name
                                        ;argument 0 is program name).
```

```
        .LOC    CLIMSG+?GRES            ;Set byte pointer to receive
                                        ;buffer.
        .DWORD  DIRNAME*2               ;Set byte pointer to directory
                                        ;name buffer (DIRNAME).

        .LOC    CLIMSG+GTLN             ;End of packet.

DIRNAME:.BLK    50.                     ;Directory name buffer.

;?GOPEN packet (needed for directory).

DIR:    .BLK    ?OPLT                   ;Allocate enough space for
                                        ;packet.
        .LOC    DIR+?OPLT               ;End of packet.

;?GNFN packet to get next filename.

GNAME:  .BLK    ?NFLN                   ;Allocate enough space for
                                        ;packet.

        .LOC    GNAME+?NFKY             ;AOS/VS uses this after first
                                        ;call.
        .DWORD  0                       ;Set to 0 for first call.

        .LOC    GNAME+?NFNM

        .DWORD  FNAME*2                 ;Set byte pointer to filename
                                        ;receive buffer.

        .LOC    GNAME+?NFTP
        .DWORD  -1                      ;There is no template (default
                                        ;is -1).

        .LOC    GNAME+?NFLN             ;End of packet.

FNAME:  .BLK    16.                     ;Area of receive filenames.

;?OPEN and I/O packet for line-printer output file.

LINEP:  .BLK    ?IBLT                   ;Allocate enough space for
                                        ;packet.

        .LOC    FILE+?ICH
        .WORD   0                       ;AOS/VS assigns channel number.
        .LOC    FILE+?ISTI              ;File specifications.
        .WORD   ?ICRF!?RTDS!?OFOT       ;Change format (?ICRF) to
                                        ;data-sensitive records
                                        ;(?RDTS), and open for input
                                        ;and output (?OFIO).

        .LOC    FILE+?ISTO
        .WORD   0                       ;Default file type to ?FUDF,
                                        ;user data file.
```

```
        .LOC    FILE+?IMRS
        .WORD   -1                      ;Default physical block size
                                        ;to 2 Kbytes.

        .LOC    FILE+?IBAD
        .DWORD  FNAME*2                 ;Set Byte pointer to record
                                        ;I/O buffer.

        .LOC    FILE+?IRES              ;Density mode (for magnetic
                                        ;tapes only).
        .WORD   0                       ;Default to Density mode set
                                        ;during VSGEN procedure.

        .LOC    FILE+?IRCL
        .WORD   136.                    ;Record length is 136
                                        ;characters.

        .LOC    FILE+?IRLR              ;Number of bytes transferred.
        .WORD   0                       ;Only ?READ and ?WRITE use
                                        ;this.

        .LOC    FILE+?IRNW               ;Reserved.
        .WORD   0                       ;Set to 0.

        .LOC    FILE+?IRNH               ;Record number.
        .DWORD  0                       ;Only ?READ and ?WRITE use
                                        ;this.

        .LOC    FILE+?IFNP
        .DWORD  LPTNM*2                 ;Set byte pointer to pathname.

        .LOC    FILE+?IDEL              ;Delimiter table address.
        .DWORD  -1                      ;Use default delimiters: null,
                                        ;NEW LINE, form feed, and
                                        ;carriage return (default is
                                        ;-1).

        .LOC    FILE+?IBLT              ;End of packet.

LPTNM:  .TXT    "@LPT"                  ;Printer queue filename.

        .END    DLIST                   ;End of DLIST program.
```

Pipe File Sample Program (Fragment)

This program fragment creates and opens a pipe (PIPEFILE), and writes a record into the pipe.

```
        .TITLE          OPIPE
        .ENT            OPIPE
        .NREL
        ; This program, in file PIPE_PROG.SR, creates
        ; pipe file "PIPEFILE" and places a message
        ; in it.
```

```
OPIPE:  WSUB    0,0                     ;Zero out AC0
        WSUB    1,1                     ;Zero out AC1
        ?OPEN   IOPKT                   ;Open the pipe
        WBR     ERROR                   ;Report error to caller

        NLDAI   ?OFOT!?RTDS,0           ;Get new flags into ?ISTI word
        XNSTA   0,IOPKT+?ISTI           ;and store into packet
        ?WRITE                          ;Write to pipe
        WBR     ERROR

        ?CLOSE  IOPKT                   ;Close pipe file
        WBR     ERROR

        WSUB    2,2
        WBR     BYE                     ;Take normal return to caller


        ERROR:  WLDAI   ?RFEC!?RFCF!?RFER,2  ;Error flags: Error code is in
                                             ;AC0 (?RFEC), message is in CLI
                                             ;format (?RFCF), and caller should
                                             ;handle this as an error (?RFER).
BYE:    ?RETURN
        WBR     ERROR

IOPKT:
        .BLK    ?IBLT
        .LOC    IOPKT+?ICH              ;Channel number
        .WORD   0

        .LOC    IOPKT+?ISTI             ;Flag word
        .WORD   ?OFCR!?OFCE!?OFOT!?RTDS ;Create pipe, open for
                                        ;output, data-sensitive
                                        ;record format
        .LOC    IOPKT+?ISTO             ;Specify pipe file type
        .WORD   ?FPIP

        .LOC    IOPKT+?IMRS             ;Pipe size
        .WORD   -1                      ;Default length (4096.)

        .LOC    IOPKT+?IBAD             ;BP -> (I/O buffer)
        .DWORD  WBUF*2

        .LOC    IOPKT+?IRES             ;Reserved
        .WORD   0

        .LOC    IOPKT+?IRCL             ;Record length
        .WORD   136.

        .LOC    IOPKT+?IRLR             ;# bytes transferred
        .WORD   0

.LOC    IOPKT+?IRNW                     ;Reserved
        .WORD   0
```

```
.LOC    IOPKT+?IRNH                 ;Record #
        .DWORD  0

        .LOC    IOPKT+?IFNP         ;BP -> (filename)
        .DWORD  FNAME*2

        .LOC    IOPKT+?IDEL         ;A(delimiter table)
        .DWORD  -1                  ;Default table

        .LOC    IOPKT+?EPIP         ;A(pipe extension)
        .DWORD  1SO+PIPEPKT         ;Set validity bit also

PIPEPKT:
        .LOC    PIPEPKT+?PIRV       ;Packet revision #
        .WORD   ?PKRO               ;Revision

        .LOC    PIPEPKT+?PIRS       ;Reserved
        .WORD   0

        .LOC    PIPEPKT+?PIFG       ;Flags word
        .DWORD  ?PNVR               ;Never pend

        .LOC    PIPEPKT+?PITI       ;Reserved
        .WORD   0

        .LOC    PIPEPKT+?PIPD       ;Reserved
        .DWORD  0

FNAME:
        .TXT    "PIPEFILE"          ;Name of file

WBUF:
        .BLK    136.                ;I/O buffer
        .LOC    WBUF
        .TXT    "This is record 1 of 1.<12>"

        .END    OPIPE
```

End of Chapter

Chapter 6
Windowing

You use the following system calls to support windowing applications:

?WINDOW	Create or manipulate a window; return information about a window.
?PTRDEVICE	Control pointer device I/O and pointer appearance.
?GRAPHICS	Manipulate information in graphics windows.

Each of these system calls is multifunctional.

On certain terminals, AOS/VS supports windowing. This chapter describes AOS/VS windowing and the system calls you can use to support windowing applications.

What Is Windowing?

Windowing lets you view and run multiple processes simultaneously on a single terminal. Each process runs independently within its own window.

Under AOS/VS, a *window* is a rectangular portion of a physical screen that has been attached to a *virtual terminal.* (A virtual terminal is the illusion of a terminal, created by software. Other software products also make use of virtual terminals; usually, each product has its own software and its own special type of virtual terminal. We discuss the windowing virtual terminal in more detail below.) Usually, the window has a *border* that separates it from the rest of the screen. Within the window border is the virtual terminal, which usually has been passed to a process.

A windowing terminal can display multiple windows simultaneously, each of which is attached to a separate virtual terminal. Figure 6–1 shows a physical terminal with two windows on it. In this figure, each window has been attached to a separate virtual terminal, which has been passed to a separate application program. Each program performs I/O independently on its own virtual terminal, just as if that terminal were a real physical terminal. AOS/VS manages the I/O among the competing virtual terminals on each physical terminal.

Windowing Terminals

AOS/VS supports windowing on pixel-mapped terminals attached to DS series workstations that run the GIS II graphics instruction set.

During system initialization, AOS/VS creates a terminal (console) file in the :PER directory for each pixel-mapped terminal attached to the system (just as it does for normal terminals). Terminal files for pixel-mapped terminals have names of the format "@PMAPn", where n is a unique integer that identifies the terminal. Pixel-mapped terminal files are also distinguishable from regular terminal files because they have the ?BMDEV (bit-mapped device) characteristic set.

You can perform only a limited set of system calls directly on a pixel-mapped terminal. They are

?ASSIGN	Assign a device to a process.
?DEASSIGN	Remove the channel assignment to a device.
?GCHR	Get device characteristics.
?GECHR	Get extended device characteristics.
?SCHR	Set device characteristics.
?SECHR	Set extended device characteristics.

In addition, you can use the ?WINDOW call, function code ?WIN_CREATE_WINDOW, to create a window on that terminal. Once a window exists on the terminal, you can communicate with that window just as you would communicate with a normal terminal.

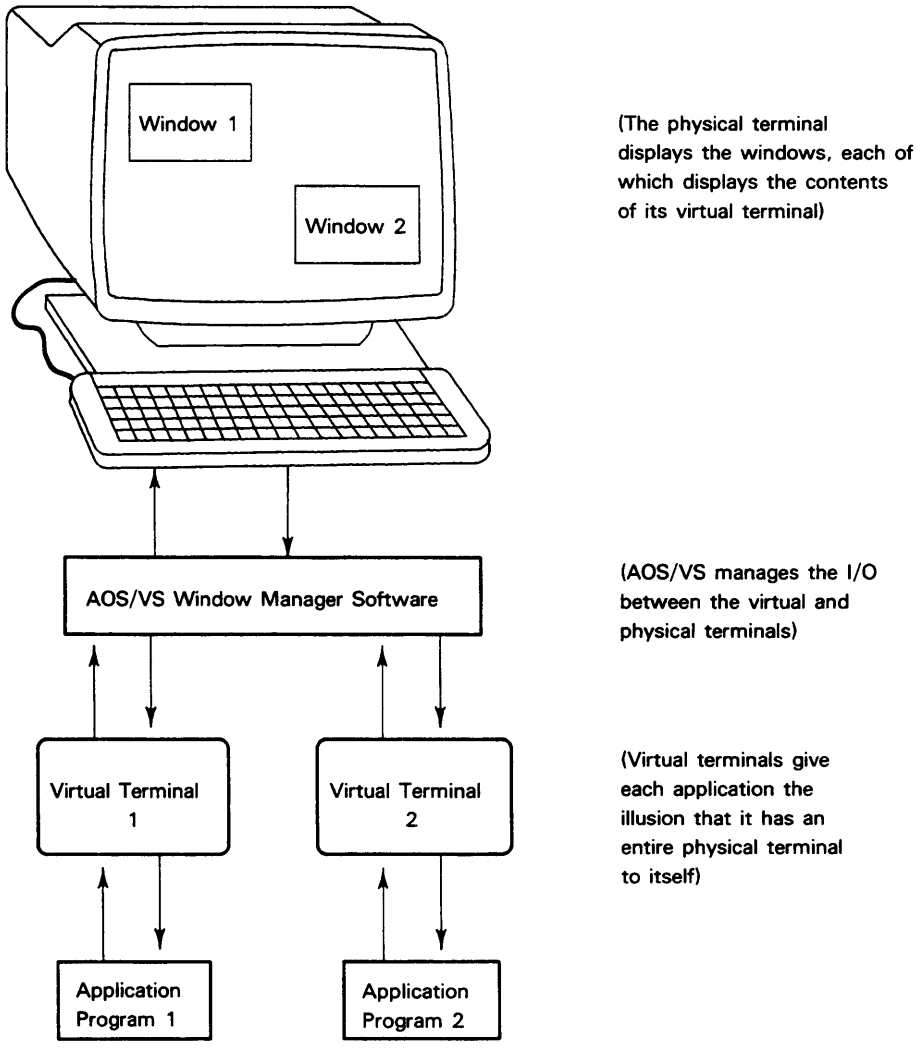


Figure 6–1. Multiple Windows on a Terminal

Window Pathnames

Every window has a unique *window pathname* in the form

```
@PMAPn:windowname
```

where

@PMAPn is the name of the pixel-mapped terminal that contains the window.

windowname is the name of the window.

When you create a window, you give it a name that is unique among windows on that terminal. The window name can be up to 31 characters long; restrictions on window names are identical to those for filenames.

Referring to a Window

When you issue windowing system calls, you can specify the target window by one of the following.

Pathname	The pathname consists of the terminal name plus the window name. (Note that there is some file system overhead involved in resolving a pathname.)
Window ID	When you create a window, AOS/VS returns the window's ID number.
Channel number	When you ?OPEN a window, AOS/VS returns the window's channel number.

The windowing system calls are ?WINDOW, ?GRAPHICS, and ?PTRDEVICE; we describe these calls later in this chapter. Most functions of these calls let you refer to the window through its pathname, ID, or channel number; a few require the window pathname.

Nonwindowing system calls treat windows as terminals; when you issue a nonwindowing system call on a window, refer to the window using either its pathname or channel number (whichever is appropriate to the call). For example, when you issue the ?OPEN call to open a window, use the window pathname (just as you would use @CONxx when opening a terminal). When issuing a ?READ or ?WRITE call to perform I/O on a window, use the window's channel number, just as you would for a terminal.

If a window is passed to a son process as the input, output, data, or console file, then the son process can refer to that window using the generic filenames @INPUT, @OUTPUT, @DATA, or @CONSOLE, respectively.

The Window Title

A window may also have a *window title,* which is a character string that's associated with the window, and can be displayed in the window's border. The window title is primarily used to tell the user what is running in the window. Your program cannot use a window title to refer to a window.

Window Types

There are two types of windows: character windows and graphics windows.

A *character window* is a window that has a virtual terminal similar to a D460 terminal. That is, the screen inside a character window looks and acts like the screen of a D460, both to the application program and to the user. A process running in a character window

does not need to be aware that it's running in a window instead of on a real D460; most programs that can run on a D460 terminal can run unchanged in a character window. (The virtual terminal has its own model ID. Since the virtual terminal does not support D460 local functions, such as compressed display and smooth scrolling, your program may want to check the model ID before attempting to use such functions. For a complete description of a character window's virtual terminal, see the section in this chapter entitled "Sending Output to a Character Window".)

In general, you perform I/O on character windows just as you would to any terminal (using the ?OPEN, ?READ, ?WRITE, and ?CLOSE calls).

A *graphics window* is a window specially designed to support high-performance graphics. While running in a windowing environment, your graphics program can write to the screen using graphics instructions, and receive input from the keyboard and the pointer device. Unlike a character window, the virtual terminal attached to a graphics window does not resemble any existing terminal. Therefore, to run in a graphics window, most existing programs need to be modified somewhat.

To get input from graphics windows, you can use ordinary ?OPEN and ?READ calls; however, your application must echo keyboard input itself, since AOS/VS cannot write characters to a graphics window.

To open a window for graphics output, you must use the ?GRAPHICS call; you cannot use the ?OPEN call. To write output to a graphics window, you must use the GIS II graphics instruction set; you cannot issue a ?WRITE call to a graphics window.

For details on graphics windows, see the section in this chapter entitled "Sending Output to a Graphics Window".

Window Characteristics

In general, you can treat windows like special terminals.

- For nonwindowing system calls, the ownership and access control rules for windows are identical to those for terminals. Exceptions for windowing system calls are noted later, when we describe each system call.
- The steps for communicating with a window are similar to those for communicating with a terminal (assign, open, read/write, and then close and remove its channel assignment.) We describe the details under "Setting Up Windows" below.
- Unlike terminals, windows have a Permanence attribute just as files do. You cannot delete a window that has Permanence on. To set the Permanence attribute of a window on or off, use the ?WIN_PERMANENCE_ON or ?WIN_PERMANENCE_OFF function code of ?WINDOW.

View Ports and Scan Ports

In AOS/VS windowing, a window on the physical screen can display part or all of the data that's currently on the virtual terminal, as shown in Figure 6–2.

In Figure 6–2, the physical screen contains a window with the title "CLI." Inside this window is the portion of the physical screen that has been attached to a virtual terminal; that portion of the screen is called the *view port*. The view port does not include the window's border or title.

The view port is attached to a rectangular section of the virtual screen; this section of the virtual screen is called the *scan port*. The view port (on the physical screen) displays only

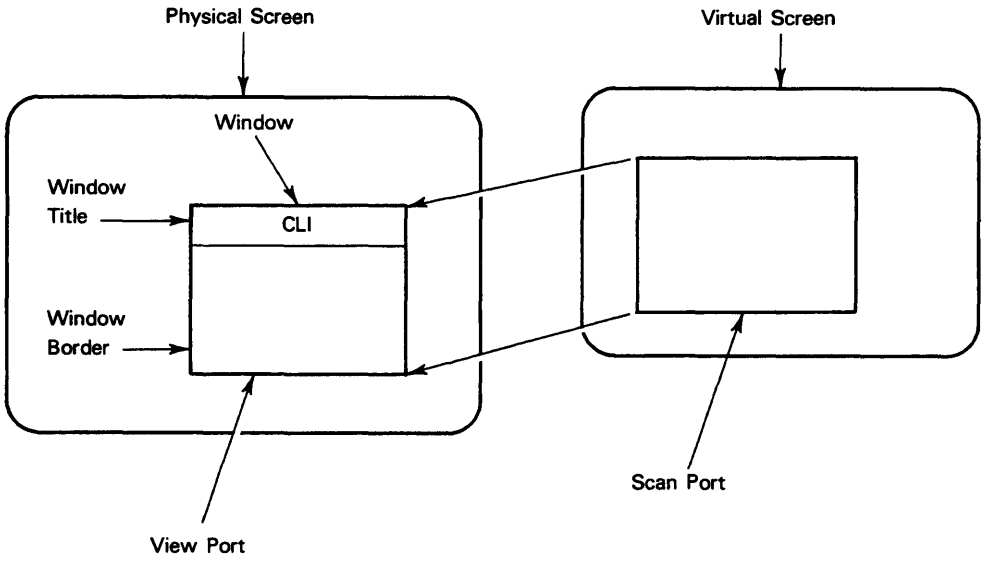


Figure 6–2. The View Port and the Scan Port

the data contained within the scan port (on the virtual screen). In Figure 6–2, the scan port contains only a subset of the data that's actually on the virtual screen; likewise, the view port shows only that data.

Using the View and Scan Ports to Change a Window

Your program can change the location or size of a window, or scroll data in a window, by altering its view or scan ports. (Use the ?WINDOW call, function code ?WIN_DEFINE_PORTS, which we describe later.)

To move the window on the physical screen, you change the coordinates of the origin (upper left corner) of the view port.

To show a different part of the virtual screen, you change the coordinates of the scan port's origin. Different data will then appear in the view port.

To change the size of the window on the physical screen, you change the width and/or height of the scan port (on the virtual screen). The size of the view port will automatically change to match that of the scan port.

You can perform more than one change at a time.

Window Overlap

When there is more than one window on a physical screen, the windows may *overlap* each other. When this happens, the front window covers all or part of the data in the view port(s) of the window(s) behind it. Figure 6–3 depicts a physical screen with two windows overlapping, and shows each window's respective virtual screen.

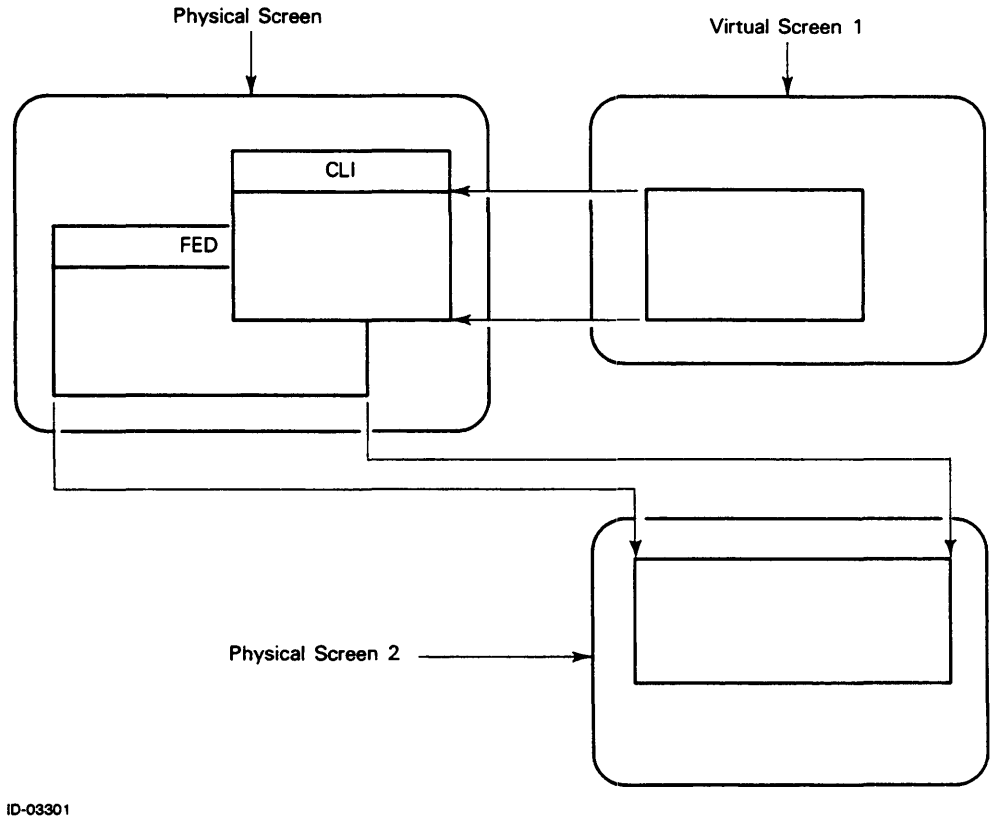
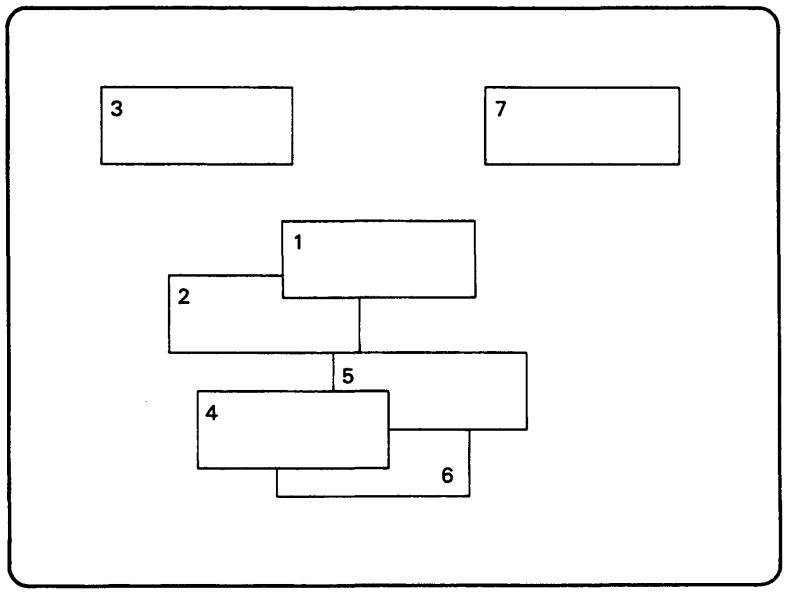


Figure 6–3. Overlapping Windows

Window Priorities

AOS/VS assigns a unique window priority to each of the windows on a physical screen. The window priority determines the front-to-back order of the windows: where windows overlap, the window with the higher priority appears in front of any window(s) with lower priority. Figure 6–4 represents a screen on which several windows overlap. We show each window's priority so that you can see how window priority determines which windows appear in front.

Notice that when windows do not overlap, window priority does not affect how the window appears on the screen. Thus, in the example above, the window with the lowest priority, Window 7, is completely unobscured, while Window 2 (the second-highest priority window) is partially covered by Window 1.



ID-03277

Figure 6–4. Window Priorities

Unlike block I/O, physical block I/O has

- 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.)

Modified Sector I/O

AOS/VS supports modified sector I/O to disks that support the hardware feature (354– and 592–megabyte "DPJ" disks). To perform modified sector I/O, the disk file must be opened using the appropriate options with the ?GOPEN call. You can only perform modified sector I/O by issuing the ?BLKIO call with the physical block I/O option set.

You use modified sector I/O to transfer *only* the disk blocks that have been modified. Modified sector I/O transfers both the modified disk blocks and a bit map of the modified blocks. For more detail on modified sector I/O functionality, refer to the *Programmer's Reference Series: Models 6236/6237 and 6239/6240 Disk Subsystems* manual.

Controlling File Access Through the AOS/VS Lock Manager

The AOS/VS system allows cooperating processes to control read or write access to whole files, or to one or more file *elements*, by locking them. The cooperating processes determine the file elements that can be locked by

- Giving each file element an identifying 32–bit number.
- Specifying the data objects to which each of the identifying 32–bit numbers refer. A file element can represent *any* data object.

By locking files and file elements, cooperating processes can control access — and prevent interprocess collision — when reading or writing to critical data objects, such as large databases. The cooperating processes *cannot* lock generic files or IPC–type files.

To lock a file or file elements, a process must issue the ?FLOCK system call. The ?FLOCK system call requests a lock from the AOS/VS lock manager. If the lock manager grants the lock request — we describe later why it may or may not — the process can then read or write to the file or file elements. When the process no longer needs access to the file or file elements, it can issue the ?FUNLOCK system call to release the lock.

A process can request one of two types of locks:

Exclusive lock — Allows *only* the locking process to write to the file or file elements.

Shared lock — Allows all processes to read the file or file elements; *no* cooperating process can write to the locked file or file elements.

The process requesting the lock can specify whether it wants to lock the whole file or one or more elements in the file. The AOS/VS lock manager allows a process to lock a file element under the following conditions.

Lock Status of File Element	Exclusive Lock Allowed	Shared Lock Allowed
Unlocked	Yes	Yes
Shared lock	No	Yes
Exclusive lock	No	No

The AOS/VS lock manager will allow you to lock a whole file *only* when there are no locked elements in the file. The AOS/VS lock manager will *not* allow you to lock a directory.

In the ?FLOCK system call, you can also specify what to do if the file or file elements are already locked. If the file or file elements are locked, ?FLOCK can

- Return an error.
- Wait for the release of the file or file elements by the process that has currently locked them.

  If there are multiple processes waiting for a lock to be released, the AOS/VS lock manager will give them their locks in FIFO (first in, first out) order.

To lock a file, or the elements in a file, the file must be open; any process that has sufficient privileges to open the file can lock the file or its elements. In the ?FLCHN offset of the ?FLOCK packet, you must give the channel number of the open file.

If more than one process has opened the file, or a single process opened it multiple times, you can give any one of the file's channel numbers in the ?FLOCK packet. The AOS/VS lock manager maintains the lock status of the file and its elements independently of the file's channel numbers: regardless of which channel number you give in the ?FLOCK packet, the lock status of the file and its elements are the same.

If a process task closes a channel, the AOS/VS lock manager will release all locks that used that channel. The AOS/VS lock manager will also return an error to any calling task in the same process that has a pending ?FLOCK system call that uses that channel.

Window Groups

A *window group* is a collection of windows that AOS/VS maintains at consecutive window priorities. Every window belongs to a window group, even if it is the only window in that group. Figure 6–5 shows how the windows in Figure 6–4 might be grouped.

Windows in a group can appear anywhere on the physical screen and can overlap each other or the windows in other groups.

Each group of windows can be considered to have a priority relative to other groups. Since Group A's windows have window priorities of 1 and 2, its windows will appear in front of windows in all the other groups. Likewise, each of Group C's windows (4, 5, and 6) have higher priorities than Group D's window (7), so Group C can obscure Group D. Since none of Group C's windows have priorities higher than 4, Groups A and B (windows 1, 2, and 3) will appear in front of Group C.

When you create a window (using the ?WINDOW call, function code ?WIN_CREATE_WINDOW), you specify whether you want it to join an existing group, or become the first window in a new group.

Windows in a group can all belong to a single process, or one or more windows in a group can belong to different processes. A single process can own windows that belong to different groups.

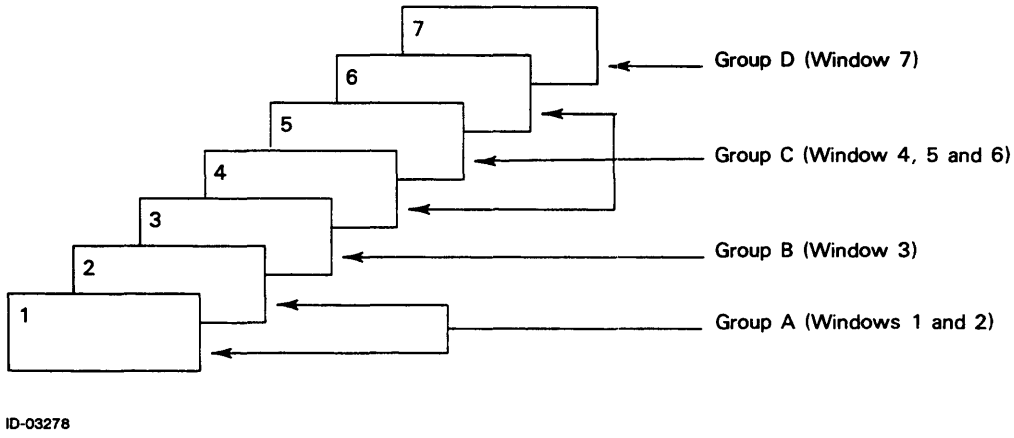


Figure 6–5. Grouping Windows

Changing Window and Group Priorities

You can alter a window's priority, relative to other windows in its group, by using two functions of the ?WINDOW call. Issuing ?WINDOW with the ?WIN_UPFRONT_WINDOW function code brings a window to the front of its group (making it the highest priority). Issuing ?WINDOW with function code ?WIN_OUTBACK_WINDOW sends a window to the back of its group (making it the lowest priority). If the window is the only member of its group, these calls have no effect.

To bring a group of windows to the front of the screen (making it the highest priority, relative to other groups), issue the ?WINDOW call, function code ?WIN_UPFRONT_GROUP. To send a group behind other groups on the screen (making it the lowest priority), use the function code ?WIN_OUTBACK_GROUP. If there is only one group of windows on the physical terminal, these calls have no effect.

Changing Window and Group Visibility

Your program can determine whether or not a window is visible to the user. When you *hide* a window, it is invisible to the user, and cannot receive user input. However, you can still perform most windowing operations on a hidden window. Hiding a window does not affect its window priority.

To make a particular window invisible to the user, issue ?WINDOW with function code ?WIN_HIDE_WINDOW; to make a hidden window reappear, issue ?WINDOW with function code ?WIN_UNHIDE_WINDOW.

You can also hide an entire group of windows; to do so, issue ?WINDOW with function code ?WIN_HIDE_GROUP. To make a hidden group visible again, issue ?WINDOW with function code ?WIN_UNHIDE_GROUP.

Rules for Window/Group Visibility

AOS/VS uses the following set of rules to determine window and group visibility.

- When you hide an individual window, it remains invisible, regardless of the state of its group.
- When you make an individual window visible again (?WIN_UNHIDE_WINDOW), that window reappears unless its group is hidden.
- When you hide a window group, every window in that group becomes invisible.
- When you make a group visible again (?WIN_UNHIDE_GROUP), its windows reappear, except for those that are still hidden individually.

The Active Group

Although multiple groups of windows can appear on the screen at once, only one group at a time can be the *active group*. The active group is the highest priority group that has at least one visible window. The active group receives all keyboard and relevant pointer input (see the "Getting Input From a Window" section, later in this chapter, for more details). Windows in the active group are in the foreground (they aren't obscured by windows of other groups); however, windows in the active group can overlap other windows in that group.

Only the active group can receive input from the user. However, all groups on the screen compete for system resources according to the priorities of the processes involved. This

means that a process whose window is not in the active group can still perform calculations, get input from files, and write to its window.

AOS/VS changes the appearance of windows to show whether or not they belong to the active group. The borders of windows in the active group are thick; borders on other windows are thin. For windows with intelligent borders, border symbols and elevators appear only on windows in the active group.

When Should You Group Windows?

Grouping windows is useful when you want several windows to be treated as a set. AOS/VS provides a menu-driven user interface program, DG/VIEW, which lets users manipulate windows on the screen (we describe DG/VIEW later). With DG/VIEW, users can make a group of windows the active group; however, they cannot change the priority of windows within a group. Users can also hide groups (make them invisible) and suspend them, but cannot perform these actions on individual windows.

You should put windows in the same group if you want the user to make them active, visible, or invisible at the same time. For example, suppose you have two associated windows: one window displays a list of objects, and the other window displays a list of actions you can perform on those objects. You would probably want DG/VIEW to treat the two windows as a group: if the user hides one of the windows, the other should also disappear, since in this case neither window is much use without the other.

Setting Up a Window

To set up a window so that you can manipulate it (move, size, etc.) or perform I/O to it, you must perform the following sequence of operations.

1. Create the window.
2. Assign the window to a process.
3. Adjust the window's appearance and priority, and make it visible to the user.
4. Open the window for I/O.

We explain each step in detail below.

Creating a Window

To create a window on a terminal, issue the ?WINDOW system call, function code ?WIN_CREATE_WINDOW.

In order to create a window, you must have either control access to the physical terminal (that is, be able to issue a ?SECHR call), or control access to an existing window on that terminal. In the main ?WINDOW packet, you specify the terminal or window to which you have access. The process differs slightly depending on which you specify.

Specifying a Physical Terminal

If you have control access to a physical terminal, you can create windows on that terminal by specifying the terminal's pathname in the main ?WINDOW packet. Each window you create using this method will begin a new window group; you cannot use this method to add a new window to an existing group.

In the ?WIN_CREATE_WINDOW subpacket, you must indicate

- That you are specifying a physical terminal in the main packet.
- That the new window will begin a new group.
- A name for the new window (the name forms part of the window pathname).

Specifying an Existing Window

If you have control access to an existing window on a terminal, you can create windows on that terminal by specifying the existing window's pathname, channel number or window ID in the main ?WINDOW packet. When you create a window using this method, the new window can either begin a new window group, or it can join the group of the existing window you specified.

In the ?WIN_CREATE_WINDOW subpacket, you must indicate

- That you are specifying a window in the main packet.
- Whether you want the new window to begin a new group or join the existing window's group.
- A name for the new window (the name forms part of the window pathname).

Optional ?WIN_CREATE_WINDOW Information

In the ?WIN_CREATE_WINDOW subpacket, you can also specify

- The type of window you are creating (character or graphics).
- The pixel depth (number of bits per pixel), for graphics windows only.
- The size of the virtual terminal for the window you are creating.
- The window title (which appears in the window border).
- The size of the window's input buffer.

AOS/VS creates the window according to your specifications in the ?WIN_CREATE_WINDOW subpacket, and returns the window ID of the newly created window.

Assigning a Window to a Process

When you create a window, ?WIN_CREATE_WINDOW automatically assigns the new window to your process.

To assign an existing window to your process, load AC0 with a byte pointer to the window pathname, and issue the ?ASSIGN call.

To assign an existing window to a new process, issue the ?PROC call from the process running in that window, and pass the window to the new process as that process's @CONSOLE file. This creates a new process, and assigns the window to the newly created process.

When you pass a window to a son process as that process's @CONSOLE file, AOS/VS saves certain window characteristics. When the son process terminates, AOS/VS restores the window to its previous state, and returns it to the parent process.

AOS/VS saves the following window characteristics.

- Window title.
- Whether or not the window is visible.
- Whether or not the window is suspended.
- Whether or not the keyboard is enabled for this window.
- The set of pointer events you have asked to receive for this window.
- How far the pointer device must move before AOS/VS reports it.
- The shape of the pointer, and whether or not it is visible.
- The color/grey settings in the palette (for graphics windows only).
- User interface parameters, such as the type of border you are using.

After you assign a window to a process, that process can use windowing system calls to manipulate the window. For example, the process can move the window, resize it, or customize its appearance. (For more information, see the section "Manipulating a Window" later in this chapter.)

Adjusting the Priority of a Newly Created Window

When you create a window, AOS/VS gives it the highest window priority within its group. If you don't want the new window to have the highest priority within its group, use the ?WIN_UPFRONT_WINDOW and ?WIN_OUTBACK_WINDOW functions of ?WINDOW to adjust the window hierarchy within that group.

For example, if you want the new window to have the lowest priority, you would use the ?WIN_OUTBACK_WINDOW function to send that window to the back of its group. To make the new window second in priority, use the ?WIN_UPFRONT_WINDOW function to make another window the highest priority in that group.

Adjusting the Appearance of a Newly Created Window

A newly created window is still hidden; it is not visible to the user. This allows you to change the window's appearance before you display it. AOS/VS creates each window with a default position, size, and border type that may nor may not be appropriate to your application.

You can issue the ?WINDOW call, function code ?WIN_DEFINE_PORTS, to change:

- The window's position on the terminal.
- The size of the view port.
- The position of the scan port on the virtual screen (what data appears).

You can issue the ?WINDOW call, function code ?WIN_SET_USER_INTERFACE, to determine:

- The border type (line, title, intelligent, or none).

- For intelligent borders, whether or not to maintain scrolling elevators.
- Whether or not the user is allowed to move, resize, or scroll the window.
- Whether or not the user can move the window borders off the screen.

Making a Window Visible

After you create a window, adjust its window priority, and determine its appearance on the screen, you are ready to make it visible to the user. To do so, issue the ?WINDOW call with function code ?WIN_UNHIDE_WINDOW. AOS/VS then makes the window visible in its current position.

Opening a Channel to a Window

After you open a window, you can write to it, read from it, and manipulate it via windowing system calls. The user can also manipulate it via DG/VIEW commands.

The process for opening I/O channels differs for graphics and character windows. We explain each process below.

Opening a Character Window

To open a character window for I/O, issue a ?OPEN call, just as you would for a physical terminal, and specify the window's pathname. AOS/VS returns the channel number of the window. You can perform I/O over this channel as you would any I/O channel. You can also use the channel number to specify the window when issuing windowing system calls.

Like a terminal, a window can have multiple I/O channels open to it; just issue multiple ?OPEN calls.

For more detail on I/O to windows, see the "Getting Input From a Window" section later in this chapter.

Opening a Graphics Window

To open a graphics window for input, issue a ?OPEN call and specify the window's pathname. AOS/VS returns a channel number; you can issue ?READ calls to get input from this channel. (Your program must perform any keystroke echoing; AOS/VS cannot write characters to a graphics window). For more detail on getting input from a graphics window, see the "Getting Input From a Window" section later in this chapter.

To open a graphics window for output, issue a ?GRAPHICS call, function code ?GRAPH_OPEN_WINDOW_PIXELMAP. AOS/VS returns a pixel map ID, which you can use when sending graphics output to the window. You cannot issue ?WRITE calls to a graphics window; to write to a graphics window, you must use the GIS II graphics instructions.

You can have more than one pixel map ID open to a single graphics window; just issue ?GRAPH_OPEN_WINDOW_PIXELMAP multiple times.

For more detail on I/O to graphics windows, see the "Sending Output to a Graphics Window" section.

Manipulating a Window

Once a window exists and has been assigned to a process, both that process and the human user can manipulate the window. Manipulating a window means changing the window itself

— for example, changing its size, or its position on the screen. This differs from performing I/O on a window, in which information enters the window's input buffer or appears on the window's virtual screen without affecting the window itself.

How Does a Process Manipulate Windows?

A process manipulates windows by issuing the ?WINDOW system call. You can use the ?WINDOW system call to manipulate individual windows or groups of windows, and to create or list windows on a physical terminal.

You can perform the following ?WINDOW functions on a single window.

Function Code	What It Lets You Do
?WIN_DELETE_WINDOW	Delete a window.
?WIN_DEFINE_PORTS	Move or resize the view or scan port.
?WIN_UPFRONT_WINDOW	Make a window the highest priority in its group.
?WIN_OUTBACK_WINDOW	Make a window the lowest priority in its group.
?WIN_HIDE_WINDOW	Make a window invisible to the user.
?WIN_UNHIDE_WINDOW	Make a hidden window visible again.
?WIN_SUSPEND_WINDOW	Suspend output to a window.
?WIN_UNSUSPEND_WINDOW	Allow output to a suspended window.
?WIN_ENABLE_KEYBOARD	Allow keyboard input to a window.
?WIN_DISABLE_KEYBOARD	Prevent keyboard input from going to a window.
?WIN_PERMANENCE_ON	Set the permanence attribute on for a window.
?WIN_PERMANENCE_OFF	Set the permanence attribute off for a window.
?WIN_SET_USER_INTERFACE	Set the user interface for a window.
?WIN_GET_USER_INTERFACE	Return the status of a window's interface.
?WIN_SET_TITLE	Set the title of a window.
?WIN_GET_TITLE	Return the title of a window.
?WIN_GET_WINDOW_NAME	Return the window name of a window.
?WIN_GET_WINDOW_ID	Return the window ID of a window.
?WIN_WINDOW_STATUS	Return the status of a window.

You can perform the following ?WINDOW functions on a group of windows.

Function Code	What It Lets You Do
?WIN_UPFRONT_GROUP	Make a group the highest priority among groups.
?WIN_OUTBACK_GROUP	Make a group the lowest priority among groups.
?WIN_HIDE_GROUP	Make a group invisible to the user.
?WIN_UNHIDE_GROUP	Make a hidden group visible again.
?WIN_SUSPEND_GROUP	Suspend output to a group.
?WIN_UNSUSPEND_GROUP	Allow output to a suspended group.
?WIN_RETURN_GROUP_WINDOWS	Return the window IDs of all windows in a group.

You can perform the following ?WINDOW functions on a physical terminal.

Function Code	What It Lets You Do
?WIN_CREATE_WINDOW	Create a new window.
?WIN_RETURN_CONSOLE_WINDOWS	Return the window IDs of all windows on a terminal.

How Does the User Manipulate Windows?

Your program can create, manipulate, and delete windows by issuing windowing system calls. However, a user sitting at a terminal can't just issue system calls from the keyboard. To allow the user to run programs in windows and to manipulate the windows on a terminal, Data General provides the DG/VIEW windowing interface program.

The system manager can specify DG/VIEW as a user's initial program. Then, when the user logs on to the system, DG/VIEW displays a menu of programs. The user can start any or all of the listed programs, and can also edit the menu to add or remove programs.

When the user starts a program under DG/VIEW, that program runs as a son process of DG/VIEW. The program still has control over all its windows, but DG/VIEW allows the user limited control over the windows as well. Table 6–1 lists the actions that users can perform with DG/VIEW's menu-driven interface.

Table 6–1. Actions Users Can Perform with DG/VIEW

User-Visible Action	What It Really Does
Start an application in a window.	Creates a window; passes it to a process.
Scroll data within a window.	Changes the scan port's origin.
Move a window on the screen.	Changes the view port's origin.
Change the size of a window.	Changes the scan port's size.
Expand a window to its full size.	Changes scan port size to largest allowable size (virtual terminal size, or an arbitrary size specified by the program in a DG/VIEW information file.)

(continued)

Table 6-1. Actions Users Can Perform with DG/VIEW

User-Visible Action	What It Really Does
Return a window to previous size.	Returns scan port size to previous size.
Make an application invisible.	Hide the group of windows belonging to an application.
Cut a block of data from a window.	Remove a rectangular block of ASCII data from a window.
Copy data from a window.	Copies a rectangular block of ASCII data from a window.
Paste data to a window.	Inserts a block of ASCII data into the window's input data stream.
Get help on a window's contents.	Invokes the DG/VIEW help system.
Suspend and hide an application.	Hides and suspends output to the group of windows belonging to a process.
Switch to an application.	Makes the group of windows belonging to a process the active group.
Switch to next application.	Makes the group of windows with the next highest priority the active group.
Exit from an application.	Terminates a process, closes its window, removes its channel assignment and deletes the window.
Exit from DG/VIEW.	Terminates all processes running as sons of DG/VIEW; closes their windows, removes their channel assignments and deletes their windows. Then, terminates the user's DG/VIEW process.

(concluded)

Note that a user cannot explicitly create or delete a window. In addition, although the user can control the relative priorities of groups of windows (via "Switch"), only the applications can determine the front-to-back arrangement of windows within groups.

Your program can control some aspects of the user interface using the ?WINDOW call, function code ?WIN_SET_USER_INTERFACE. For example, you can determine whether the user is allowed to move or resize a particular window, and you can set the border type (line, title, or intelligent) for a particular window.

For more details on the DG/VIEW user interface, see *Using DG/VIEW.*

Getting Input from a Window

The user can send input to your application in either of two ways: by pressing keys on the keyboard, or by manipulating the pointer device. Both methods send data to the current window's input buffer.

AOS/VS inserts keyboard input into a window's input data stream as the user types on the keyboard. The *keyboard cursor* indicates which window is receiving keyboard input (typically, input is inserted at the cursor position, but your application can specify otherwise). The cursor cannot move outside of the current window, and is under the control of both

the application and the user. There is only one keyboard cursor on the screen; it can appear only within a window in the currently active group.

AOS/VS inserts pointer device input into a window's input data stream as the user manipulates a pointer device (such as a mouse or a light pen). The pointer device controls a *pointer* on the screen. There is only one pointer for the entire screen; it is completely under the user's control (the application cannot affect its position). When the user moves the pointer, or presses a button on the pointer device, AOS/VS generates a 16–byte sequence called a *pointer event*, and inserts this sequence into the input data stream. Each pointer event sequence tells you the type of action that took place, and gives the screen coordinates at which it occurred.

Opening a Window for Input

Before a process can receive input from a window, you must open a channel to that window. To open a window for input, issue a ?OPEN call and give the window's pathname; AOS/VS returns a channel number. For character windows, you can use this channel number for both input and output.

For graphics windows, you can use the channel number only to receive input. The channel number lets you receive both keyboard input and pointer events. (You cannot use a channel number to send output to a graphics window; all output to a graphics window is in the form of graphics instructions that you perform on a pixel map.)

NOTE: AOS/VS does not echo keystroke input to a graphics window. If you want the user's keystrokes to appear on the screen, your application must use graphics instructions to write the characters on the window's pixel map.

The Input Buffer

Every window has its own *input buffer*, just as a terminal does. When a window receives input data from the keyboard or the pointer device, AOS/VS inserts the data into the window's input data stream. From the data stream, the data enters the window's input buffer, where it remains until a process issues a ?READ call. The ?READ call removes data from the buffer in the order in which the data arrived.

Your application can read data from a window's input buffer at any time, as long as there is still data in the buffer (and as long as your process has access to that window).

When you create a window, you can specify the size of the input buffer. If you are planning to use pointer device input in a window, you may want to specify a larger buffer size, since pointer events can quickly fill a small buffer.

Input Focus

AOS/VS only sends data to the input buffers of windows in the currently active group. Within that group, one window at a time can receive keyboard input; also, one window at a time can receive pointer input. The input focus is the window whose input buffer is receiving new input.

AOS/VS determines the input focus based on window priority and visibility. There can be two input foci: one for keyboard events and one for pointer events.

Input Focus for Keyboard Input

AOS/VS sends keyboard input to the input buffer of one window at a time within the currently active group.

To be eligible to receive keyboard input, a window must be visible, be in the active group, and have keyboard input enabled (the default is to have keyboard input enabled). AOS/VS sends keyboard input to the highest priority eligible window in the currently active group.

If no windows in the active group are eligible to receive keyboard input, any keyboard input is ignored.

A process can control which window receives keyboard input by changing the priority or visibility of windows, or by enabling and disabling keyboard input for certain windows.

Input Focus for Pointer Events

AOS/VS sends pointer device input to the input buffer of one window at a time within the currently active group.

To be eligible to receive pointer device input, a window must be visible and in the active group, and you must have issued the ?PTRDEVICE call (function code ?PTRDEV_SET_EVENTS) for that window. This call tells AOS/VS which pointer events are relevant for that window; the default is for a window not to receive any pointer events.

AOS/VS sends pointer events to the input buffer of the window that currently contains the pointer. The pointer must be within the window's view port. As the pointer moves from window to window, AOS/VS changes the input focus for pointer device input.

If the pointer is not in an eligible window, pointer events are ignored.

A process can control which window receives pointer events by changing the visibility of windows, or by enabling and disabling pointer events for certain windows.

For example, if you wanted to allow pointer device input to go to only one window in a group, you could disable pointer events for all other windows in the group (via ?PTRDEVICE, function code ?PTRDEV_SET_EVENTS). Or, you could allow some types of pointer events (for example, button presses and releases) for certain windows, but not for others.

Controlling Input from the Keyboard

Your application can move the keyboard cursor from window to window (within a group) by changing the keyboard input focus. Within a window, you can control the position of the cursor by issuing cursor control commands, just as you would on a nonwindowing terminal.

If the input focus will rotate among several windows in a group, you may want those windows to remain eligible for keyboard input. To make a particular window the keyboard input focus, issue ?WINDOW with function code ?WIN_UPFRONT_WINDOW (to make that window the highest priority in its group). This approach also ensures that the entire window is visible while the user is entering input from the keyboard.

Sometimes, you may want to change the front-to-back order of windows in a group without affecting the keyboard input focus. To keep the keyboard input focus in a particular window, regardless of the window's priority, you should disable keyboard input for all windows except that window.

Two functions of ?WINDOW let you disable and re-enable keyboard input to a window. When you create a window, the default is for keyboard input to be enabled. To disable input to a window, issue ?WINDOW with function code ?WIN_DISABLE_KEYBOARD,

and specify that window in the main ?WINDOW packet. To re-enable keyboard input for a window, issue ?WINDOW with function code ?WIN_ENABLE_KEYBOARD; again, specify that window in the main ?WINDOW packet.

Controlling and Interpreting Input from a Pointer Device

A process cannot move the pointer from window to window as it can the keyboard cursor. The pointer is completely under the user's control: the input focus for pointer events changes as the user moves the pointer from window to window.

However, for each window, you can specify which pointer events you want to receive. For example, you could ask to be notified only when the user presses a button on the pointer device, or only when the user moves the pointer device. This gives you some measure of control over the input focus: if you don't want a window to receive any pointer device input, you can tell AOS/VS not to send it any pointer events at all.

You use the ?PTRDEVICE system call to control and interpret pointer device input. ?PTRDEVICE provides the following functions.

Function Code	What It Lets You Do
?PTRDEV_SET_EVENTS	Specify what events you want to receive.
?PTRDEV_SET_DELTA	Specify how far the pointer device must move before AOS/VS sends you a movement event.
?PTRDEV_LAST_EVENT	Return information about the last event performed on the pointer device.
?PTRDEV_SET_POINTER	Specify the shape of the pointer; enable/disable the pointer in a window.
?PTRDEV_GET_PTR_STATUS	Return the status of the pointer device.

Specifying the Pointer Events You Want

By default, a window does not receive any pointer events at all. To receive pointer events, you must issue ?PTRDEVICE, function code ?PTRDEV_SET_EVENTS, and specify the events you want by setting the appropriate flags in the ?PTRDEV_SET_EVENTS.EVT offset. The flags are

Parameter	Pointer Event You Will Receive
?PTRDEV_SET_EVTS.EVTS.MOVEMENT	Movement event
?PTRDEV_SET_EVTS.EVTS.BTN_DOWN	Button down event
?PTRDEV_SET_EVTS.EVTS.BTN_UP	Button up event
?PTRDEV_SET_EVTS.EVTS.WINDOW	Window events (lets you receive the following four types of events: Enter window event Exit window event Window activation event Window deactivation event)
?PTRDEV_SET_EVTS.EVTS.DBL_CLICK	Double-click event

Once you ask to receive a particular pointer event, AOS/VS inserts information in the window's input stream whenever the event occurs, and that window is eligible to become the input focus for pointer events.

Below, we explain each type of pointer event in detail.

Movement Event — A *movement event* is a 16–byte pointer event sequence that describes a pointer movement. AOS/VS inserts movement events in a window's input stream if you ask to receive movement events for this window, and if any of the following actions occur.

- The pointer moves within the window.
- A window event (enter, exit, activate, or deactivate window) occurs, and you did not ask to receive window events.

The pointer event's 16–byte sequence tells you the location of the pointer at the time AOS/VS generated the event.

By default, if you have requested movement events, AOS/VS sends you a pointer event as often as possible (whenever the pointer moves at all). If you don't want to be notified that often, you can specify how far the pointer must move before AOS/VS tells you about it (issue ?PTRDEVICE with function code ?PTRDEV_SET_DELTA).

Button Down Event — A *button down event* is a 16–byte pointer event sequence that describes a button press. AOS/VS inserts a button down event in a window's input stream if you ask to receive button down events for this window, and if the user presses a button on the pointer device while the pointer is within the window.

The pointer event's 16–byte sequence tells you which button was pressed, and the location of the pointer at the time.

NOTE: If you ask to receive button down events, you must also set flags in offset ?PTRDEV_SET_EVTS.BTNS to indicate which button(s) you are interested in. AOS/VS will then send you all button down events that involve the button(s) you indicate.

Button Up Event — A *button up event* is a 16–byte pointer event sequence that describes a button release. AOS/VS inserts a button up event in a window's input stream if you ask to receive button up events for this window, and if the user releases a button on the pointer device while the pointer is within the window.

The pointer event's 16–byte sequence tells you which button was released, and the location of the pointer at the time.

NOTE: If you ask to receive button up events, you must also set flags in offset ?PTRDEV_SET_EVTS.BTNS to indicate which button(s) you are interested in. AOS/VS will then send you all button up events that involve the button(s) you indicate.

Window Events — A *window event* is a 16–byte pointer event sequence that describes window activation and deactivation. If you ask to be sent window events, AOS/VS sends you the following four events as they occur.

Event	Meaning
Enter window	The window has become the pointer input focus because the pointer has entered the window by crossing its edge.

Exit window — The window is no longer the pointer input focus because the pointer has left the window by crossing its edge.

Activate window — The window has become the pointer input focus, but the pointer has not moved. This might happen if you brought this window or its group to the front. It might also happen if you hid or suspended the currently active window or group, and this window or its group were the next in priority. (If the window becomes active, but the pointer is not within it, AOS/VS will not generate an event.)

Deactivate window — This window is no longer the pointer input focus, but the pointer has not moved. This might happen if you brought another window or group to the front, or if you hid or suspended this window or its group.

The pointer event's 16–byte sequence tells you the location of the pointer at the time of the event.

Double-Click Event — A *double-click event* is a 16–byte pointer event sequence that describes a button double-click. AOS/VS inserts a double-click event in a window's input stream if you ask to receive double-click events for this window, and if the user double-clicks (presses and releases twice quickly) a button on the pointer device while the pointer is within the window.

The pointer event's 16–byte sequence tells you which button was clicked, and the location of the pointer at the time.

NOTE: If you ask to receive button down events, you must also set flags in offset ?PTRDEV_SET_EVTS.BTNS to indicate which button(s) you are interested in. AOS/VS will then send you all button down events that involve the button(s) you indicate.

Pointer Event Format

AOS/VS inserts pointer events into a window's input stream just as it inserts keystrokes. You can then read these pointer events just as you would read normal keystrokes, by issuing the ?READ system call. Each pointer event is a printable 16–byte sequence in the following format.

<036><157> <event> <button> <X-coordinate> <Y-coordinate>

Where

<036><157> is the header sequence, which identifies the beginning of a pointer event.

<event> is an ASCII value that indicates which event occurred. It can be one of the following eight values.

<060> — movement event

<061> — button down event

<062> — button up event

<063> — enter window event

<064> — exit window event

<065> — window activation event

<066> — window deactivation event

<067> — double-click event

<button> is an ASCII value that indicates which button was involved in a button up, button down, or double-click event. It can be one of the following four values.

<060> — the event was not a button up, button down, or double-click event

<061> — Button 1

<062> — Button 2

<063> — Button 3

<X-coordinate> is a set of six ASCII values. It specifies the X coordinate of the pointer at the time of the event. Every pointer event includes an X coordinate.

<Y-coordinate> is a set of six ASCII values. It specifies the Y coordinate of the pointer at the time of the event. Every pointer event includes a Y coordinate.

AOS/VS translates the X and Y coordinates into printable ASCII characters by separating each 32–bit coordinate into five groups of 6 bits each, with 2 bits left over (a total of six groups). AOS/VS then adds an octal 060 to each group (to ensure that each character is printable). The resulting 6–byte sequence becomes part of the pointer event.

For example, suppose the pointer is at the location (500., 1000.); in octal, this is (764, 1750). The 32–bit representation of these coordinates is

```
X:  00  000000  000000  000000  000111  110100
Y:  00  000000  000000  000000  001111  101000
```

AOS/VS then adds octal 060 to each group of bits to ensure printability. This results in the following octal values.

```
X:  <060>  <060>  <060>  <060>  <067>  <144>
Y:  <060>  <060>  <060>  <060>  <077>  <130>
```

These values become part of the pointer event's 16–byte sequence.

Translating Coordinate Values into Real Coordinates — In order to get the numerical coordinates of the pointer's location, you must translate backwards from the six ASCII values. To do so, perform the following steps for each coordinate you want to translate.

1. Allocate a 32–bit memory location for the coordinate.
2. Subtract octal 060 from each of the 6 bytes that make up the coordinate.
3. Construct a 32–bit coordinate value by concatenating the low-order 2 bits of the first byte with the low-order 6 bits of each of the other 5 bytes.
4. Store the resulting value in the previously allocated memory location.

After performing these steps for both the X and the Y coordinate values, the result will be a pair of numerical graphics coordinates that describe the position of the pointer.

Coordinates of the Pointer Location

AOS/VS gives you the coordinates of the pointer in pixel units. The coordinates are relative to the upper left corner of the virtual terminal. (Note that the upper left corner of the virtual terminal is not necessarily the same as the upper left corner of the window. If the window is smaller than the virtual terminal, its origin may differ.)

If you are using a character window, you'll have to convert the pixel coordinates to character coordinates in order to figure out what character row and column the pointer is on. To do so, divide the pixel coordinate by the number of pixels in a character's width or height. (To find out how many pixels there are in a character, issue a ?WINDOW call with function code?WIN_WINDOW_STATUS. This returns the scan port size in characters and the view port size in pixels. You then divide the view port size by the scan port size to get the number of pixels per character.)

Getting Pointer Event Information

Once you have requested a set of pointer events for a particular window, AOS/VS inserts events into the window's input buffer as they occur. There are three ways you can get pointer event information from the buffer.

- Issue a ?READ call on the window, and scan the characters the ?READ call returns.
- Issue a ?READ call on the window, and specify that you want pointer events to act as delimiters. The ?READ will then terminate when a pointer event occurs.
- Issue the ?PTRDEVICE call with function code ?PTRDEV_LAST_EVENT for information on the last event that occurred in the window.

Scanning for Pointer Events

One way to get pointer event information is simply to issue a ?READ call on the window's input buffer (specify the window's I/O channel number). AOS/VS returns the contents of the window's input buffer. You must then scan the contents of the buffer, character by character, for the characters <036><157> — the header characters that identify a pointer event. The 14 characters following the header sequence make up the rest of the event.

Pointer Events as Delimiters

You can request that AOS/VS treat pointer events as delimiters (much as you can for function keys). To receive pointer events as delimiters

1. Issue the ?SECHR (Set Extended Characteristics) call; set the ?TRPE (Terminate ?READs on Pointer Events) characteristic bit in the fifth characteristic word.
2. Issue a data-sensitive ?READ call. Supply an extended screen management packet in addition to the ?READ packet.

When the ?READ call terminates

1. Examine the bit ?ESPE in offset ?ESSE in the screen management packet. If the bit ?ESPE is set, the ?READ call terminated because a pointer event occurred.
2. If the data-sensitive ?READ terminated due to a pointer event, the last 16 characters in your ?READ buffer will be the 16-byte pointer event sequence.

The Last Pointer Event

You can get information about the most recent pointer event for a window by issuing the ?PTRDEVICE call with function code ?PTRDEV_LAST_EVENT. AOS/VS returns the following information in the subpacket.

- The current location of the pointer (X and Y coordinates, in pixels).
- The number of the button (if the last pointer event involved a button).
- The type of event that just took place.

Controlling the Appearance of the Pointer

Although a process cannot control the location of the pointer, it can control how the pointer appears when it's within the process's window. You can choose from four different pointer shapes, and you can enable or disable the pointer in your window. (Your choices do not affect the pointer when it is in another application's window, or when it is not in any window.)

By default, AOS/VS determines the shape of the pointer. To change the pointer shape, issue ?PTRDEVICE with function code ?PTRDEV_SET_POINTER. You can choose from the following pointer shapes.

- Arrow
- Pointing finger
- Small cross-hair
- Full-screen cross-hair

Getting Information About the Pointer Device

You can request information about the current state of the pointer device in a particular window by issuing ?PTRDEVICE with function code ?PTRDEV_GET_PTR_STATUS. AOS/VS returns the following information.

- The number of pixels the pointer will traverse along the X axis before AOS/VS reports a movement event.
- The number of pixels the pointer will traverse along the Y axis before AOS/VS reports a movement event.
- The events you have requested for this window.
- The button(s) to which button-related events will apply.
- The current pointer type for this window.
- Whether or not you have enabled the pointer in this window.

Suspending Output to a Window

Your program can suspend output to a window, so that AOS/VS pends output requests to that window. However, a suspended window can still appear on the screen, and you can still perform most windowing operations on a suspended window. Suspending a window does not affect its window priority.

To suspend output to a particular window, issue ?WINDOW with function code ?WIN_SUSPEND_WINDOW. To resume output to a suspended window, issue ?WINDOW with function code ?WIN_UNSUSPEND_WINDOW.

You can also suspend output to an entire group of windows; to do so, issue ?WINDOW with function code ?WIN_SUSPEND_GROUP. To resume output to a suspended group of windows, issue ?WINDOW with function code ?WIN_UNSUSPEND_GROUP.

AOS/VS uses the following set of rules to determine the suspension of windows and groups.

- When you suspend an individual window, it remains suspended, regardless of the state of its group.
- When you resume output to an individual window (?WIN_UNSUSPEND_WINDOW), that window resumes processing unless its group is suspended.
- When you suspend a window group, every window in that group becomes suspended.
- When you resume output to a group (?WIN_UNSUSPEND_GROUP), its windows resume processing, except for windows that are still suspended individually.

Sending Output to a Character Window

Sending output to a character window is much like sending output to a character-oriented terminal. This is because a character window's virtual terminal is a software emulation of a D460 terminal. You can write to its virtual screen using ?WRITE, just as you would with a real D460 terminal. You can also use D460 commands, D460 character sets, user-definable character sets, and D460 graphics commands.

Although a character window's virtual terminal is similar to a D460 terminal, there are a few D460 commands that are not supported in character windows. Because of this, the virtual terminal has its own model ID.

Model ID

The model ID of a character window's virtual terminal is different from the normal D460 model ID. This lets your program tell whether it is running on a real D460 or in a character window. When you issue the D460 command "Read Model ID" or the ANSI command "Read Terminal Configuration," AOS/VS returns one of the following model IDs.

<120> The virtual terminal is a D460 emulation that does not support the blink attribute. (AOS/VS uses this model ID when the D460 emulation is running on a pixel-mapped terminal that does not support the blink attribute; for example, a monochrome terminal with a pixel depth of 1.)

<121> The virtual terminal is a D460 emulation that supports the blink attribute.

Initial State

When you first create a character window, or after you reset it, the virtual terminal has the following characteristics.

7–bit/8–bit mode: The initial setting corresponds to that of a physical D460 with DIP switch settings in DG 8–bit operating mode. AOS/VS immediately issues a Select 7/8–Bit Mode command to put the virtual terminal into 7–bit mode. You can then use the Select 7/8–Bit Mode command to switch back and forth between 7– and 8–bit mode.

Margins: Left margin is set to column 0.

Right margin is set to column 79, or to the rightmost column if the virtual terminal is less than 80 columns wide.

Display screen:	The virtual screen is cleared.
Keyboard cursor:	The keyboard cursor is a reverse video block at the upper left corner of the virtual screen.
Character sets:	The primary character set is U.S. ASCII. User-defined character sets are cleared. Line drawing pattern is set to solid.
D460 "window":	The horizontal D460 "window" settings are cleared (the entire virtual screen is one D460 "window").
Screen roll:	Enabled.
Character blinking:	Enabled (except on monochrome terminals).
Character attributes:	Off.
Character protection:	Disabled.

DG Mode Restrictions

When the virtual terminal is in DG mode, commands such as Write Screen Address work only on row and column positions from 0 to 255. (This is because such commands use only one byte's worth of information.) If you create a window with a virtual terminal that has more than 255 rows or columns, you must use other commands (such as Cursor Right) to work with row and column positions beyond 255. When the terminal is operating in ANSI mode, commands work on any row and column position.

Unsupported Commands

Character windows support all D460 functions as described in the DASHER® D410 and D460 Display Terminals User's Manual, with the following exceptions.

Function	Description
Printer commands	Character windows do not support a local printer. D460 printer commands return a CTRL–F, just as they do on a real D460 terminal when there is no local printer. D460 printer commands are Form Bit Dump Print Form Print Pass Through On Print Pass Through Off Print Window Window Bit Dump Media Copy (ANSI)
Compressed spacing	Character windows support only normal character spacing. The following spacing commands are ignored: Select Compressed Spacing Select Normal Spacing Set Parameters (ANSI): the character spacing parameter has no effect.

Variable scrolling	Character windows support only the "jump" scroll rate. The following scroll rate commands are ignored: Set Scroll Rate Set Parameters (ANSI): the variable scroll rate parameter has no effect.
Horizontal scrolling	Character windows support only vertical scrolling. The following horizontal scrolling commands are ignored: Scroll Right Scroll Left Show Columns Read Horizontal Scroll Offset Horizontal Scroll Enable Horizontal Scroll Disable Set/Reset Mode (ANSI): horizontal scroll enable/disable has no effect.
Local mode	Character windows do not have an "off line" or "local" mode; they are always "on line."
XON/XOFF	Character windows do not support the XON/XOFF protocol when operating in either DG or ANSI mode. This affects the XON and XOFF commands in ANSI mode.

Sending Output to a Graphics Window

Sending output to a graphics window is different from sending output to a character window. This is because a graphics window's virtual terminal is a virtual graphics device (unlike a character window, whose virtual terminal is a software emulation of a character device).

Just as you might design a program to run on a particular graphics terminal, you may want to tailor your program so that it can run on a virtual graphics device. Your program could then perform full graphics functions while running in a complete windowing environment.

In order to run successfully on a virtual graphics device, your program must use the GIS II (Graphics Instruction Set II) instructions and the ?GRAPHICS system call. You will use the ?GRAPHICS call to open and close the graphics window. You'll use GIS II instructions to perform specific graphics operations, such as drawing lines.

Data General offers graphics packages that are built on GIS II. You can use these packages (DG/GI and GKS) instead of issuing GIS II instructions directly.

How Do Graphics Windows Differ From Character Windows?

Character windows and graphics windows both consist of a virtual terminal within a window border. The difference between the two is in the type of virtual terminal.

For character windows, the virtual terminal is a software emulation of a D460 terminal, which supports character I/O and D460-level graphics. To an application program, a character window looks much like a D460 terminal. This means that most programs that run on a D460 terminal can run in a character window.

For graphics windows, the virtual terminal is a *virtual graphics device* — a virtual pixel map graphics terminal that supports graphics operations. To an application program, a

graphics window looks exactly like a physical pixel map terminal. However, unlike a physical terminal, a virtual graphics device is inside a window which can be manipulated by an application or a user.

Every window, character or graphics, has the following attributes:

- A virtual terminal (part or all of which appears on the physical screen, surrounded by an optional border, with an optional title).
- A window name (you assign a window name when you create the window).
- A window pathname (@PMAPn:windowname) you can use to refer to the window.
- A window ID number (AOS/VS returns the ID number when you create the window).
- An input buffer that can receive input from the keyboard and the pointing device, and that you can read from by issuing a ?READ call.
- A channel number to the window's I/O channel (AOS/VS returns the channel number when you do a ?OPEN on the window). Input for the I/O channel comes from the window's input buffer, which can receive keystrokes from the keyboard and pointer events from the pointer device.

Table 6–2 compares character and graphics windows.

Table 6–2. Graphics Windows Versus Character Windows

Characteristic	Graphics Window	Character Window
Virtual terminal	Resembles a pixel map terminal.	Resembles a D460 terminal.
Virtual screen	A pixel map array in memory on which you can execute graphics instructions.	A character-oriented virtual screen.
Open for input	You issue the ?OPEN call. AOS/VS returns a channel number you can use for input only.	You issue the ?OPEN call. AOS/VS returns a channel number you can use for both input and output.
Receive input	You issue the ?READ call.	You issue the ?READ call.
Echo keyboard input on ?READ	You must perform keyboard echoing by writing to the window's pixel map, using graphics instructions. AOS/VS cannot echo keyboard input on a ?READ call.	AOS/VS can perform keyboard echoing on a ?READ call.
Close for input	You issue the ?CLOSE call and specify the channel number. This closes the window for input only.	You issue the ?CLOSE call and specify the channel number. This closes the window for both input and output.
Open for output	You issue ?GRAPHICS with function code ?GRAPH_OPEN_WINDOW_PIXELMAP. AOS/VS returns a pixel map ID you can use for graphics output only.	You issue the ?OPEN call. AOS/VS returns a channel number you can use for both input and output.

(continues)

Table 6-2. Graphics Windows Versus Character Windows

Characteristic	Graphics Window	Character Window
Write output	You write to the window's pixel map using graphics instructions.	You issue the ?WRITE call, or D460 graphics commands.
Close for output	You issue ?GRAPHICS with function code ?GRAPH_CLOSE_PIXELMAP, and specify the pixel map ID. This closes the window for graphics output only.	You issue the ?CLOSE call and specify the channel number. This closes the window for both input and output.

(concluded)

Pixel Maps

A pixel map is a rectangular array of memory. Each entry in the array is a set of bits that represents one pixel. The number of bits in each pixel is called the pixel depth; it determines how many colors are available for that pixel map.

When you create a graphics window, AOS/VS creates a pixel map in memory that's directly associated with the window. This pixel map is the graphics window's virtual screen. If the graphics window is visible, some or all of the pixel map's contents appear on the physical screen.

You can also create pixel maps that are not associated with a graphics window. Such pixel maps will not be visible on the physical screen, but you can still execute GIS II instructions and ?GRAPHICS functions on them. You might use such a pixel map to contain a font, since you can quickly copy a section of that pixel map (such as a single character) to the graphics window's pixel map.

You can create as many pixel maps in memory as you want. However, if you create too many pixel maps, the system response time may suffer.

Two or more processes can share a graphics window's pixel map. To do so, each process issues a ?GRAPHICS call with function code ?GRAPH_OPEN_WINDOW_PIXELMAP, and supplies the graphics window's ID or pathname. AOS/VS returns a separate pixel map ID for each ?GRAPH_OPEN_WINDOW_PIXELMAP call. (There can be only one channel at a time open to a memory pixel map.)

Working with Pixel Maps

The ?GRAPHICS call provides several functions you can use to work with pixel maps. Each function has a separate function code you specify in the main ?GRAPHICS packet. The functions are

Function Code	What It Lets You Do
?GRAPH_OPEN_WINDOW_PIXELMAP	Open a window's pixel map for graphics I/O. You can open such pixel maps multiple times. (This function returns a pixel map ID.)
?GRAPH_CREATE_MEMORY_PIXELMAP	Create a pixel map in memory and open it for graphics I/O. (This function returns a pixel map ID.)

?GRAPH_CLOSE_PIXELMAP	Close a pixel map. (The pixel map can belong to a graphics window, or it can exist in memory only.) When you close a memory-only pixel map, AOS/VS automatically deletes it. When you close a pixel map that belongs to a graphics window, AOS/VS closes only the channel to the pixel map ID you specify. If you opened that pixel map multiple times, you must close each pixel map ID separately. (To delete a window's pixel map, you must delete the window using the ?WINDOW call, function ?WIN_DELETE_WINDOW.)
?GRAPH_PIXELMAP_STATUS	Get information about a pixel map.

As you can see, ?GRAPHICS provides functions you use to create, open, and close pixel maps. (?GRAPHICS provides other functions for copying information between pixel maps and disk files, working with palettes, and setting clip rectangles; we describe these functions later in this chapter.)

Sending Output to a Pixel Map

To send output to a pixel map, you write to it using the GIS II instruction set. (You cannot send character output to a pixel map using ?WRITE, as you can with a character window.) You can use GIS II to write to any pixel map, whether it belongs to a graphics window or exists only in memory.

To send output to a graphics window's pixel map, you must first open it for output by issuing a ?GRAPHICS call with function code ?GRAPH_OPEN_WINDOW_PIXELMAP. This opens the window's pixel map and returns a pixel map ID, which you can treat as a graphics output channel. Once you have a pixel map ID, you can use GIS II instructions to send output to that pixel map.

You don't need to explicitly open a memory-only pixel map; when you create it (using ?GRAPHICS function ?GRAPH_CREATE_MEMORY_PIXELMAP), AOS/VS automatically opens it and returns a pixel map ID. You can then use that ID when you execute GIS II instructions on that pixel map.

Working with Pixel Map Files

Using the ?GRAPHICS call in conjunction with assembly language instructions, you can store the contents of a memory pixel map in a disk file, or read information from a file into a memory pixel map.

The ?GRAPHICS call provides two functions that let you copy data between nonstandard pixel map files and pixel maps in memory.

Function Code	What It Lets You Do
?GRAPH_MAP_PIXELMAP	Map an existing pixel map into your program's address space. You can then use assembly language instructions to move data from a pixel map file to the address space.

?GRAPH_UNMAP_PIXELMAP Map information from your program's address space to an existing memory pixel map. You can then use graphics instructions to edit the contents of the pixel map.

Copying Data from a File to a Pixel Map — Follow these steps to copy information from a pixel map file to a pixel map.

1. Create a destination pixel map and open it for graphics output. The destination pixel map must be a memory pixel map, and should be large enough to contain the information you are copying.

2. Issue the ?GRAPHICS call with function code ?GRAPHICS_MAP_PIXELMAP, and specify the pixel map ID of the destination pixel map.

   AOS/VS then maps the contents of the pixel map into your program's address space, and returns the following information:

   - A word pointer to the address space.
   - Pertinent information about the pixel map in the address space (such as its size and pixel depth).

3. You can now treat what's in your program's address space as an array of bits, rather than a pixel map. You can use assembly language instructions (such as LOAD and STORE) to move data from the disk to the address space.

   CAUTION: After you map the pixel map into your program's address space, do not use graphics instructions to write to it; the results are undefined. Before using graphics instructions on that pixel map you must issue a ?GRAPHICS call with function code ?GRAPHICS_UNMAP_PIXELMAP.

4. When you have copied all the data you want, issue ?GRAPHICS with function code ?GRAPHICS_UNMAP_PIXELMAP. AOS/VS then maps the contents of the address space (which now includes information from your pixel map file) back into the memory pixel map. The contents of your pixel map file are now in a memory pixel map, where you can perform GIS II instructions on them.

   CAUTION: Do not use assembly language instructions to edit the data in a pixel map in memory; the results are undefined.

Figure 6–6 illustrates this process.

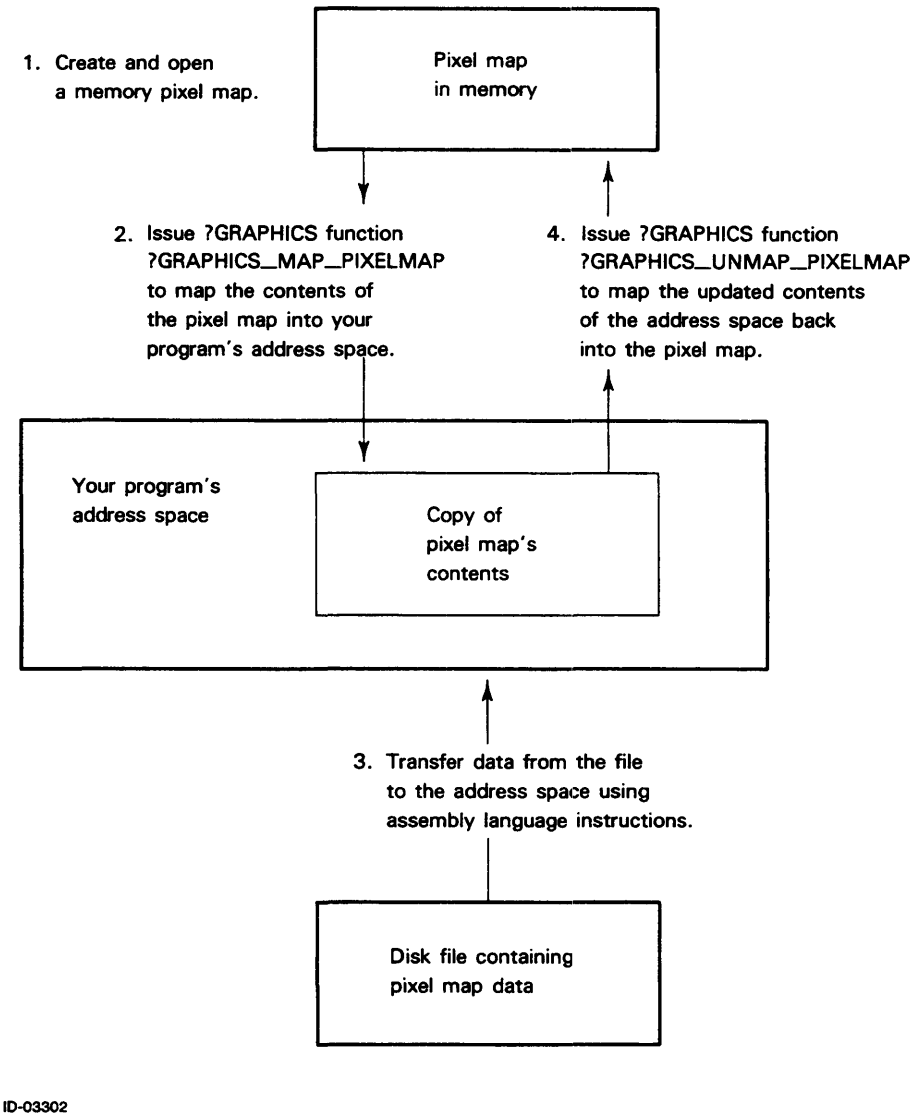


Figure 6–6. Copying Data from a File to a Pixel Map.

Copying Data from a Pixel Map to a File — You can also use the ?GRAPHICS call, in conjunction with assembly language instructions, to copy information from a pixel map to a pixel map file. To do so, follow these steps.

1. The source pixel map must exist in memory and be open for graphics output.
2. Issue the ?GRAPHICS call with function code ?GRAPHICS_MAP_PIXELMAP, and specify the pixel map ID of the source pixel map.

AOS/VS then maps the contents of the pixel map into your program's address space, and returns the following information.

- A word pointer to the address space.
- Pertinent information about the pixel map in the address space (such as its size and pixel depth).

3. You can now use assembly language instructions to move data from the address space to your disk file, in whatever format you want.
4. When you have copied all the data you want, issue ?GRAPHICS with function code ?GRAPHICS_UNMAP_PIXELMAP. AOS/VS then maps the contents of the address space back into the memory pixel map, so that you can again perform graphics instructions on that pixel map.

Figure 6–7 illustrates this process.

Palettes

Every pixel map has an associated *palette*. The palette lets you determine the colors you will use for that pixel map.

Each pixel in the pixel map array is represented by a series of bits. The number of bits for each pixel is the *pixel depth*. Thus, pixels in a pixel map with a depth of 4 would each consist of 4 bits. The bits in each pixel can be turned on and off to make different numbers, each of which can represent a different color. For example, a 4-bit pixel might be set to 0000 to display green, 0001 to display yellow or 0010 to display bluish purple, and so on.

You can define what colors you want each pixel value to represent. The palette is where you store your definitions.

The Blink Clock

For each pixel value, a palette contains two color definitions. If the two colors are identical, that pixel value displays a steady color. If the two colors for that pixel value differ, the pixel value will blink between the two colors.

Every graphics board has a *blink clock*. The clock flips steadily back and forth between two states, Phase 0 and Phase 1, at a predetermined time interval. During Phase 0 of the blink clock, the graphics board uses the first color for that pixel value. It uses the second color during Phase 1.

We explain how to use this feature in the section "Blending Steady/Blinking Colors" later in this chapter.

Palette Format

A palette consists of a pair of arrays in memory. The first array is used when the blink clock is in Phase 0; the second array is used during Phase 1.

The size of the arrays is determined by the pixel depth of the associated pixel map. Each array contains an entry for each possible pixel value. (For example, if the pixel depth is 4, each array has 16 entries. In general, if the pixel depth is n, each array in the palette has 2^n entries.) Each entry in each array has a particular color associated with it. You create the colors yourself, by blending different levels of red, green, and blue.

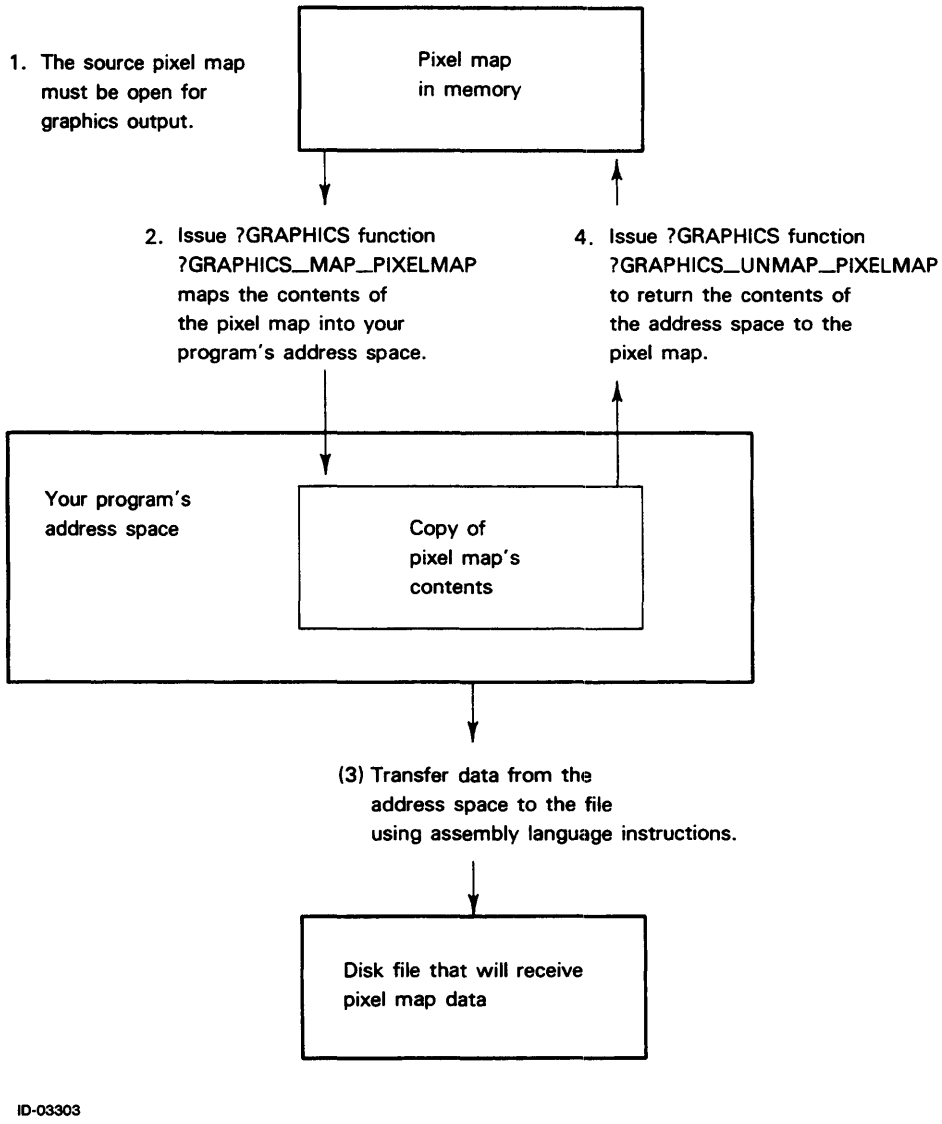


Figure 6–7. Copying Data from a Pixel Map to a File

Table 6–3 shows a sample palette whose pixel map has a pixel depth of 4. There are two arrays in the palette, one for Phase 0 of the blink clock, the other for Phase 1. Each array has 16 entries.

Table 6-3 Sample Palette

Pixel Value		Array 0 — Color Levels				Array 1 — Color Levels			
		Red	Green	Blue	Grey	Red	Green	Blue	Grey
0	(0000)	full	full	full	full	full	full	full	full
1	(0001)	off	off	off	off	off	off	off	off
2	(0010)	off	full	off	off	off	full	off	off
3	(0011)	full	off	off	off	full	off	off	off
4	(0100)	off	½	½	off	off	½	½	off
5	(0101)	off	full	off	off	full	off	off	off
	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.
14	(1110)	off	½	½	off	off	full	full	off
15	(1111)	½	¼	full	off	½	¼	full	off

For each array, an entry consists of four 32–bit numbers, each of which represents the level for a color. There are three colors: red, green, and blue. The relative amount of each color you use determines the final color for that entry. (There is also a grey level, which lets you set the brightness for a monochrome terminal.)

Blending Colors

To create a color on your screen, you combine the three primary colors (red, green, and blue).

NOTE: Mixing colored light is different from mixing colored paints; the primary colors for paints are red, yellow and blue. For example, to get yellow paint, you must start with yellow; to get yellow light, you mix equal parts of red light and green light.

To blend a color, you set each of the three color levels to determine how much of each color will appear in the final color. Each color can have a value from 0 to $2^{32}-1$ (from no bits set in the 32–bit number, to all bits set). The lowest color level, 0, turns that color off completely; the highest level, $2^{32}-1$, turns that color to its brightest possible level. For color levels in between, you must use binary fractions (set some, but not all, of the bits).

For example, to blend a pure, bright blue, you would set the first, second and fourth numbers in the entry to off (0, to turn off the red, green and grey), and the third (blue) to "full" ($2^{32}-1$, 37777777777_8, or simply -1). The entry for that pixel value would then consist of the following four 32–bit numbers.

0	0	37777777777_8	0
(red)	(green)	(blue)	(grey)

For a bright reddish purple, you would set green and grey to completely off (0), and set blue and red to completely on ($2^{32}-1$). The entry for that pixel value would then consist of the following four 32–bit numbers.

37777777777_8	0	37777777777_8	0
(red)	(green)	(blue)	(grey)

Using Binary Fractions to Blend Colors — To create an in-between color level, in which the color is neither fully on nor fully off, you must use a binary fraction. To do

so, you set some, but not all, of the bits in the 32–bit number. You can think of the 32–bit number that represents the color level as a representation of a fraction between 0 and 1. When no bits are set, the number equals zero; with all bits set, the number is almost equal to 1. The closer you get to 1, the brighter the color level. Figure 6–8 shows the correspondence between the fraction from 0 to 1 and the 32–bit number from 0 to $2^{32}-1$.

Using this correspondence, you can treat $2^{32}-1$ as a representation of 1. To get the binary representation of ½, you could then follow the calculations below to get the result 20000000000_8 (a 32–bit number with the high-order bit set).

$$\frac{1}{2} = \frac{1}{2} \times 1 = \frac{1}{2} \times 2^{32} = 2^{31} = 20000000000_8 = \text{a 32–bit number with the high-order bit set.}$$

The binary representation of ⅞ is a 32–bit number with the first three high-order bits set.

$$\frac{7}{8} = \frac{7}{8} \times 1 = \frac{7}{8} \times 2^{32} = 7 \times 2^{29} = 34000000000_8 = \text{a 32–bit number with the first three high-order bits set.}$$

Thus, to produce a dim blue, you could set the blue level to "½," and all other colors to "off." To produce a binary fraction equal to ½, set the high-order bit to produce the fraction ½ (20000000000_8). The entry for that pixel value would then consist of the following four 32–bit numbers.

0	0	20000000000_8	0
(red)	(green)	(blue)	(grey)

For a dim greenish blue, you might set blue to "½" (set the high-order bit) and green to "¼" (set the second highest bit to produce the fraction ¼ (10000000000_8). The entry for that pixel value would then consist of the following four 32–bit numbers.

0	10000000000_8	20000000000_8	0
(red)	(green)	(blue)	(grey)

For shades of grey, set all colors to the same level — completely on for white, completely off for black. For example, to blend a light grey, you might set all color levels to "⅞"

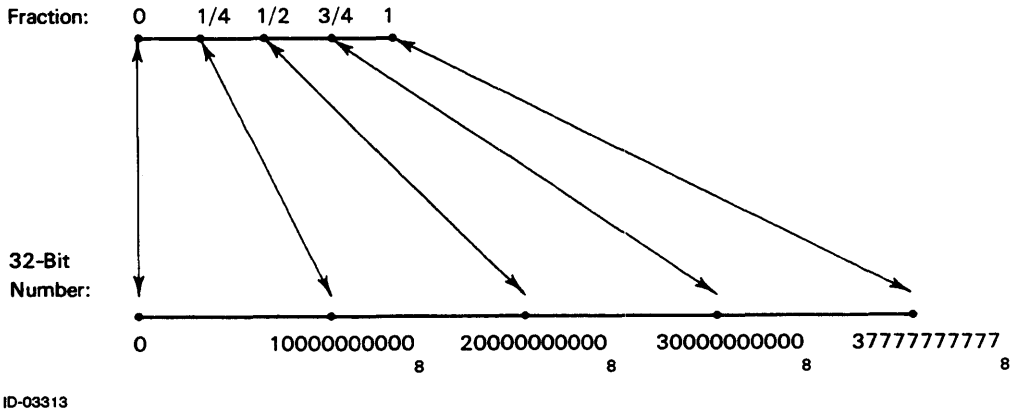


Figure 6–8. Correspondence Between Fraction and 32–Bit Number for Color Levels

(set the first three bits to produce the fraction ⅞ (34000000000_8). The entry for that pixel value would then consist of the following four 32–bit numbers.

34000000000_8	34000000000_8	34000000000_8	0
(red)	(green)	(blue)	(grey)

The Grey Level — The grey level determines the brightness of the pixels on monochrome screens only. When you display a color on a color screen, the grey level is ignored. This means that you can set up a palette so that it works on both monochrome and color screens. For each pixel value, you set up both a color (blending red, blue and green) and a grey level. When the palette is used on a color screen, the color appears. On a monochrome screen, the color values are ignored and the grey level will be used instead.

On a monochrome screen, a grey level of "full" (2^{32} –1, 37777777777_8 or simply –1) appears as white; a grey level of "off" (0) appears as black. Levels in between range from near-black to near-white.

Blending Steady/Blinking Colors

As we explained earlier, each palette contains two arrays. Array 0 determines the colors during Phase 0 of the blink clock; Array 1 determines the colors during Phase 1.

To get a blinking color, the entry in Array 0 must differ from its corresponding entry in Array 1. Let's look at an example.

Suppose you want the pixel value 0010 to denote a color value that blinks from red to green. For the third entry in the palette (the one for 0010), you could set the following color levels. Note that the entry consists of four 32–bit numbers for Array 0 and four for Array 1.

Array 0:	37777777777_8	0	0	0
	(red)	(green)	(blue)	(grey)
Array 1:	0	37777777777_8	0	0
	(red)	(green)	(blue)	(grey)

During Phase 0 of the blink clock, all pixels set to the value 0010 would appear bright red. During Phase 1, all pixels set to 0010 would instead appear bright green.

If you don't want a particular pixel value to blink, make sure its corresponding entries in Array 0 and Array 1 are identical.

For example, the following palette entry would produce a dim, steady aqua color, because the color settings do not change from phase to phase of the blink clock.

Array 0:	0	10000000000_8	10000000000_8	0
	(red)	(green)	(blue)	(grey)
Array 1:	0	10000000000_8	10000000000_8	0
	(red)	(green)	(blue)	(grey)

Working with Palettes

The ?GRAPHICS call provides two functions you can use to work with palettes. Each function has a separate function code you specify in the main ?GRAPHICS packet.

Function Code	What It Lets You Do
?GRAPH_WRITE_PALETTE	Write a set of contiguous entries to the palette associated with a pixel map.

?GRAPH_READ_PALETTE Read a set of contiguous entries from the palette associated with a pixel map.

Both functions operate on any palette, whether it belongs to a memory-only pixel map or to the pixel map associated with a graphics window.

Writing Entries to a Palette

To define the colors you want for particular pixel values, issue the ?GRAPHICS call with function code ?GRAPH_WRITE_PALETTE. This call writes a series of contiguous entries to a palette. To write to a palette, follow these steps.

1. Set up an array in memory for each array you want to write. You can write a set of entries for the Phase 0 array, the Phase 1 array, or both. The size of the arrays depends on the number of entries you want to write. Multiply the number of entries by the size of each palette entry (?GRAPH_PALETTE_LEN, which is always 4 double words). For example, if you want to write 10 entries, each array would be 10 * 4 double words, or 40 double words per array. (If you are writing both arrays, you must write the same number of entries for each array; the arrays must be the same size.)

2. Issue the ?GRAPHICS call with function code ?GRAPH_WRITE_PALETTE. In the subpacket, you specify

   - Whether you want to write Array 0, Array 1, or both.
   - The number of palette entries you want to write.
   - The pixel map ID of the pixel map associated with the palette you want.
   - The palette index at which you want to start writing.
   - The addresses of the arrays you have stored in memory.

AOS/VS then copies the palette entries from the arrays in memory to the palette you specified.

Reading Entries from a Palette

To get the palette values for an existing palette, issue the ?GRAPHICS call with function code ?GRAPH_READ_PALETTE. This call copies a series of contiguous entries from the palette. You can copy the Phase 0 array, the Phase 1 array, or both. To use this call, follow these steps.

1. Allocate space in memory for each array you are copying. Each array needs as much space as the number of entries you will copy multiplied by the size of each entry (4 double words). So, to copy 20 entries, you would allocate 80 double words for each array.

2. Issue the ?GRAPHICS call with function code ?GRAPH_READ_PALETTE. You specify

   - Whether you are copying Array 0, Array 1, or both.
   - The pixel map ID of the pixel map associated with the palette you want.
   - The palette index at which you want to start reading.

- The number of palette entries you want to read.
- The addresses of the arrays you have stored in memory.

AOS/VS then copies the entries from the palette you specified to the arrays in memory.

Copying Information from a Disk File to a Palette

To copy palette information from a disk file to a palette

1. Read the disk file into memory using assembly language instructions. The file should be an array or arrays in palette format (as described above).
2. Issue ?GRAPHICS with function code ?GRAPH_WRITE_PALETTE, and supply the addresses of the arrays in memory (the ones you just read from disk). Supply other information as described in the section "Writing Entries to a Palette" above.

AOS/VS then copies the information from the array(s) in memory to the palette you specify.

Copying Information from a Palette to a Disk File

To copy information from a palette to a disk file

1. Issue ?GRAPHICS, function ?GRAPH_READ_PALETTE; supply information as described in the section Reading Entries from a Palette above.
2. Write the information from memory into a disk file using assembly language instructions.

Clip Rectangles

AOS/VS lets you specify a rectangular section of a pixel map as a *clip rectangle*. When a clip rectangle is in effect, you can still specify any location on the pixel map as an argument to a graphics instruction. However, any changes affect only the portion of the pixel map within the rectangle.

For example, suppose you want to protect a drawing while a user types a label for it. You want to let the user edit the portion of the pixel map that will contain the label, but you don't want the user to be able to alter the rest of the pixel map. A clip rectangle can restrict changes to the rectangular area that will contain the label.

To manipulate a clip rectangle on a pixel map, use the ?GRAPHICS call, function code ?GRAPH_SET_CLIP_RECTANGLE. There can be only one clip rectangle per pixel map.

When you issue the call, you specify

- Whether you are enabling or disabling the clip rectangle. You can enable and disable the same rectangle as many times as you want. This lets you leave the rectangle in place while you are not using it.

  When you enable the clip rectangle, changes are possible only to the area within the rectangle.

  When you disable the clip rectangle, it is left in place but no longer protects the pixel map from changes.
- The position of the clip rectangle on the pixel map.
- The size of the clip rectangle.

When You're Done with a Window

When you have finished with a window and want to get rid of it, you first close the window and then delete it.

Closing a Window

Before you delete a window, you must close all channels you have opened to that window.

To close a regular I/O channel (a channel created with the ?OPEN system call), issue the ?CLOSE call and specify the channel number. AOS/VS then releases that channel number. If you have opened several I/O channels to that window, you must issue a separate ?CLOSE call for each channel.

To close a graphics window's pixel map ID (which you can think of as a "graphics output channel"), issue the ?GRAPHICS call with function code ?GRAPH_CLOSE_PIXELMAP, and specify the pixel map ID. If you have opened several pixel map IDs to that window, you must issue a separate ?GRAPH_CLOSE PIXELMAP call for each channel.

After you close a window, you can still issue ?OPEN (or ?GRAPH_OPEN_WINDOW_PIXELMAP) to open new channels to the window.

Deleting a Window

After you close all channels and pixel map IDs for a window, you can delete the window. To do so, issue the ?WINDOW call with function code ?WIN_DELETE_WINDOW. This marks the window for deletion, and automatically removes its channel assignment. If the window is not assigned to any other processes, it goes away. If it is still assigned to other processes, the window is marked for deletion; the window name doesn't disappear (unlike files marked for deletion). When the window's last process terminates, or relinquishes its channel number, the window finally goes away.

End of Chapter

Chapter 7
Initiating and Managing Tasks

You use the following system calls to initiate and manage tasks:

?DFRSCH	Disable scheduling and indicate prior state of scheduling.
?DQTSK	Remove a task or tasks from a queue.
?DRSCH	Disable task scheduling.
?ERSCH	Enable task scheduling.
?IDGOTO	Redirect a task.
?IDKIL	Terminate a task specified by its TID.
?IDPRI	Change the priority of a task specified by its TID.
?IDRDY	Ready a task specified by its TID.
?IDSTAT	Return a task statistic flag. (16–bit processes only)
?IDSUS	Suspend a task specified by its TID.
?IFPU	Initialize the floating-point status registers.
?IQTSK	Create a queued task manager (for ?TASK queuing).
?KILAD	Define a termination-processing routine for a task.
?KILL	Terminate the calling task.
?MYTID	Return the TID of the calling task.
?PRI	Change the priority of the calling task.
?PRKIL	Terminate all tasks of a given priority.
?PRRDY	Ready all tasks of a given priority.
?PRSUS	Suspend all tasks of a given priority.
?REC	Receive an intertask message.
?RECNW	Receive an intertask message without waiting.
?SUS	Suspend the calling task.
?TASK	Initiate one or more tasks.
?TIDSTAT	Return the status of a task specified by its TID. (32–bit processes only)
?TLOCK	Protect a task from being redirected.
?TRCON	Read a task message from a process terminal.
?TUNLOCK	Revoke redirection protection for the current task in the current ring.
?UIDSTAT	Return the status of a task and an unambiguous identifier.
?WDELAY	Suspend a task for a specified time.
?XMT	Transmit an intertask message.
?XMTW	Transmit an intertask message and wait for its receipt.

This chapter describes what a task is, how to create a task, and how to manage a multitasking environment.

What Is a Task?

A *task* is a path through a process that can only execute code within the bounds of the address space that AOS/VS allocates to its process.

Each process consists of one or more tasks, which execute asynchronously (at different times). You can design your program so that several tasks execute a single re-entrant sequence of instructions, or so that two or more of the tasks execute separate, distinct instruction paths.

A task is the basic element of a process. When you first create a process, it is a single-task program. However, you can initiate other tasks from that process. These tasks exist either for the life of the process, or until you terminate them.

The Advantages of Multitasking

If you have used high-level languages such as BASIC or FORTRAN, you are familiar with single-task programs. Single-task programs display one path that connects all branches of logic, no matter how complex. Multitasking, which allows you to execute up to 32 tasks within a single process, gives you greater flexibility when writing complex programs. Specifically, multitasking gives you the following advantages:

- Parallelism

  Multitasking is a straightforward way to handle complex parallel events within one program; for example, it is 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 multitasking program so that several tasks execute one re-entrant instruction sequence, or you can create a different instruction path for each task.

AOS/VS Task Protection

AOS/VS prevents tasks executing in an outer ring from interfering with tasks executing critical inner-ring code paths. AOS/VS protects tasks from interference through two mechanisms: *ring maximization* and *ring specification.*

Ring Maximization

Under the ring maximization mechanism, AOS/VS considers a task that is executing in a user ring to be less privileged than those tasks executing in lower user rings.

AOS/VS uses the ring-maximization mechanism when it validates user-supplied channels, word pointers, or byte pointers for all system calls. Consequently, a process cannot pass a

channel as input to a system call that it has issued from a higher ring than the system call that opened the channel. Also, system calls issued from one user ring cannot pass as input pointers to lower-ring memory locations.

The ring-maximization mechanism parallels the hierarchical protection scheme of the MV–series memory-management hardware.

Ring Specification

The ring-specification mechanism protects tasks executing in one user ring from interference by tasks executing in other user rings. The connection-management facility and the IPC facility use the ring-specification mechanism 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 receive 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 advisable. Tasks that share TID 0 cannot issue ?IDSTAT, ?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 find 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, the 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 Initiation

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:

- 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 issue a ?TASK system call that specifies a starting PC within Ring 7, 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 progam. 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 might 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 use the extended ?TASK packet, and you must issue the ?IQTSK system call before you issue the ?TASK system call. 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.

Allocating a Task's Stack Space and Defining the Stack

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 specify the default stack fault 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 you choose this option, you must allocate the stack before the newly initiated task issues any system calls. You must allocate a stack of 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. (For information on loading a program file into a specific ring, see the description of the ?RINGLD system call in Chapter 3.)

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:

- 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.

- 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)

  The Link utility 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 image, you should also specify 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 the 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.

AOS/VS initializes inner-ring stacks through the following steps:

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 two 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 the stack pointer and the frame pointer to the stack base. This ensures that tasks can reuse 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 reuse 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 process must be in Superprocess mode.

Tasks pass through several different states while a process is executing. A task passes from the 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 7–1 illustrates the task states and the system calls that affect them.

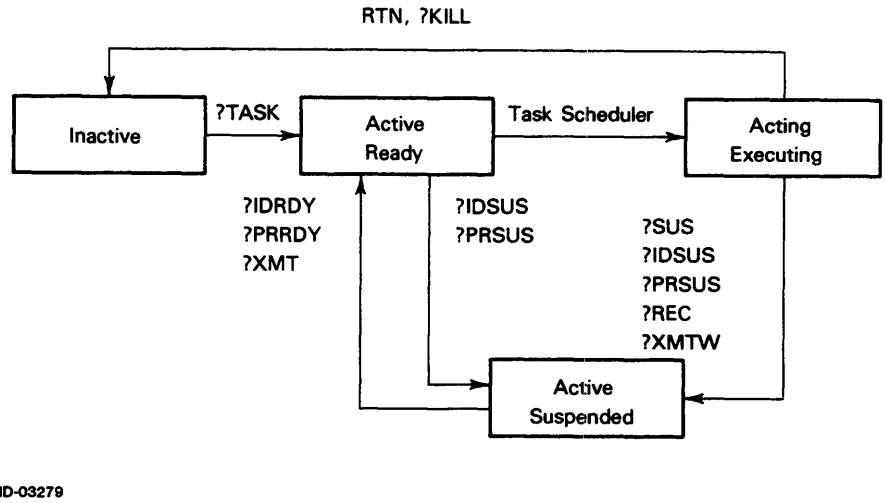


Figure 7–1. Task States

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 priority-level task that is ready to run and a round-robin interval has elapsed. (You specify the round-robin interval during the system-generation procedure.)

Disabling Task Scheduling

To disable task 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 the ?DFRSCH system call, which does. *These system calls disrupt the entire multitasking environment — you should only issue them if you are certain that you need to disable task scheduling.*

To re-enable scheduling after you have disabled it with a ?DRSCH or a ?DFRSCH system call, issue ?ERSCH. (For more information on the ?DRSCH and the ?DFRSCH system calls, see the Locking and Unlocking Critical Regions section later in this chapter.)

Task Suspension

Several different events, including some system calls, will suspend an active task. The task will then remain suspended until the event that caused the suspension completes, or until the suspended task is *readied* by either AOS/VS or by another task (we describe how to ready a task in the next section.)

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. System 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.

While tasks compete for all system resources, 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

- 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 the ?REC and ?XMTW system calls).
- Issues most system calls. A system call is usually a request to use some system resource.

If every task in a process is suspended, then that 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 the ?UBLPR system call to unblock that process.

Task Readying

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 (the task must not have been suspended for any other reason).

  The ?BLKPR and ?UBLPR system calls work together; 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 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.

  The task that is being readied must have been previously suspended by a ?SUS, ?IDSUS, or ?PRSUS system call. The task that you are readying must belong to the same process as the task from which you issue the ?IDRDY system call.

- A message for which the task was explicitly requested to wait (through the ?IREC system call) becomes available. In this case, the task only becomes ready when the message is from another task within the same process.
- A system resource becomes available after an implicit wait for that system resource during a system call.
- A task issues a system call that terminates (?IDKIL or ?PRKIL) or redirects (?IDGOTO) a suspended task.

NOTE: Before AOS/VS executes the system call, it automatically readies the target task.

Redirecting a Task

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 terminal interrupt sequence. (A CTRL–C CTRL–A sequence interrupts terminal output. For details about this function, see the description of ?IDGOTO in Chapter 13.)

Protecting Inner-Ring Tasks from Redirection

In general, you should not allow tasks executing in outer rings to redirect tasks executing in inner rings. However, task redirection is a common method of responding to external events. In fact, typing a CTRL–C CTRL–A terminal interrupt sequence frequently causes an ?IDGOTO system call to perform task redirection on the main task(s) of a process.

To prevent the unwanted redirection of a task, you can use the ?TLOCK and ?TUNLOCK system calls 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 either the same ring or a higher ring. The ?TUNLOCK system call unlocks a previously locked task.

A task can issue a ?TLOCK system call to protect itself from being redirected by any task that is in a higher ring or, optionally, in the same ring. The AOS/VS ring-maximization mechanism determines which task can redirect another task: It ensures that a task can redirect a lower-ring task to only an address within the ring that contains the task issuing the task-redirection system call. For example, a ring 7 task can redirect a ring 4 task, but only to a ring 7 address.

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 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 redirections 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.

Terminating a Task

You can terminate a task explicitly or implicitly. To explicitly terminate a task, issue one of the following system calls:

?IDKIL Terminates a task of a specified TID.

?PRKIL Terminates all tasks of a specified priority.

?KILL Terminates the calling task.

To terminate a task implicitly, begin the new task with a WSSVS or WSSVR (wide-save) instruction, 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-terminating routine (which you 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-terminating routine.

Because terminating a task does not guarantee an orderly release of its user-related resources, you may want to define a termination-processing routine for this purpose (for example, to close the task's currently open channels).

Defining Termination-Processing Routines

You can define either a unique termination-processing routine for each task, or a general termination-processing routine for all tasks within a process.

After you initiate a task, you can define a unique termination-processing routine for that task by issuing the ?KILAD system call. When you terminate that task with either the ?IDKIL or ?PRKIL system calls, AOS/VS will invoke the termination-processing routine that you specified.

If you define a general kill-processing routine, you must assign the routine the label ?UKIL and link it with your program.

You can use both ?KILAD and user-defined ?UKIL termination-processing routines within the same program.

If there is no user-defined ?UKIL routine to terminate a task, AOS/VS uses the dummy ?UKIL routine in URT32.LB. This routine returns control to AOS/VS, which then terminates the task. AOS/VS only invokes ?UKIL termination processing on behalf of tasks that initiated processing within Ring 7 (tasks for which initial PCs are Ring 7 addresses).

Detecting Task Creation and Termination

A local server often needs to maintain accurate task-specific databases. To keep those task-specific databases accurate, the local server must be able to keep track of when the process creates and terminates tasks.

AOS/VS keeps track of tasks through the use of unique storage position (USP) pointers. All active tasks have distinct 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 uniquely identifies every task within a process to aid in identifying task-specific databases with their tasks. The ?UIDSTAT system call returns the unique TID associated with a given task.

When a task terminates, AOS/VS serially invokes a ?UKIL postprocessor 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.

Terminal-to-Task Communication

AOS/VS allows you to pass a message from your terminal to individual tasks in a multitasking environment.

The ?TRCON system call creates a message-management system task on your behalf, which parses each message from you and transmits that message to the proper calling task.

Task-to-Task Communication

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 send a message with an ?XMT or an ?XMTW system call, you must initialize the appropriate mailbox to 0.

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 an ?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 an ?RECNW system call against an empty mailbox, the system call fails, and AOS/VS returns an error code to AC0.

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.

Locking and Unlocking a Critical Region

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, either by 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 an 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 using the ?DRSCH and the ?DFRSCH system calls. Although there 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.

MV–Family Floating-Point Registers

The MV–Family hardware has five registers that allow you to manipulate floating-point numbers:

- Four floating-point registers, FAC0, FAC1, FAC2, and FAC3.
- One floating-point status register, FPSR, which records information about the current state of the MV/Family floating-point processor.

Before you can use any of the MV/Family floating-point instructions from a task, you must issue ?IFPU to initialize the floating-point status register. To obtain accurate results for floating-point arithmetic, you must do this even for single-task programs.

Multitasking Sample Programs

The initial task of the following program, NEWTSK, creates a new task that has a priority of 1 and a TID of 2. The initial task opens the terminal, creates the new task, announces its termination, gets its priority, and terminates itself. Then, the new task takes control, writes a message, and returns to the CLI. (The last task cannot terminate itself with a ?IDKIL system call.)

NEWTSK uses the ?TASK, ?MYTID, and ?IDKIL system calls.

```
                .TITLE  NEWTSK
                .ENT    NEWTSK
                .TSK    2

;Open terminal (CON), create a new task, and kill self.

NEWTSK: ?OPEN   CON                 ;Open terminal (CON) for I/O.
        WBR     ERROR               ;?OPEN error return.

        ?TASK   TPKT                ;Create new task, TID 2, with
                                    ;priority of 1.
        WBR     ERROR               ;?TASK error return.
        ?WRITE  CON                 ;Display termination message
                                    ;on terminal.
        WBR     ERROR               ;?WRITE error return.
        ?MYTID                      ;Get TID in AC0 and priority
                                    ;in AC1.
        WBR     ERROR               ;?MYTID error return.
        WMOV    0,1                 ;Move TID into AC1
        ?IDKIL                      ;and die.
        WBR     ERROR               ;?IDKIL error return.

;New task is now the only task.

NTSK:   XLEFB   0,NMSG*2            ;Get byte pointer to message.
        XWSTA   0,CON+?IBAD         ;Put message in I/O packet.
        ?WRITE  CON                 ;Display message on terminal.
        WBR     ERROR               ;?WRITE error return.
        WSUB    2,2                 ;Set AC2 for normal return.
        WBR     BYE                 ;Go and return.
```

```
;Error handler.

ERROR:  NLDAI   ?RFEC!?RFCF!?RFER,2     ;Error flags: Error code is in
                                        ;AC0 (?RFEC), message is in
                                        ;CLI format (?RFCF), and caller
                                        ;should handle this as an error
                                        ;(?RFER).

BYE:    ?RETURN                         ;Return to CLI.
        WBR     ERROR                   ;?RETURN error return.

;?OPEN and I/O packet for terminal.

CON:    .BLK    ?IBLT                   ;Allocate enough space for
                                        ;packet.
        .LOC    CON+?ISTI               ;File specifications.
        .WORD   ?ICRF!?RTDS!?OFIO       ;Change format to data-sensitive
                                        ;records and open for
                                        ;input and output.
        .LOC    CON+?IMRS
        .WORD   -1                      ;Default physical block size
                                        ;to 2 K bytes.

        .LOC    CON+?IBAD
        .DWORD  ITEXT*2                 ;Set byte pointer to record
                                        ;I/O buffer.

        .LOC    CON+?IRCL
        .WORD   120.                    ;Record length is 120
                                        ;characters.

        .LOC    CON+?IFNP
        .DWORD  CONS*2                  ;Set byte pointer to pathname.

        .LOC    CON+?IDEL               ;Delimiter table address.
        .DWORD  -1                      ;Use default delimiters: null,
                                        ;NEW LINE, form feed, and
                                        ;carriage return (default is
                                        ;-1).

        .LOC    CON+?IBLT               ;End of packet.

;Filename and messages.  A .NOLOC 1 follows.

CONS:   .TXT    "@CONSOLE"              ;Use generic name.

ITEXT:  .TXT    "I'm the default task. I have opened the terminal and
                 I'm about to ?IDKIL myself.<12>"

NMSG:   .TXT    "I'm the new task. I am about to ?RETURN.<12>"

        .NOLOC  0
```

```
;?TASK packet for new task.

TPKT:   .BLK    ?DSLTH              ;Allocate enough space for the
                                    ;standard packet.

        .LOC    TPKT+?DLNK
        .WORD   1                   ;Set to 1 for standard packet.
        .LOC    TPKT+?DLNL          ;Reserved.
        .WORD   0                   ;Set to 0.

        .LOC    TPKT+?DLNKB          ;Reserved.
        .DWORD  0                    ;Set to 0.

        .LOC    TPKT+?DPRI
        .WORD   1                   ;Assign priority 1 to the new
                                    ;task (default is 0, which
                                    ;assigns the new task the same
                                    ;priority as the caller).

        .LOC    TPKT+?DID
        .WORD   2                   ;Assign TID 2 to the new
                                    ;task (default is 0, which
                                    ;does not assign a TID to
                                    ;the new task).
        .LOC    TPKT+?DPC
        .DWORD  NTSK                ;Task's starting address is
                                    ;NTSK.

        .LOC    TPKT+?DAC2
        .DWORD  0                   ;There is no message for the
                                    ;new task.

        .LOC    TPKT+?DSTB
        .DWORD  STACK               ;Stack base address is STACK.

        .LOC    TPKT+?DSFLT         ;Stack fault handler address.
        .WORD   -1                  ;Use default stack fault
                                    ;handler in URT32.LB (default
                                    ;is -1).
        .LOC    TPKT+?DSSZ
        .DWORD  60.                 ;Stack size is 60 words.

        .LOC    TPKT+?DFLGS         ;Task flag word.
        .WORD   0                   ;Set to 0.

        .LOC    TPKT+?DRES          ;Reserved.
        .WORD   0                   ;Set to 0.

        .LOC    TPKT+?DNUM
        .WORD   1                   ;Create one task.

        .LOC    TPKT+?DSLTH         ;End of packet.

STACK:  .BLK    60.                 ;60-word stack for new task.

        .END    NEWTSK              ;End of NEWTSK program.
```

The following program, BOOMER, is a fast, two-task copy program that uses ?IXMT and ?REC system calls to synchronize ?READ and ?WRITE system calls. BOOMER copies an existing input file to an output file.

BOOMER uses the ?TASK, ?XMTW, ?REC, ?KILL, ?IXMT, ?READ, and ?WRITE system calls.

```
                    .TITLE  BOOMER
                    .ENT    BOOMER
                    .TSK    2
                    .NREL 1
;Initial task uses ?GTMES to get output filename (second argument) and
;opens it.  Repeats ?GTMES to get input filename (first argument) and
;opens it.  Creates output task.

BOOMER: ?GTMES  GPKT                    ;Get input filename.
        WBR     ERROR                   ;?GTMES error return.
        LLEFB   0,FNAME*2               ;Get byte address of filename
                                        ;that ?GTMES returns.
        LWSTA   0,INPUT+?IFNP           ;Put in input I/O packet.

        ?OPEN   INPUT                   ;Open INPUT file.
        WBR     ERROR                   ;?OPEN error return.
        NLDAI   1,0                     ;Get 1 in AC0.
        LNSTA   0,GPKT+?GNUM            ;Specify argument 1.

        ?GTMES  GPKT                    ;Get output filename.
        WBR     ERROR                   ;?GTMES error return.
        LLEFB   0,FNAME*2               ;Get byte address of filename
                                        ;that ?GTMES returns.
        LWSTA     0,OUTPUT+?IFNP        ;Put in output I/O packet.

        ?OPEN   OUTPUT                  ;Open OUTPUT file.
        WBR     ERROR                   ;?OPEN error return.

        ?TASK   TPKT                    ;Create output task.
        WBR     ERROR                   ;?TASK error return.

;Loop reads into BUF1, transmits it to output task, reads into BUF2,
;and transmits it to output task.  Message for output task is buffer
;address.

READER: ?READ   INPUT                   ;Read buffer from INPUT file.
        WBR     ERROR                   ;?READ error return.
        LLEF    0,MAILBOX               ;Get message address.
        LWLDA   1,INPUT+?IBAD           ;Message is buffer address.

        ?XMTW                           ;Wake up output task.
        WBR     ERROR                   ;?XMTW error return.
```

```
;Swap buffer byte pointers for next read.

        LLEFB   0,BUF1*2                ;Get byte pointer to BUF1.
        LLEFB   2,BUF2*2                ;Get byte pointer to BUF2.
        WSNE    1,2                     ;Was BUF1 used for last read?
        WMOV    0,2                     ;No. Make BUF1 current buffer.
        LWSTA   2,INPUT+?IBAD           ;Yes. Put byte pointer to
                                        ;current buffer into input
                                        ;packet.
        WBR     READER                  ;Read into current buffer.

;On end-of-file condition, get number of characters to read from input
;packet and make this number the buffer length for the last ?XMT.

EOF?:   NLDAI   EREOF,1                 ;Was error code"end-of-file"
                                        ;(EREOF)?
        WSEQ    0,1                     ;Yes.
        WBR     ERROR                   ;No. Try to handle the error.
        LLEF    0,MAILBOX               ;Get address of message.
        LWLDA   1,INPUT+?IBAD           ;Message is byte pointer to
                                        ;buffer.
        WMOV    1,2                     ;Copy to AC2 for indexing.
        NLDAI -1,3                      ;Put -1 in AC3.
        WLSH    3,2                     ;Make byte pointer to buffer
                                        ;a word pointer.
        LNLDA   3,INPUT+?IRLR           ;Get number of characters read
                                        ;from input I/O packet.
        LWSTA   3,-2,2                  ;Make buffer length (AC2-2)
                                        ;the number of characters
                                        ;read.

        ?XMTW                           ;Send last buffer.
        WBR     ERROR                   ;?XMTW error return.

        ?KILL                           ;Input is done; output task
                                        ;will return to CLI.

                                        ;Error handler.

ERROR:  WLDAI   ?RFEC!?RFCF!?RFER,2     ;Error flags: Error code is in
                                        ;AC0 (?RFEC), message is in
                                        ;CLI format (?RFCF), and caller
                                        ;should handle this as an error
                                        ;(?RFER).

        ?RETURN                         ;Return to CLI.
        WBR     ERROR                   ;?RETURN error return.

                                        ;Output task does the writing:
```

```
WRITER: LLEF    0,MAILBOX               ;Get message address.
        ?REC                            ;Wait for message.
        WBR     ERROR                   ;?REC error return.
        LWSTA   1,OUTPUT+?IBAD          ;Got message, which was byte
                                        ;pointer to buffer. Put in
                                        ;I/O packet.
        WMOV    1,2                     ;Copy to AC2 for indexing.
        NLDAI   -1,3                    ;Put -1 in AC3.
        WLSH    3,2                     ;Make byte pointer into word
                                        ;pointer.
        XWLDA   0,-2,2                  ;Get buffer length left by
                                        ;input task (original length,
                                        ;unless task hit end of file).

        LNSTA   0,OUTPUT+?IRCL          ;Make this maximum receive
                                        ;length in I/O packet.

        ?WRITE  OUTPUT                  ;Write buffer to OUTPUT file.
        WBR     ERROR                   ;?WRITE error return.
        WLDAI   BUFLGTH,1               ;Get original buffer length.
        WSNE    0,1                     ;Is current buffer length same
                                        ;as original buffer length?
        WBR     WRITER                  ;Yes. Get another buffer.
        WSUB    2,2                     ;No.  Done.  Set for normal
                                        ;return.

        ?RETURN                         ;Return to CLI.
        WBR     ERROR                   ;?RETURN error return.

;Buffers, message, packets in unshared code.

        .NREL

;Buffer declarations.

BUFLGTH = 16384.                        ;Need to change only this
        BUFLGTH                         ;for residual characters after
                                        ;end of file.
BUF1:   .BLK    (BUFLGTH+1)/2           ;Size of BUF1
        BUFLGTH                         ;for residual characters after
                                        ;end of file.
BUF2:   .BLK    (BUFLGTH+1)/2           ;Size of BUF2.

                                        ;Mailbox for message.

MAILBOX: 0

;?GTMES packet to get input and output filenames.

GPKT:   .BLK    ?GTLN                   ;Allocate enough space for
                                        ;packet.

        .LOC    GPKT+?GREQ              ;Request type.
        .WORD   ?GARG                   ;Put argument in ?GRES only.
```

```
        .LOC   GPKT+?GNUM
        .WORD  2                        ;Argument 2 is input filename.

        .LOC   GPKT+?GRES

        .DWORD FNAME*2                  ;Set byte pointer to receive
                                        ;buffer.

        .LOC   GPKT+?GTLN               ;End of packet.

;?OPEN and I/O packet for input task.

INPUT:  .BLK   ?IBLT                    ;Allocate enough space for
                                        ;packet.

        .LOC   INPUT+?ISTI              ;File specifications.
        .WORD  ?ICRF!?RTDY!?OFIN        ;Change format to dynamic-length
                                        ;records and open for input
                                        ;only.

        .LOC   INPUT+?IMRS
        .WORD  -1                       ;Default physical block size
                                        ;to 2 Kbytes.

        .LOC   INPUT+?IBAD
        .DWORD BUF1*2                   ;Set byte pointer to record
                                        ;I/O buffer.

        .LOC   INPUT+?IRCL
        .WORD  BUFLGTH                  ;Record length is BUFLGTH.

        .LOC   INPUT+?IRLR
        .WORD  0                        ;Set to 0 (used by ?READ and
                                        ;?WRITE only).

        .LOC   INPUT+?IFNP
        .DWORD FNAME*2                  ;Set byte pointer to pathname.

        .LOC   INPUT+?IDEL
        .DWORD -1                       ;Use default delimiters: null,
                                        ;NEW LINE, form feed, and
                                        ;carriage return (default is
                                        ;-1).

        .LOC   INPUT+?IBLT              ;End of packet.

;?TASK packet for output task (minimum packet).

TPKT:   .BLK   ?DSLTH                   ;Allocate enough space for the
                                        ;standard packet.

        .LOC   TPKT+?DLNK
        .WORD  1                        ;Set to 1 for standard packet.
```

```
        .LOC    TPKT+?DLNL                  ;Reserved.
        .WORD   0                           ;Set to 0.

        .LOC    TPKT+?DLNKB                 ;Reserved
        .DWORD  0                           ;Set to 0.

        .LOC    TPKT+?DPRI
        .WORD   1                           ;Assign priority 1 to the new
                                            ;task (default is 0, which
                                            ;assigns the new task the same
                                            ;priority as the caller).
        .LOC    TPKT+?DID
        .WORD   2                           ;Assign TID 2 to the new
                                            ;task (default is 0, which
                                            ;does not assign a TID to
                                            ;the new task).
        .LOC    TPKT+?DPC
        .DWORD  WRITER                      ;Task's starting address is
                                            ;WRITER.

        .LOC    TPKT+?DAC2
        .DWORD  0                           ;There is no message for the
                                            ;new task.

        .LOC    TPKT+?DSTB
        .DWORD  STACK                       ;Stack base address is STACK.

        .LOC    TPKT+?DSFLT
        .WORD   -1                          ;Use default stack fault
                                            ;handler in URT32.LB (default
                                            ;is -1).

        .LOC    TPKT+?DSSZ
        .DWORD  60.                         ;Stack size is 60 words.

        .LOC    TPKT+?DFLGS                 ;Task flag word.
        .WORD   0                           ;Set to 0.

        .LOC    TPKT+?DRES                  ;Reserved.
        .WORD   0                           ;Set to 0.

        .LOC    TPKT+?DNUM
        .WORD   1                           ;Create one task.

        .LOC    TPKT+?DSLTH                 ;End of packet.

;?OPEN and I/O packet for output task.

OUTPUT: .BLK    ?IBLT                       ;Allocate enough space for
                                            ;packet.

        .LOC    OUTPUT+?ISTI                ;File specifications.
        .WORD   ?OFCR!?OFCE!?ICRF!?RTDY!?OFIO ;Delete file, recreate
                                            ;file, change format to
                                            ;dynamic-length records, and
                                            ;open for input and output.
```

```
        .LOC    OUTPUT+?IMRS            ;Physical block size (in bytes).
        .WORD   -1                      ;Block size is 2 Kbytes
                                        ;default is -1).

        .LOC    OUTPUT+?IBAD
        .DWORD  BUF1*2                  ;Set byte pointer to record
                                        ;I/O buffer.

        .LOC    OUTPUT+?IRCL
        .WORD   BUFLGTH                 ;Record length is BUFLGTH.

        .LOC     OUTPUT+?IRLR
        .WORD   0                       ;AOS/VS returns characters
                                        ;transferred (used by ?READ
                                        ;and ?WRITE only).

        .LOC    OUTPUT+?IFNP
        .DWORD  FNAME*2                 ;Set byte pointer to pathname.

        .LOC    OUTPUT+?IDEL
        .DWORD  -1                      ;Use default delimiters: null,
                                        ;NEW LINE, form feed, and
                                        ;carriage return (default is
                                        ;-1).

        .LOC    OUTPUT+?IBLT            ;End of packet.

FNAME:  .BLK    (?MXPL+1)/2             ;Filename buffer. System
                                        ;limit for number of
                                        ;characters.

STACK:  .BLK    60.                     ;60-word task stack.

        .END    BOOMER                  ;End of BOOMER program.
```

End of Chapter

Chapter 8
Using the Interprocess Communications (IPC) Facility

You can use the following system calls to perform interprocess communications:

?GCPN	Return the global port number of the target process's terminal.
?GPORT	Return the PID associated with a global port number.
?ILKUP	Return a global port number.
?IMERGE	Modify a ring field within a global port number.
?IREC	Receive an IPC message.
?ISEND	Send an IPC message.
?ISPLIT	Find the owner of a port (including its ring number).
?IS.R	Send and then receive an IPC message.
?TPORT	Translate a local port number to its global equivalent.

This chapter describes how to use the Interprocess Communications (IPC) Facility, through which processes can communicate with each other. Using the IPC Facility, you can

- 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 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' terminations.

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 the ?PVIP privilege, it must use the IPC facility as a standard peripheral device, which it can then access by device-independent I/O techniques. (See Chapter 5 for information on how to do this.) Also, you can use the connection-management facility, which is described in Chapter 8, to establish communications between processes. (Note that if 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 sends 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 the 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, and 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 the target ring), AOS/VS sends the message.

Only a specific task in the target ring can receive the IPC message; it is very important that you specify the target ring. This prevents a task in one ring from intercepting a message intended for a task that is executing in another ring.

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 the ?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 global 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 nonzero 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 terminal's associated process, you can identify its terminal port number by issuing the ?GCPN system call.

The ?ISPLIT system call extracts the ring field from a global port number, while the ?IMERGE system call permits both 16– and 32–bit users to modify the ring field within a global port number.

Typical IPC System Call Sequence

The following steps describe a typical IPC sequence:

1. The sending process uses the ?CREATE system call to create an IPC file entry (type ?FIPC) in its working directory. This file entry serves as the origin port for the message. (See Chapter 4 for a description of the ?CREATE system call.)

2. The sending process issues the ?ISEND system call and specifies the following in the header: its own local port number, the receiver's global port number, the length and address of the message buffer, and (optionally) system and user flags.

3. (optional) The receiving process issues the ?ILKUP system call to determine the sender's global port number.

4. The receiving process issues the ?IREC system call and specifies the following in the ?IREC header: its own local port number, the sender's global port number, and (optionally) user flags.

While the sequence above assumes that the sender issues the ?ISEND system call before the receiver issues the complementary ?IREC system call, the send and receive system calls need not be sequential. If there is no outstanding message for a receiver, AOS/VS either suspends the receiving task until you issue the ?ISEND system call, or returns an error (an option in the ?IREC headers). Similarly, if there is no ?IREC system call for an ?ISEND system call, AOS/VS either stores the message in the memory buffers or returns an error to the sender (an option in the ?ISEND header).

Send and Receive Headers

The ?ISEND and ?IREC headers consist of ?IPLTH words. The ?IS.R header is similar to the ?ISEND header, except that it contains an extension for receive information, because the ?IS.R system call performs both send and receive functions. The ?IS.R header consists of ?IPRLTH words. Figure 8–1 shows the structures of the IPC headers, and Table 8–1 describes each header offset.

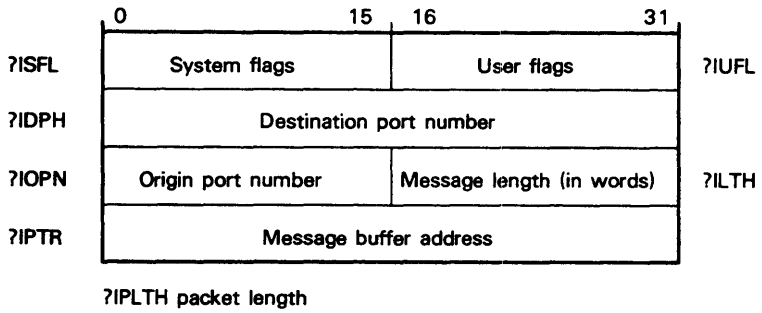
As Table 8–1 shows, the sender specifies the receiver's global port number in offset ?IDPH. When AOS/VS transmits the message, it translates this value to a local port number for the receiver and places it in offset ?IDPN of the receive header.

Similarly, the receiver specifies the sender's global port number in offset ?IOPH. AOS/VS translates this to a local port number during the transmission and records it in offset ?IOPN in the send header.

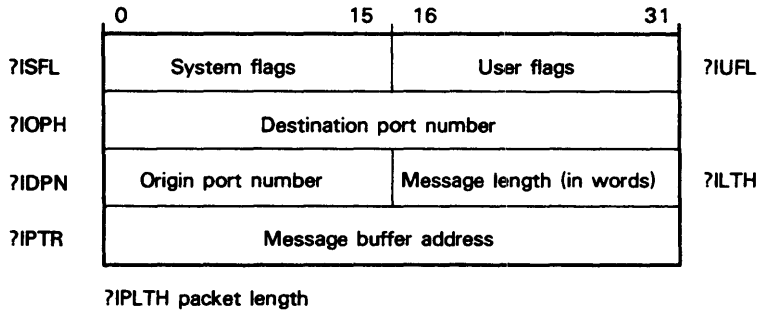
Offset ?ILTH in the send header contains the length of the IPC message, and offset ?IPTR points to the start of the message in the sender's logical address space. Within the receive header, these same offsets describe the size of the receive buffer and its starting address, respectively. AOS/VS copies the contents of these offsets from the send header to the receive header during the transmission.

If you set ?ILTH to 0 in the send header, you can use offset ?IPTR to send data directly to the header, rather than to a buffer. However, you must set up both the send and receive headers in advance.

/ISEND HEADER



?IREC HEADER



?IS.R HEADER

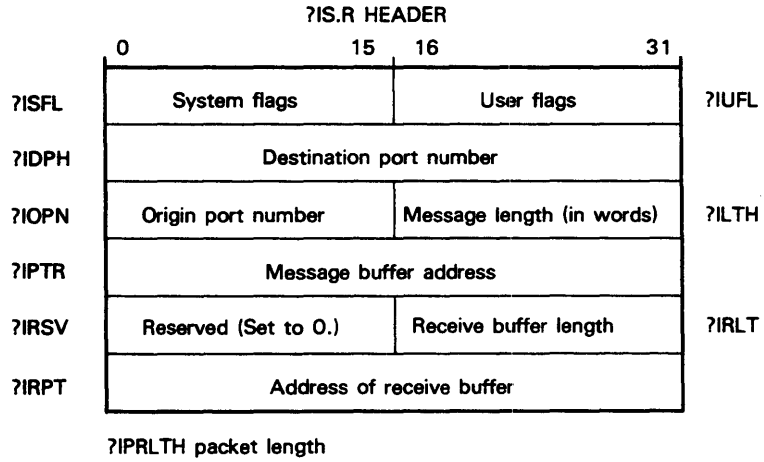


Figure 8–1 Structure of IPC Send and Receive Headers

Table 8-1. Contents of IPC Send and Receive Headers

Offset	Contents	Offset	Contents
?ISFL	System flags.	?ISFL	System flags.
?IUFL	User flags.	?IUFL	User flags (copied from send header).
?IDPH (double word)	Destination port number.	?IOPH (double word)	Origin port number.
?IOPN	Origin port number.	?IDPN	Destination port number (translated from send header).
?ILTH	Length of message in words.	?ILTH	Length of message buffer words (copied from send header).
?IPTR (double word)	Address of message Buffer.	?IPTR (double word)	Address of message buffer.
?IS.R Extension			
?IRSV	Reserved. (Set to 0.)		
?IRLT	Length of the receive buffer.		
?IRPT (double word)	Address of receive buffer.		

There is no default unless otherwise specified.

System and User Flags

In addition to the origin, destination, and message parameters, the headers for the ?ISEND and ?IREC system calls contain a system flag word (?ISFL) and a user flag word (?IUFL). Table 8–2 describes the optional contents of ?ISFL in the ?ISEND and ?IREC headers.

Table 8-2. Contents of System Flag Word (Offset ?ISFL)

Flag	Description	Flag	Description
?IFSTM	Loop the message (send the message back to the sender.)	?IFRFM	Receive a looped message (sent by this process to itself).
?IFNSP	Do not buffer the message; signal an error if there is no ready receiver.	?IFSOV	Buffer the message if the receive buffer is small.
		?IFBNK	Signal an error if there is no spooled message for this receiver.
		?IFRING	Contains the sender's ring field (returned by AOS/VS).
		?IFPR	Indicates .PR file type of sender: 0 if sender is a 32-bit process; 1 if sender is a 16-bit process (returned by AOS/VS).

A process can *loop* a message (send a message to itself). To do this, the process must perform the following steps:

1. Set bit ?IFSTM in the ?ISEND header.
2. Issue an ?ISEND system call.
3. Set bit ?IFRFM in the ?IREC header.
4. Issue an ?IREC system call.

Usually, a process loops a message for testing purposes. A processor does not need to specify the origin and destination ports in the headers for a looped message.

Bit ?IFNSP in the ?ISEND header directs AOS/VS to signal an error if there is no outstanding receiver for the sender's message.

Within the ?IREC headers, bit ?IFSOV directs AOS/VS to store the IPC message in the memory buffers if the receive buffer is too small to accommodate it. If the receiver does not set this bit and the receive buffer is too small, AOS/VS transmits as much of the message as possible and discards the overflow.

A receiver can set bit ?IFNBK to direct AOS/VS to return an error if there is no outstanding message for it. Otherwise, AOS/VS suspends the receiving task until a message is sent to the receiving process.

Bits ?IFRING and ?IFPR in the receive header provide the receiver with information about the sending process, such as the sender's ring field (?IFRING) and program type (?IFPR). AOS/VS controls these flag bits; the receiving process cannot set them.

User Flag Word

The user flag word, offset ?IUFL, serves two purposes:

- AOS/VS copies the contents of offset ?IUFL from the send header to the receive header during a transmission. Therefore, if senders and receivers set up the two headers properly, they can use offset ?IUFL to pass information.
- AOS/VS uses offset ?IUFL to pass termination and obituary message information when a process terminates or breaks a connection with another process.

Process Termination Messages

When AOS/VS terminates a process, it uses the IPC facility to send a termination message to the father process of the terminated process. AOS/VS terminates a process when

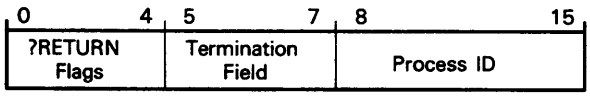
- The process issues a ?RETURN system call.
- A superior process, a process with the Superprocess privilege, or the process itself issues a ?TERM system call.
- The process encounters a user trap.

To receive the termination message, the father process must have previously issued the ?IREC system call and set offsets ?IOPH in the ?IREC header to global port number ?SPTM. The ?SPTM port is the predefined origin port for termination messages.

When a process terminates, AOS/VS writes information about the terminated process into the ?IUFL offset of the father process's ?IREC header. For some process terminations,

AOS/VS sends *only* the information in the ?IUFL offset; for most processes, AOS/VS sends a full termination message. We describe the format of these termination messages in the sections to follow.

Figure 8-2 shows the structure of the ?IUFL offset.



ID-03281

Figure 8-2. Structure of the ?IUFL Offset

Bits 0 through 4 in the ?IUFL offset contain ?RETURN flags. AOS/VS either writes a ?RETURN flag that describes the reason for the termination into this field, or it sets the field to zero. We describe the conditions under which AOS/VS performs these actions in the sections to follow.

Bits 5 through 7 in the ?IUFL offset contain the termination field. In the termination field, AOS/VS either writes a code that indicates the reason for the termination, or in the case of the ?TEXT termination code, points to an extended termination code that describes the reason for the termination. This extended termination code is always in the first word of the termination message. We describe the termination codes that AOS/VS writes to the ?IUFL offset in Table 8–3.

Table 8–3. Process Termination Codes in Offset ?IUFL for ?IREC and ?ISEND Headers

Code	Meaning
?TSELF	Either a 16-bit process terminated itself with a ?TERM or a ?RETURN system call or a 32–bit process terminated itself with a ?RETURN system call.
?TRAP	A user trap terminated a 16–bit process; the IPC message describes this trap.
?TCIN	An abort terminal interrupt (CTRL–C CTRL–B sequence) terminated a process.
?TAOS	AOS/VS terminated a process because of an error; offset ?IPTR in the IPC header contains the error code.
?TBCX	A process broke a connection that was established via the connection-management system calls. (See Chapter 9 for information on the connection-management facility.)
?TCCX	The connection still exists, but the process chained. (See Chapter 9 for information on the connection-management facility.)
?TSUP	A superior process terminated the process with a ?TERM system call.
?TEXT	An extended termination code; the extended code appears in the first word of the IPC message. A termination code of ?TEXT in the ?IUFL offset indicates that the actual termination code is a right-justified 16-bit code in the first word of the termination message. We list the extended termination codes when we describe the termination messages in the sections to follow.

If the father process is a PID-size type A process, bits 8 through 15 of the ?IUFL offset contain the PID of the terminated process. We describe the relationship between termination messages and a process's PID-size type in the following sections.

Termination Message Formats and Process PID-Size Types

The format of a termination message depends on the PID-size type of process to which AOS/VS sends the message. A process can be one of three PID-size types:

PID-Size Type Description (PID-type)

A — A executing smallPID program — can run only within the PID range of 1–256.

B — An executing hybrid program — can run only within the PID range of 1–256, *but* can create son processes running outside of that range.

C — An executing anyPID program — can run under *any* PID.

The format of the termination message that AOS/VS sends to an type A process differs from that which AOS/VS sends to a type B or C process. The format of the message can also differ depending on whether the terminated process or the father process is a 32–bit or 16–bit process.

Consequently, if you convert a smallPID program into a hybrid or anyPID program (we describe how to do this in Chapter 3), you may have to modify that program so that it can read the termination messages that AOS/VS sends to it. We describe the termination message formats for each process type in the following sections.

Termination Message Format for PID-Size Type B and C Processes

If a 32– or 16–bit process terminates and its father process is a PID-size type B or C process, AOS/VS

- Writes zeros into the PID field of the ?IUFL offset.
- Sets the termination field in the ?IUFL offset to the ?TEXT code.
- Writes the appropriate extended termination code in the first word of the termination message.
- Table 8–4 lists these termination codes.

Table 8–4. Extended Termination Messages

Code	Meaning
?XT16T	A 16–bit process has terminated itself by issuing a ?RETURN or ?TERM system call.
?XTR16	A user trap terminated a 16–bit process.
?XTCIN	An abort terminal interrupt (CTRL–C CTRL–B sequence terminated a process.
?XTSUP	A superior process terminated a process.
?XTAOS	AOS/VS terminated the process because of an error; the ?IPTR offset in the IPC header contains the error code.
XTBCX	A process broke a connection that you established through the connection-management system calls (for information on the connection-management system calls, see Chapter 9).
?XTCCX	A connection still exists, but the process chained.
?XTABR	Sent to a server process by a task within a customer process that has terminated.
?XT32T	A 32–bit process has terminated itself.
?XTR32	A user trap has terminated a 32–bit process.

The format of the termination messages that AOS/VS sends to type B and C processes is always the same, regardless of the reason for termination; however, the information that AOS/VS includes in the messages differs depending on the reason for termination. We describe the information that AOS/VS returns for each type of process termination below.

Table 8–5 describes the format for termination messages that AOS/VS sends to type B and C processes.

Table 8–5. Termination Message Format for PID-Size Type B and C Processes.

Word	Contents
0	Extended termination code.
1	Packet revision number.
2	Reserved.
3	PID of terminated process.
4 through 13	Reserved.
14 and 15	Contents of AC0. Only valid when a user trap terminated the process. High order bits (word 14) undefined when the terminated process is a 16-bit process.
16 and 17	Contents of AC1. Only valid when a user trap terminated the process. High order bits (word 16) undefined when the terminated process is a 16-bit process.
18 and 19	Contents of AC2. Only valid when a user trap terminated the process. High order bits (word 18) undefined when the terminated process is a 16-bit process.
20 and 21	Contents of AC3. Only valid when a user trap terminated the process. High order bits (word 20) undefined when the terminated process is a 16-bit process.
22	Bit 0, carry; bits 2 through 15, program counter (high-order bits). Only valid when a user trap terminated the process. This word undefined when terminated process is a 16-bit process.
23	Program counter (low-order bits). Only valid when a user trap terminated the process.
24 and 25	Elapsed time since process creation (in seconds).
26 and 27	Processor time used (in milliseconds).
28 and 29	Number of blocks read or written.
30 and 31	Page usage over processor time (page/seconds).
32 and 33	Number of page faults since process creation.
34 and 35	Number of page faults (no disk I/O).
36 through 43	Reserved.
44	Trap code. Only valid if a user trap terminated the process. To describe the trap, AOS/VS sets the bits in word 44 as follows:

(continues)

Table 8–5. Termination Message Format for PID-Size Type B and C Processes.

Word	Contents
44(Cont)	Bit 0=0 Trap occurred while control was in the user context. Bit 0=1 Trap occurred while control was in the operating system. Bit 3=1 A node time-out occurred. (This is a hardware error.) Bit 4=1 Process tried to execute a privileged instruction. Bit 5=1 Process tried to return to an inner ring from a subroutine call. (This is a violation of the ring structure.) Bit 6=1 Process tried to issue a subroutine call to an outer ring. (This is a violation of the ring structure.) Bit 7=1 Gate protection error. (This is a violation of the ring structure.) Bit 8=1 Process tried to reference an address in an inner ring. (This is a violation of the ring structure.) Bit 9=1 Process tried to read a read-protected page. Bit 10=1 Process tried to execute data in an execute-protected area. Bit 12=1 Process tried to write into a write-protected area. Bit 13=1 Memory map validity error. (The process tried to refer to an address outside the user context.) Bit 14=1 A defer error. (The process tried to use more than 16 levels of indirection in an address reference.) Bit 15=1 Process tried to issue a machine-level I/O instruction without issuing the ?DEBL system call. (See Chapter 13.)
45	Length of message passed (optional) by terminated process to father process. Set to zero when • A process terminates itself with a ?RETURN or ?TERM system call *without* passing a message to its father process. • The extended termination code is ?XTCIN (terminated by abort terminal interrupt) or ?XTSUP (terminated by superior process).
46 through n	Message area. Words 46 and 47 contain a default message, which consists of the contents of the ?IPTR offset of the IPC header. A terminating process can use the ?IPTR offset to pass a message to its father process. Following the default message, the message area contains any additional message that the terminated process passes to its father process. AOS/VS gives gives the length of this additional message in word 45. If there is no additional message, AOS/VS sets word 45 to zero.

(concluded)

Termination with a ?RETURN System Call

If a type B or C process terminates itself with a ?RETURN system call, AOS/VS places a ?RETURN code in the ?RETURN field of the ?IUFL header. We list these codes in Table 8–6.

Table 8-6. ?RETURN Codes.

Code	Meaning
?RFCF	The termination message is in CLI format (the CLI is the father).
?RFEC	AC0 contains the error code.
?RFWA	A warning condition caused the termination.
?RFER	An error condition caused the termination.
?RFAB	An abort condition caused the termination.

If the terminated process's father process is the CLI, then AOS/VS writes the ?RFCF code into the ?RETURN field. The CLI in turn displays the system error message that corresponds to the error code in AC0, and any additional messages that the terminated process specified in the ?RETURN system call.

If the terminated process's father is *not* the CLI, AOS/VS writes the ?RFEC, ?RFWA, ?RFER, or ?RFAB code into the ?RETURN field for whatever interpretation the father and son processes previously agreed.

Specifying the Termination Message Format that a Process Receives

To ensure that a process will always receive termination messages in the format that we described for type B and C processes — future revisions of AOS/VS may modify the format of the termination message — you can issue the ?TMSG system call. In the ?TMSG system call, you specify the termination message format that we described for PID-size type B and C processes with the ?TM6 termination message code.

32-Bit Termination Messages for PID-Size Type A Processes

When a 32–bit process terminates and its father is a PID-size type A process, AOS/VS

- Sets the ?RETURN flags field in the ?IUFL offset to zero.
- Writes the ?TEXT code to the termination field in the ?IUFL offset.
- Writes the appropriate extended termination code (?T32T or ?TR32) in the first word of the termination message.

If the process terminated on a ?RETURN or a ?TERM system call — not a user trap — AOS/VS sends the termination message that we describe in Table 8–7.

Table 8-7. 32-Bit Termination Message to PID-Size Type A Processes.

Word	Contents
0	?T32T Extended termination code for 32–bit self-termination.
1	Length of message sent by terminating process to its father process.
2 and 3	Error code (optional). Specified by the terminated process when issuing a ?RETURN system call.
4	First word of message sent by the terminated process to its father process (optional). Specified by the terminated process when issuing a ?RETURN system call.

If the process terminated because of a user trap, AOS/VS sends a termination message in the format that we describe in Table 8–8.

Table 8–8. Format of Termination Message Sent to a PID-Type A Process on a 32-Bit Process User Trap.

Word	Contents
0	?TR32 Extended termination code for a 32-bit user trap.
1 and 2	AC0 contents at time of trap.
3 and 4	AC1 contents at time of trap.
5 and 6	AC2 contents at time of trap.
7 and 8	AC3 contents at time of trap.
9	Bit 0, carry; bits 1 through 15, high-order bits of program counter.
10	Low-order bits of program counter.
11	To describe the trap, AOS/VS sets the bits in word 11 as follows: Bit 0=0 Trap occurred while control was in the user context. Bit 0=1 Trap occurred while control was in the operating system. Bit 3=1 A node time-out occurred. (This is a hardware error.) Bit 4=1 Process tried to execute a privileged instruction. Bit 5=1 Process tried to return to an inner ring from a subroutine call. (This is a violation of the ring structure.) Bit 6=1 Process tried to issue a subroutine call to an outer ring. (This is a violation of the ring structure.) Bit 7=1 Gate protection error. (This is a violation of the ring structure.) Bit 8=1 Process tried to reference an address in an inner ring. (This is a violation of the ring structure.) Bit 9=1 Process tried to read a read-protected page. Bit 10=1 Process tried to execute data in an execute-protected area. Bit 12=1 Process tried to write into a write-protected area. Bit 13=1 Memory map validity error. (The process tried to refer to an address outside the user context.) Bit 14=1 A defer error. (The process tried to use more than 16 levels of indirection in an address reference.) Bit 15=1 Process tried to issue a machine-level I/O instruction without issuing the ?DEBL system call. (See Chapter 13.)

Termination Messages for PID-Type A Processes — 16-Bit Sons

When a 16–bit process terminates by issuing a ?RETURN system call and its father is a type A process, AOS/VS

- Returns flag ?TSELF to the termination field in offset ?IUFL.
- Copies one or more of the codes listed in Table 8–9 to ?RETURN field of the ?IUFL offset.

Table 8-9. ?RETURN Codes.

Code	Meaning
?RFCF	The termination message is in CLI format (the CLI is the father).
?RFEC	AC0 contains the error code.
?RFWA	A warning condition caused the termination.
?RFER	An error condition caused the termination.
?RFAB	An abort condition caused the termination.

If the terminated process's father process is the CLI, then AOS/VS writes the ?RFCF code into the ?RETURN field. The CLI in turn displays the system error message that corresponds to the error code in AC0, and any additional messages that the terminated process specified in the ?RETURN system call.

If the terminated process's father process is not the CLI, AOS/VS writes the codes ?RFEC, ?RFWA, ?RFER, or ?RFAB to the ?RETURN field for whatever interpretation the father and son processes previously agreed on.

When a 16–bit process terminates itself with a ?TERM system call, AOS/VS returns either the termination message specified by the process, or, if the process did not specify a message, one of the termination codes. AOS/VS sends the termination message directly to the father's receive buffer. It sends the termination code to the ?IUFL termination field in the father's receive buffer, and writes zeros into the ?RETURN field.

If the 16–bit process terminated because of a user trap, AOS/VS sets the father's ?IUFL termination field to ?TRAP, and sends the father one of the six-word termination messages listed in Table 8–10.

Table 8-10. Termination Message Format for 16-Bit Process User Traps.

Word	Contents
0	AC0 contents at time of trap.
1	AC1 contents at time of trap.
2	AC2 contents at time of trap.
3	AC3 contents at time of trap.
4	Bit 0, carry; Bits 1 through 15, program counter value.
5	To describe the trap, AOS/VS sets the bits in word 5 as follows: Bit 0=0 Trap occurred while control was in the user context. Bit 0=1 Trap occurred while control was in the operating system. Bit 12=1 Process tried to write into a write-protected area. Bit 13=1 Memory map validity error. (The process tried to refer to an address outside the user context.) Bit 14=1 A defer error. (The process tried to use more than 16 levels of indirection in an address reference.) Bit 15=1 Process tried to issue a machine-level I/O instruction without issuing the ?DEBL system call. (See Chapter 10.)

If the 16–bit process terminated because of an abort terminal interrupt (a CTRL-C CTRL-B sequence) or a ?TERM system call issued by a superior process, AOS/VS returns the proper code to the father's ?IUFL termination field (?TCIN or ?TSUP), but does not send a message.

?ISEND and ?IREC System Call Logic

The flowcharts in Figures 8–3 and 8–4 show the sequence of operations for the ?ISEND and ?IREC system calls, respectively.

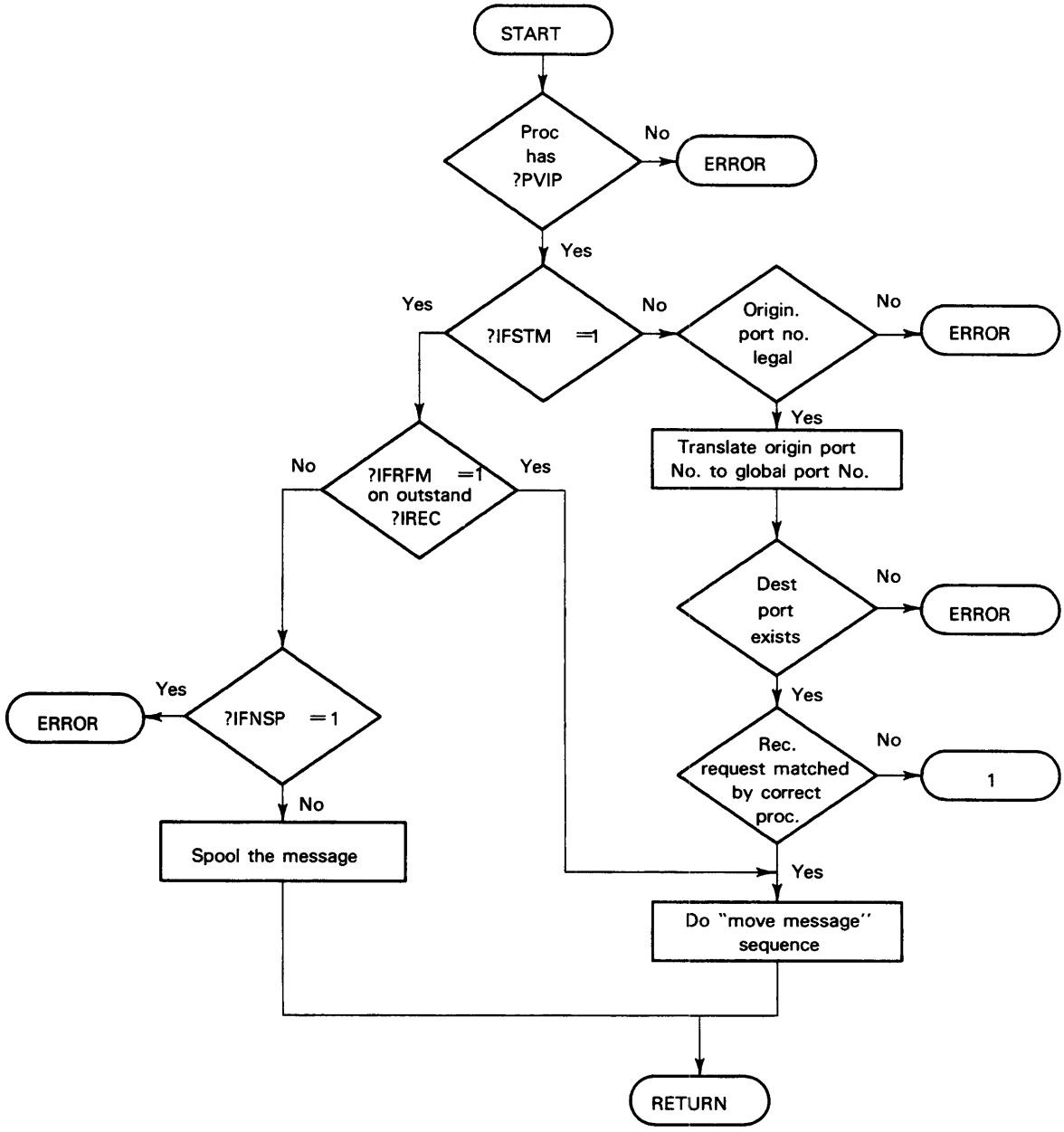


Figure 8–3. ?ISEND Logic Flowchart

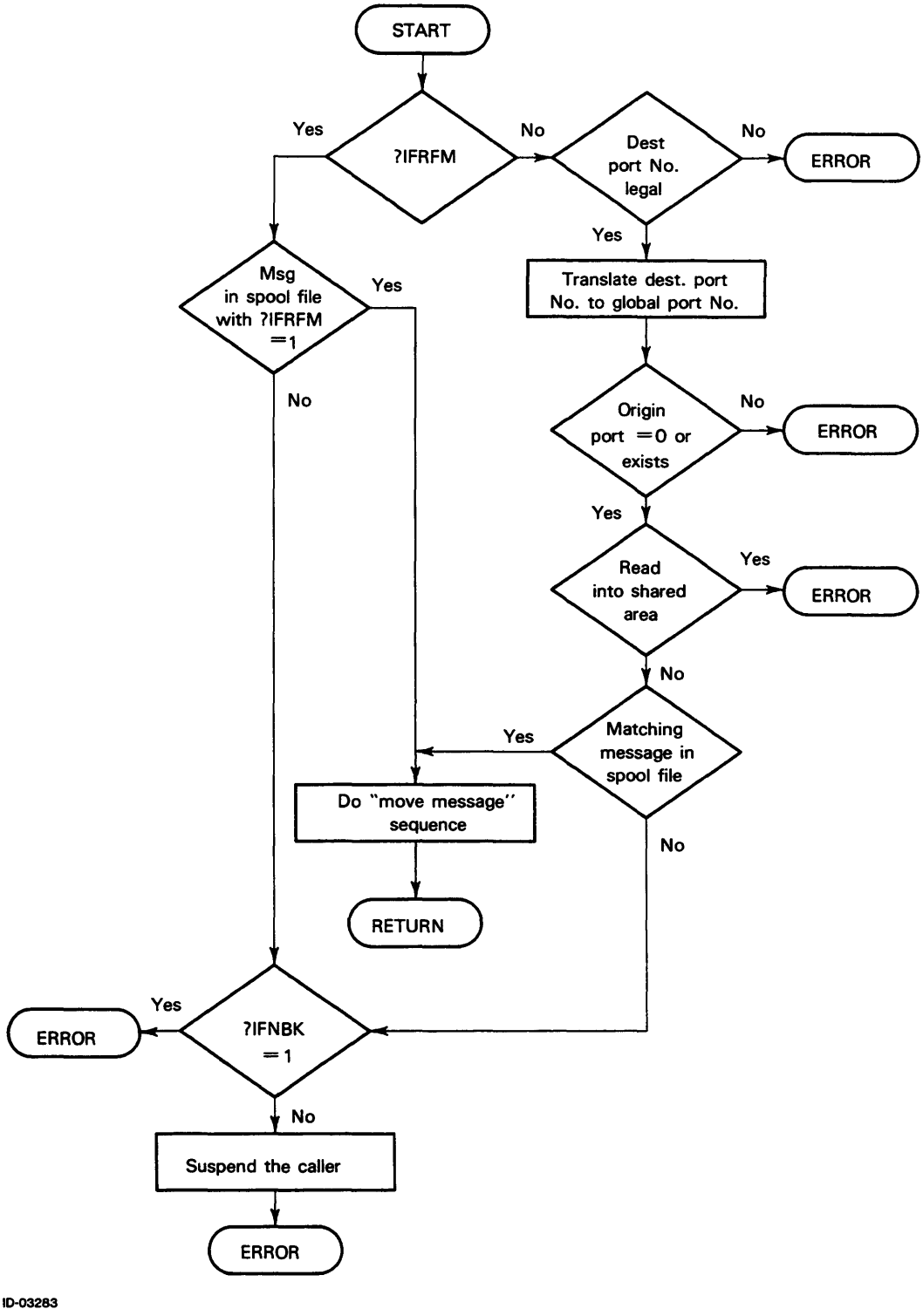


Figure 8–4. ?IREC Logic Flowchart

Sample IPC Programs

The following programs, SPEAK and HEAR, illustrate interprocess communications with the IPC system calls ?ILKUP, ?IREC, and ?ISEND.

Program SPEAK uses routine SON (see the Processes and Memory Sample Programs) to execute program HEAR. HEAR issues an ?IREC system call to receive a message from SPEAK. Then, SPEAK issues ?ISEND to send the message to HEAR. HEAR and SPEAK both use the ?ILKUP system call to discover the other's port number.

The HEAR Program

```
;Open terminal (CON) for input and output.  (See Chapter 6 for more
∞formation on ?OPEN.)

HEAR:   ?OPEN   CON                 ;Open terminal (CON) for I/O.
        WBR     .ERROR              ;Report error and quit.

mod     ?WRITE  CON                 ;Display message on terminal screen
                                    ;(byte pointer is already in
                                    ;I/O packet).
        WBR     .ERROR              ;?WRITE error return.

§art the SON process to run SPEAK.PR.

        XLEFB   0,SPEAK*2           ;Get byte pointer to filename.
        XJSR    @.SON               ;SON creates process.

;SPEAK is running.  Create IPC entry for receive.

        XLEFB   0,PORTR*2           ;Set byte pointer to port name.
        ?CREATE IPCEN               ;Create IPC entry PORTR.
        WBR     .ERROR              ;?CREATE error return.
        XLEFB   0,MES1*2            ;Set byte pointer to message.
        XWSTA   0,CON?IBAD          ;Put in I/O packet.
        ?WRITE  CON                 ;Write message to terminal.
        WBR     .ERROR              ;Try to handle the error.

;See if SPEAK's entry and its ports have been created.

GETOP:  XLEFB   0,PORTS*2           ;Set byte pointer to port name.
        ?ILKUP                      ;Get port number from AC1.
        WBR     TEST                ;Try to handle the error.
        XLEFB   0,MES2*2            ;Get byte pointer.
        XWSTA   0,CON+?IBAD         ;Put in I/O packet.
```

```
        ?WRITE  CON                     ;Display success message on
                                        ;terminal screen.
        WBR     .ERROR                  ;Try to handle the error.
        XWSTA   1,RHDR+?IOPH            ;Put origin port number in
                                        ;record header (note wide
                                        ;storage).
        NLDAI   1,0                     ;Generate 1.
        XNSTA   ORHDR+?IDPN             ;Put destination port 1 in
                                        ;record header (note narrow
                                        ;storage).

        ?IREC   RHDR                    ;Receive SPEAK message.
        WBR     .ERROR                  ;?IREC error message.
        XLEFB   1,MSBUF*2               ;Message received. Get byte
                                        ;pointer to message buffer.
        WLDAI   ?RFCF!100.,2            ;Put flag and 100 words for
                                        ;message in AC2.

        ?RETURN                         ;Return to CLI with message.
        WBR     .ERROR                  ;Try to handle the error.

;?ILKUP error.  Check for error code ERFDE (file does not exist) and
;delay if present.

TEST:   WLDAI   ERFDE,1                 ;Put ERFDE number in AC1.
        WSEQ    0,1                     ;Skip if error code is ERFDE.
        WBR     .ERROR                  ;Try to handle the error.
        XLEFB   1,MES3*2                ;Get byte pointer to message.
        XWSTA   0,CON+?IBAD             ;Put in I/O packet.

        ?WRITE  CON                     ;Display message on terminal.
        WBR     .ERROR                  ;Try to handle the error.
        WLDAI   5000.,0                 ;5 seconds.

        ?WDELAY                         ;Wait for 5 seconds.
        WBR     .ERROR                  ;Try to handle the error.
        WBR     GETOP                   ;Do ?ILKUP again.

;Error instruction, byte pointer, filename, and port name.

.ERROR: XJMP    ERROR                   ;To error handler.

.SON:   SON                             ;To subroutine SON.

SPEAK:  .TXT    "SPEAK.PR"              ;Filename of program.

PORTR:  .TXT    "PORTR"                 ;Name of receive port.

PORTS:  ;.TXT    "PORTS"                ;Name of send port.

;?OPEN and I/O packet for terminal.

CON:    .BLK    ?IBLT                   ;Allocate enough space for
                                        ;packet.
        .LOC    CON+?ISTI               ;File specifications.
```

```
        .WORD   ?ICRF!?RTDS!?OFIO      ;Change format to
                                        ;data-sensitive records and
                                        ;open for input and output.
        .LOC    CON+?IMRS
        .WORD   -1                      ;Physical block size is 2
                                        ;Kbytes.

        .LOC    CON+?IBAD               ;Set byte pointer to record I/O
        .DWORD  MES*2                   ;buffer.

        .LOC    CON+?IRCL
        .WORD   120.                    ;Record length is 120
                                        ;characters.

        .LOC    CON+?IFNP               ;Set byte pointer to pathname.
        .DWORD  CONS*2

        .LOC    CON+?IDEL               ;Delimiter table address.
        .DWORD  -1                      ;Use default delimiters: null,
                                        ;NEW LINE, form feed, and
                                        ;carriage return (default is
                                        ;-1).
        .LOC    CON+?IBLT               ;End of packet.

;Filename, buffer, messages.  A .NOLOC 1 follows.

CONS:   .TXT    "@CONSOLE"              ;Use generic name.

MES:    .TXT    "From HEAR--I have opened the terminal and I am ready
                to call SON.<12>"

MES1:   .TXT    "From HEAR--I am back from SON. SPEAK is running. <12>
                I created an IPC entry.<12>"

MES2:   .TXT    "From HEAR--Have ?ILKUPed the IPC port entry.<12>"

MES3:   .TXT    "From HEAR--?ILKUP error.  I will wait and then try
                again.<12>"

        .NOLOC 0

;Header for IPC entry.

IPCEN:  .LOC    IPCEN+?CFTYP
        .WORD   ?FIPC                   ;IPC file.

        .LOC    IPCEN+?CPOR
        .WORD   1                       ;Port number is 1.

        .LOC    IPCEN+?CTIM
        .DWORD  -1                      ;Use default current time.

        .LOC    IPCEN+?CACP
        .DWORD  -1                      ;Use default current ACL.
```

```
;?IREC Receive header RHDR.

RHDR:   .LOC    RHDR+?ISFL
        .WORD   0                       ;There are no system flags.

        .LOC    RHDR+?IUFL
        .WORD   0                       ;There are no user flags.

        .LOC    RHDR+?IOPH
        .DWORD  0                       ;AOS/VS returns origin port
                                        ;number here.

        .LOC    RHDR+?IDPN
        .WORD   0                       ;AOS/VS returns destination
                                        ;port number here.

        .LOC    RHDR+?ILTH
        .WORD   100.                    ;Message buffer is 100 words.

        .LOC    RHDR+?IPTR
        .DWORD  MSBUF                   ;Message buffer address.

        .LOC    RHDR+?PLTH              ;End of ?IREC header.

MSBUF:  .BLK    101.                    ;Message buffer.

;Error handler.

ERROR:  WLDAI   ?RFEC!?RFCF!?RFER,2     ;Error flags: Error code is
                                        ;in AC0 (?RFEC), message is in
                                        ;CLI format (?RFCF), and
                                        ;caller should handle this as
                                        ;an error (?RFER).

        ?RETURN                         ;Return to CLI.
        WBR     ERROR                   ;?RETURN error return.

      .END    HEAR                      ;End of HEAR program.
```

The SPEAK Program

The following program, SPEAK, sends an IPC message to another process. Then, SPEAK terminates itself. SPEAK's origin port name is PORTS; its destination port name is PORTR.

```
                    .TITLE  SPEAK
                    .ENT SPEAK
                    .NREL

;Create and IPC entry port named PORTS.

SPEAK:  XLEFB   0,PORTS*2           ;Set byte pointer to port name.
        ?CREATE IPCEN               ;Create an IPC port.
        WBR     .ERROR              ;Try to handle the error.

;See if PORTR, the receive port, has been created.

GETNM:  XLEFB   0,PORTR*2           ;Set byte pointer to port name.
        ?ILKUP                      ;Put port number in AC1.
        WBR     TEST                ;Does the port exist?
        XWSTA   1,SHDR+?IDPH        ;Yes. Put port number in send
                                    ;header.
        NLDAI   1,0                 ;No. Generate 1.
        XNSTA   0,SHDR+?IOPN        ;Put destination port 1 in
                                    ;send header (narrow storage).
        ?ISEND  SHDR                ;Send SPEAK message.
        WBR     .ERROR              ;Try to handle the error.

;The message has been sent.  Wait for other process to receive message
;before terminating yourself.

        ?WLDAI  10000.,0            ;10 seconds.
        ?WDELAY                     ;Wait for 10 seconds.
        WBR     .ERROR              ;Try to handle the error.
        NLDAI   -1,0                ;Get -1 to terminate yourself.
        WSUB    2,2                 ;There is no IPC message to
                                    ;the father.
        ?TERM                       ;Terminate.
        WBR     .ERROR  ;Try to handle the error.

;?ILKUP error. Check to see whether the error code is ERDNE (Does Not
;Exist). If the error code is ERDNE, wait.

TEST:   WLDAI   ERFDE,1             ;Put error code number in AC1.
        WSEQ    0,1                 ;Was error code ERFDE?
        WBR     .ERROR              ;No. Try to handle the error.
        WLDAI   5000.,0             ;5 seconds.
        ?WDELAY                     ;Yes. Wait for 5 seconds.
        WBR     .ERROR              ;Try to handle the error.
        WBR     GETNM               ;Do ?ILKUP again.
```

```
;Error instructions, pointer, filenames, and port names.

.ERROR: XJMP    ERROR                   ;To error handler.

PORTS:  .TXT    "PORTS"                 ;Name of send port.

PORTR:  .TXT    "PORTR"                 ;Name of receive port.

;Header for IPC entry.

IPCEN:  .LOC    IPCEN+?CFTYP
        .WORD   ?FIPC                   ;IPC file.

        .LOC    IPCEN+?CPOR
        .WORD   1                       ;Port number is 1.

        .LOC    IPCEN+?CTIM
        .DWORD  -1                      ;Default to current time.

        .LOC    IPCEN+?CACP
        .DWORD  -1                      ;Default to current ACL.

;?ISEND send header SHDR.

SHDR:   .LOC    SHDR+?ISFL
        .WORD   0                       ;There are no system flags.

        .LOC    SHDR+?IUFL
        .WORD   0                       ;There are no user flags.

        .LOC    SHDR+?IDPH
        .DWORD  0                       ;AOS/VS returns destination

        .LOC    SHDR+?IOPN
        .WORD   0                       ;AOS/VS returns origin
                                        ;port number here.

        .LOC    SHDR+?ILTH
        .WORD   100.                    ;Message buffer is 100 words.

        .LOC    SHDR+?IPTR
        .DWORD  MSBUF                   ;Message buffer address.

        .LOC    SHDR+?PLTH              ;End of ?ISEND header.
```

```
;Message that we want to send. A .NOLOC 1 follows.

MSBUF:  .TXT    "Hello.  This is your son speaking. As you read <12>
                these words, I am terminating and so are you.<12>

        .NOLOC  0                       ;Resume listing everything.

;Error handler.

ERROR:  WLDAI   ?RFEC!?RFCF!?RFER,2     ;Error flags: Error code is
                                        ;in AC0 (?RFEC), message is in
                                        ;CLI format (?RFCF), and
                                        ;caller should handle this as
                                        ;an error (?RFER).
        ?RETURN                         ;Return to CLI.
        WBR     ERROR                   ;?RETURN error return.

        .END    SPEAK                   ;End of SPEAK program.
```

End of Chapter

Chapter 9
Creating and Managing Process Connections

You can use the following system calls to create and manage process connections:

?CON	Become a customer of a specified server.
?CTERM	Terminate a customer process.
?DCON	Break a connection (disconnect) in Ring 7.
?DRCON	Break a connection (disconnect) in a specified ring.
?MBFC	Move bytes from a customer's buffer.
?MBTC	Move bytes to a customer's buffer.
?PCNX	Pass a connection from one server to another in Ring 7.
?PRCNX	Pass a connection from one server to another in a specified ring.
?RESIGN	Resign as a server.
?SERVE	Become a server process.
?SIGNL	Signal another task.
?SIGWT	Signal another task and then wait for a signal.
?VCUST	Verify a customer in Ring 7.
?VRCUST	Verify a customer in a specified ring.
?WTSIG	Wait for a signal from another task or process.

This chapter describes how to create and manage customer/server relationships — called connections — between processes.

Why Use Process Connections?

When you create a customer/server relationship between processes, you can use the server process to perform certain functions on behalf of its customer processes. Typically, a server process performs general routines that customer processes can access. For example, 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.

Creating a Process Connection

To make a connection between two processes you must specify one process to be the server process and the other to be the customer process. To specify the server and customer processes, issue

- The ?SERVE system call to specify the calling process to be the server process.
- The ?CON system call to specify the customer and establish the logical connection between the customer and an existing server.

Figure 9–1 shows a server process with connections to three customer processes.

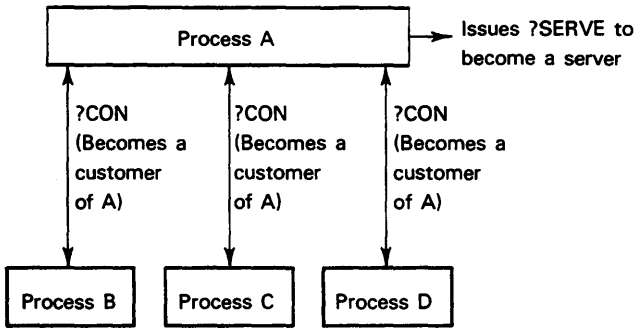


Figure 9–1. Model Customer/Server Configuration

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. Figure 9–2 shows a multilevel connection, where process A is the server of processes B, C, and D, and a customer of process X. Multilevel connections let you set up intermediate servers for some functions, and one or more superior servers for other functions.

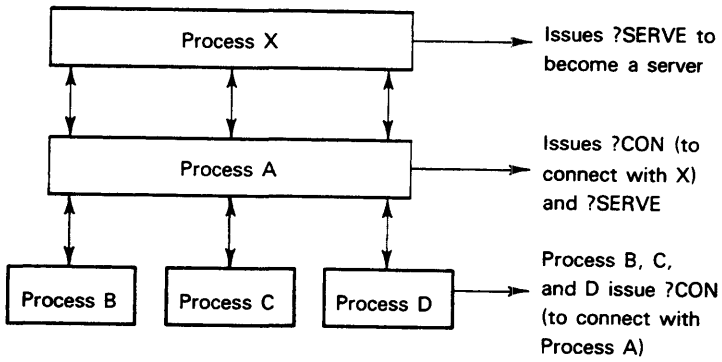


Figure 9–2. Multilevel Customer/Server Configuration

Creating a Double Process Connection

You can also make a double connection between two processes. A double connection allows each process to act as either the customer or the server of the other, depending on the action to be performed. As Figure 9–3 illustrates, a double connection requires two ?SERVE system calls and two ?CON system calls. AOS/VS creates two connection-table entries, one for each ?CON system call.

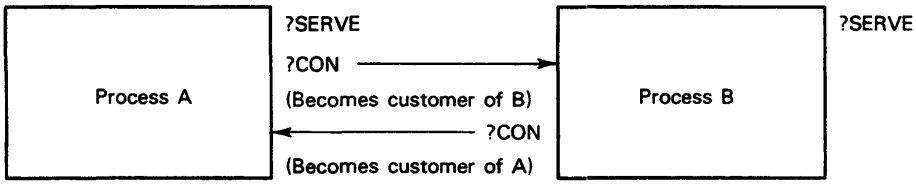


Figure 9–3. Double Connection

The Server Process

Once a process has server status (established with the ?SERVE system call), it can issue the following system calls:

?CTERM	Terminates a customer.
?MBFC	Moves bytes from a customer's buffer.
?MBTC	Moves bytes to a customer's buffer.

?PCNX Passes a connection from one server to another in Ring 7.

?PRCNX Passes a connection from one server to another in a specified ring.

?RESIGN Resigns as a server.

?VCUST Verifies a customer in Ring 7.

?VRCUST Verifies a customer in a specified ring.

The ?CTERM system call terminates a customer process. The ?RESIGN system call signals AOS/VS that the caller has resigned as a server.

The ?MBTC and ?MBFC system calls allow the server to move bytes to or from a customer's logical address space. However, before AOS/VS executes either of these system calls, it checks the connection table to make sure that there is a valid connection between the two processes, and that the customer's buffer is in the ring defined at connect time, which must be in the caller's ring or in a higher ring. Also, there must be enough space at the destination for the data to reside entirely within the specified destination ring.

The ?PCNX system call passes a customer/server connection from one server to another in Ring 7 and directs AOS/VS to revise the connection-table entry accordingly. The ?PRCNX system call is similar to the ?PCNX system call, except the ?PRCNX system call is not restricted to Ring 7. Both the ?PCNX and the ?PRCNX system calls are useful for passing a valid customer from a dispatching server to a specialized server process.

The ?VCUST system call determines whether a target process in Ring 7 is a customer of the ?VCUST caller. The ?VRCUST system call is similar to the ?VCUST system call, except the ?VRCUST target process need not be in Ring 7. If the ?VCUST or the ?VRCUST target process is not a customer, AOS/VS takes the error return and passes error code ERCDE to AC0. If the connection between the two has been broken, the system call fails on error code ERCBK.

Typically, server processes communicate with their customers via the IPC system calls ?SEND, ?IREC, and ?IS.R. However, they can also use the fast interprocess communication system calls, ?SIGNL, ?WTSIG, and ?SIGWT, to communicate with their customers. (See the Fast Interprocess Synchronization section in this chapter for more information on the ?SIGNL, ?WTSIG, and ?SIGWT system calls.)

Terminating Process Connections

AOS/VS breaks the customer/server connection when a process traps or when the process issues one of the following calls:

?CTERM Terminates a customer (a server-only system call).

?DCON Breaks a connection in Ring 7.

?DRCON Breaks a connection in a specified ring.

?RESIGN Resigns as a server (a server-only system call).

?TERM Terminates a process (self-terminates).

Notice that the ?CTERM system call is a server-only system call. The ?DCON, ?DRCON, and ?TERM system calls are available to both servers and customers. (See Chapter 3 for information on terminating processes with the ?TERM system call.)

When AOS/VS detects a broken connection, it sets a flag bit in the appropriate connection table entry. For AOS/VS to actually clear the entry, however, it must receive disconnects from both the customer and the server. For example, a customer could issue a ?DCON system call to break its connection with the server, but the PIDs of both processes will remain in the connection table until the server issues a ?DCON, ?RESIGN, or ?TERM (self-termination) system call.

You should issue disconnects from both processes as soon as a connection has served its purpose. This keeps the connection-table entries within the maximum range and allows AOS/VS to reassign the PIDs. (The maximum number of connections allowed under AOS/VS is revision dependent.)

Obituary Messages at Disconnection

When a customer or server process disconnects, AOS/VS sends the other process an obituary message. An obituary message is a zero-length IPC message (an IPC header, essentially). A customer can suppress the obituary message by setting bit ?COBIT in AC1 when it issues the ?CON system call.

To receive an obituary message, a process must issue the ?IREC system call; it must specify 0 and ?SPTM in ?IREC offsets ?IOPH and ?IOPL, respectively (origin port), and 0 in offset ?IDPN (destination port). AOS/VS returns the obituary message as termination code ?TBCX in offset ?IUFL of the ?IREC header.

Obituary Message Formats

The format of the obituary message that AOS/VS sends differs depending on the type of process receiving the obituary message. AOS/VS sends obituary messages in a different format to A–type processes (executing smallPID programs) than it does to B– or C–type processes (executing hybrid or anyPID programs, respectively).

To A–type processes, AOS/VS sends obituary messages that give

- The PID of the disconnected process in the second word of the obituary message (bits 8 through 15 of the ?IUFL offset).
- The connection bit map in the last word of the obituary message (for 32–bit processes — bits 8 through 15 of the ?ILPN offset; for 16-bit processes — bits 8 through 15 of the ?IPTR offset).

To B– or C–type processes, which require a larger PID field, AOS/VS sends obituary messages that give

- The PID of the disconnected process in the last word of of the obituary message (for 32–bit processes — all 16 bits of the ?ILPN offset; for 16–bit processes — all 16 bits of the ?IPTR offset).
- The connection bit map in second word of the obituary message (bits 8 through 15 of the ?IUFL offset).

Inner-Ring Connection Management

Segment images that are loaded into different user rings within the sample process often have very different aims and identities. Consequently, the connection-management facility identifies all connections as being between ordered pairs of PID/ring-within-PID tandems (called *PID/ring tandems*). A ring within a process can be connected as a customer (and/or as a server) with multiple rings that are within another process or processes.

Although multiple ?CON system calls that connect the same ordered pairs of PID/ring tandems are legal, they will result in only a single connection. However, connections between rings that are within the same process are illegal.

For a server, the privilege to move bytes to and from a customer is limited to only those rings in the customer that are higher than or equal to the lowest ring that issued a ?CON system call to create a connection to the server. Every IPC message (obituary message, chain, etc.) issued by the connection-management facility, is sent to the ring from which the ?CON or ?SERVE system call was issued. The system flag word of the IPC header holds a field, ?IFRING, that contains the ring number of the segment image that caused the system to generate the message.

If a server is concurrently connected to multiple rings within the customer, AOS/VS indicates the status of those connections with a single IPC message. This prevents the race conditions that might occur if AOS/VS issued multiple messages.

For 32–bit receivers, flag bits are returned in the ?IPTL word of the IPC header. For 16–bit receivers (that is, tasks in Ring 7 of a 16–bit process), the flag bits are returned in the ?IPTR word of the IPC header. The flag bits include both a single *explicit disconnect* flag and a bit map that contains the connection status of the various inner rings.

The explicit disconnect flag expands the information that the *connection broken* (?TBCX) termination message contains when it is going to a server on a customer process termination. If the explicit disconnect bit is set in the connection broken termination message, then one of the rings of the customer process issued a ?DCON or a ?DRCON system call to break the connection. If the explicit disconnect bit is not set, then one of the following caused the broken connection:

- A customer process terminated.
- A customer process chained, but it did not have a connection in its Ring 7.

  A connection broken (?TBVC) rather than a *customer chained* (?TCCX) termination message describes this special case of a customer process chain, but it is also valid for a server process chain. All other types of process chain events cause customer chained messages (?TCCX), because Rings 4 through 6 are *unloaded* when Ring 7 chains. (Effectively, Rings 4 through 6 terminate on a Ring 7 chain.)

The meaning of individual bits within the bit map depends upon the type of event being signaled:

- When a customer is chained, bits set in the bit map indicate which rings were connected before the chain. In this case, AOS/VS automatically preserves the connections to Ring 7 and 3, providing they existed before the chain.
- When a connection is broken, if the explicit disconnect bit is set, then the bits set in the bit map indicate rings to which there are remaining connections. If the explicit disconnect bit is clear, then the bits set in the bit map indicate which rings were connected before the termination or chain.

The following parameters characterize the bit flags:

Parameter	Meaning
?CXMBM	Word mask that allows you to extract both the explicit disconnect flag and the connection bit map.
?CXMED	Bit mask for the explicit disconnect flag.
?CXBBM0	Bit position of the explict disconect flag. (The explicit disconnect flag immediately precedes the connection bit map portion of the ?IPTL or ?IPTR word.) ?CXBBM0 + N defines the position of the bit that corresponds to Ring N within the bit map. (Rings 1 through 7 are mapped in the bit map.) You can point to the explicit disconnect bit as if it were the first bit in the bit map (that is, N = 0).
?CXBVED	Position of the explicit disconnect bit within the extracted word.

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 sychronization facility that common local servers can use to pend and unpend tasks, depending on the state of databases 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 system 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.

End of Chapter

Chapter 10
Managing a Multiprocessor Environment

You can use the following system calls to manage a multiprocessor environment:

?JPINIT	Initialize a job processor.
?JPMOV	Move a job processor from one logical processor to another.
?JPREL	Release a job processor from the system.
?JPSTAT	Get status information on a single job processor, or on all job processors.
?SYSPRV	Enable, disable, or get status of System Manager mode, Superprocess mode, or Superuser mode.

This chapter describes how to manage a multiprocessor system. It describes the processor configuration that you get when you initialize the AOS/VS system. It also describes how to initialize and release processors from the system, and how to reconfigure the processor configuration.

Processor Configuration after AOS/VS System Initialization

When you initialize the AOS/VS system, you initialize a single default processor, called the *mother* processor. If your computer has other processors, you can then initialize these processors, called *child* processors.

The AOS/VS system processes run on only the mother processor. All other processes can run on both the mother and child processors.

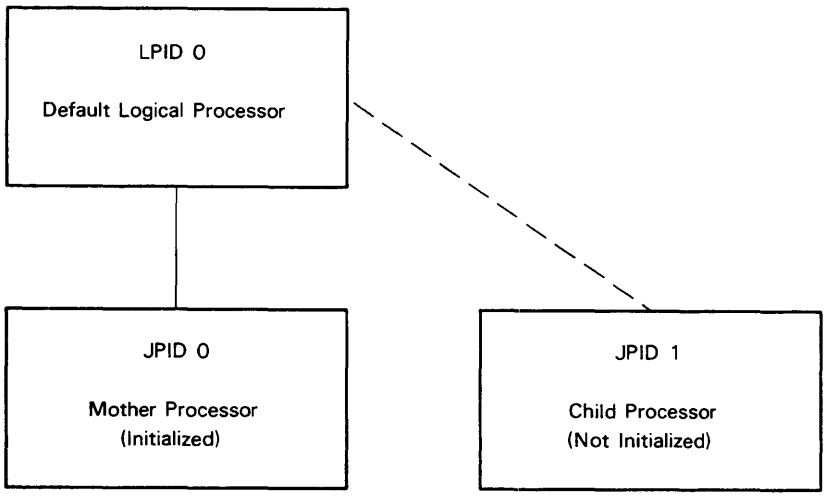
At system initialization, AOS/VS attaches the mother processor, which is a *job (physical) processor*, to a single default *logical processor*. A logical processor is a scheduling arrangement: it specifies the way in which the job processors attached to it are to schedule processes. The default logical processor specifies standard scheduling: when an attached job processor becomes free, that processor runs the highest priority, ready process.

After system initialization, you can initialize any child processor on the system and attach it to the default logical processor (for more information, see the "Initializing a Job Processor" section in this chapter).

If you enable class scheduling, which we describe how to do in the next chapter, you can create other logical processors. You can use these logical processors to schedule processes differently from the default logical processor.

To identify the physical processors on the system, the hardware returns an ID number, called a JPID, to AOS/VS at system initialization. The mother processor will usually have a JPID of 0; a child processor will, in turn, have a JPID of 1 through 15.

Similarly, AOS/VS identifies each logical processor on the system by giving it an ID number, called an LPID. The default logical processor always has an LPID of 0. (See Figure 10–1.)



ID-03314

Figure 10–1 Processor Configuration after AOS/VS System Initialization

Entering System Manager Mode

To issue the system calls that initialize and release job processors, the calling process must be in System Manager mode. To put a process into System Manager mode, you must have the System Manager privilege in your user profile. If you pass this privilege on to your son processes, the son processes can turn System Manager mode on by issuing the ?SYSPRV system call. Once in System Manager mode, a process can initialize or release job processors from the system.

Initializing a Job Processor

To initialize a job processor, issue the ?JPINIT system call. The ?JPINIT system call attaches the job processor to a logical processor that you specify and, optionally, loads a microcode file into the job processor. If the job processor has a some characteristic that makes it different from other processors — for example, it has a floating-point processor and the others do not — you will need to load a specific microcode file into it. After you attach the job processor to the logical processor, the logical processor begins scheduling processes to run on it.

To specify the job processor that you want to initialize, and the logical processor to which you want to attach it, you include each processor's ID (JPID and LPID) in the system call packet.

Specifying a Microcode File

The ?JPINIT system call allows you to either use the microcode that is currently in the job processor (loaded at system generation *or* by an earlier ?JPINIT system call), or to load a microcode file that you specify into the job processor. If you choose to load another microcode file into the job processor, that file can be

- A microcode file that you specify.
- A standard microcode file — the system determines which microcode file to load based on the model number of the job processor.

Attaching a Job Processor to Another Logical Processor

If you have enabled class scheduling and created another logical processor, you can attach a job processor to that logical processor by

- Attaching an inactive job processor to the logical processor when you initialize it (described above).
- Moving an active job processor from one logical processor to another.

To move a job processor from one logical processor to another, issue the ?JPMOV system call. After you move a job processor, it begins to schedule processes according to the new logical processor's scheduling arrangement.

To specify the job processor that you want to move, and the logical processor to which you want to attach it, you must include each processor's ID (JPID and LPID) in the system call packet.

The ?JPMOV system call also allows you to set a bit that prevents you from moving a job processor when it is the *only* job processor attached to the logical processor. If you set this bit, the system will return an error when you try to move the only job processor attached to the logical processor. If you do not set this bit, the system *will* move the job processor, creating an unattached, or orphaned logical processor. If class scheduling is enabled, those processes that receive processor time solely through this logical processor will then be unable to run.

Releasing a Job Processor from the AOS/VS System

To release a job processor from the AOS/VS system, and disable it from further processing, issue the ?JPREL system call. The system will remove the job processor from its logical processor, making the job processor inactive.

When you issue the ?JPREL system call, the system returns an error under the following conditions:

- The job processor that you specify (through its JPID) is not active (uninitialized).
- The job processor that you specify is running a vital system task, such as performing disk I/O, servicing the file system, and so on.

Like the ?JPMOV system call, the ?JPREL system call allows you to set a bit that prevents you from releasing a job processor when it is the *only* job processor attached to the logical processor. If you set this bit, the system will return an error should you try to release the only job processor attached to the logical processor. Again, if you do not set this bit, the

system *will* release the job processor, creating an unattached, or orphaned logical processor. If class scheduling is enabled, any processes that receive processor time through this logical processor exclusively will then be unable to run.

End of Chapter

Chapter 11
Creating and Managing a Class Scheduling Environment

You can use the following system calls to create and manage a class scheduling environment

?CLASS	Get or set class IDs and names.
?CLSTAT	Return status of class scheduling.
?CMATRIX	Get or set class scheduling matrix.
?CLSCHED	Enable or disable class scheduling.
?LPCLASS	Get or set class assignments and the process interval for a logical processor.
?LPCREA	Create a logical processor.
?LPDELE	Delete a logical processor.
?LPSTAT	Get status information on a logical processor.
?SYSPRV	Enter, leave, or get the status of System Manager mode, Super-process mode, or Superuser mode.

This chapter describes how to create and manage a class scheduling environment. It describes what a class is and why you might want to use class scheduling on your system. It then describes how to create and manage a class scheduling environment.

What Is a Class?

A *class* is a set of processes for which you want special scheduling treatment. Usually, this treatment involves allocating a percentage of processor time to these processes.

When you create a process, AOS/VS assigns that process to a class. The class to which AOS/VS assigns a process depends on the process's *user locality* and *program locality*. A process receives its user locality from your user profile, which specifies both a default user locality, and any other user localities to which you can assign a process. A process receives its program locality, which you specify using the selective preamble editor (SPRED), from the header of its program file.

A process's user and program localities are its *locality pair*. You can use the system calls that we describe later in this chapter to specify which locality pairs run under each class.

User and Program Localities

You can define classes using up to 16 user and program localities, which range from 0 through 15.

Let's say you want a class for privileged users and that you want all processes of user locality 1 to run in this class. You could create this class so that the locality table, or *class matrix*, looks as shown in Figure 11–1.

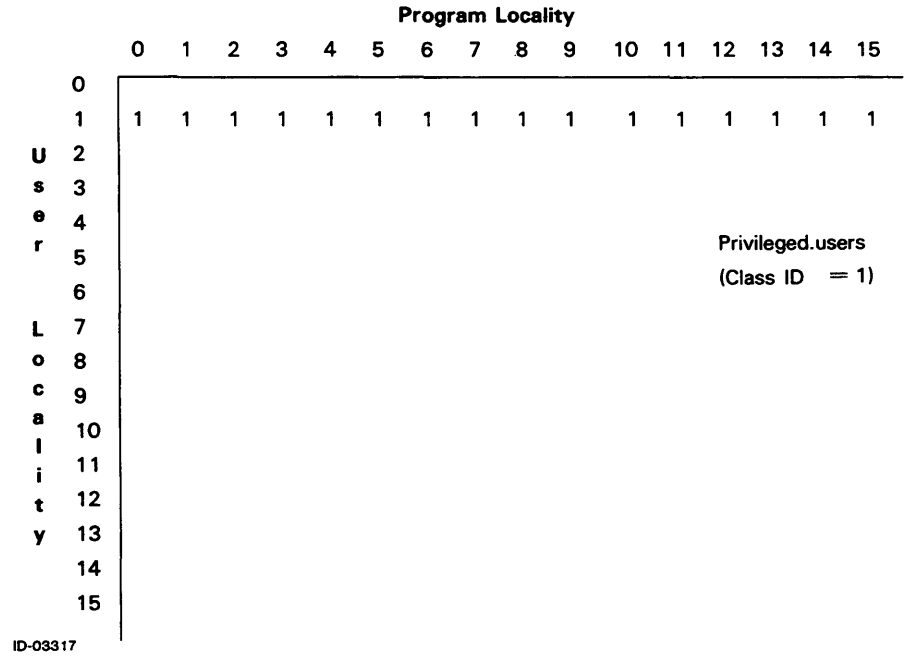


Figure 11–1. A Class for Privileged Users

After creating the Privileged.users class, you could then allocate a specific amount of processor time to that class, say 40 percent.

A user process can then join this class by assuming a user locality of 1. (The program locality doesn't matter, since the class is defined in all 16 program localities. The determining factor is the user, not the program.) To start a user's process in the privileged class, the system manager can run PREDITOR on the user's profile and specify a default user locality of 1. Or, the system manager can have the user start in another class and allow that user to join the privileged class by changing their locality to 1.

Why Use Class Scheduling?

Class scheduling gives you more control over processor usage than you ordinarily have under standard scheduling. Using class scheduling, you can tailor process scheduling to better fit your system's processing needs.

What Is Standard Scheduling?

If class scheduling is not enabled — by default, it is not — the system uses standard scheduling. Under standard scheduling, when a processor becomes free, the system runs the highest priority, ready process. This process can then use the processor for a *subslice period* (32 milliseconds, usually). The system will wait until the end of the subslice period to reschedule, unless

- The process encounters a blocking event.
- Another higher priority process becomes ready.

Again, the system will run the highest priority, ready process. This process may be the same process that just ran or another process.

Under standard scheduling, processes receive one of two types of scheduling: round-robin or heuristic scheduling.

Under round-robin scheduling, the system tries to give each process of the same priority an equal amount of time. A process that uses a lot of processor time isn't penalized.

Under heuristic scheduling, the system may reduce a process's internal priority, based on its behavior. The system penalizes a process that uses a lot of processor time — relative to other, more interactive processes of the same priority.

The kind of scheduling that a process receives depends on its group. The priority of the process (range 0 to 511) determines which group it's in (for more information on priority groups, see Chapter 3). The system schedules the process groups as follows:

- *Group 2 processes receive heuristic scheduling* (penalized for heavy processor demand). Within any group 2 priority level, the system favors interactive processes. Noninteractive (compute-bound) processes get reduced internal priority. Heuristic *favoring* often gives interactive processes better response time than they'd get under round-robin scheduling. By default, all user processes (and their sons) are group 2 processes.
- *Group 1 and group 3 processes receive round-robin scheduling.* Compute-bound processes in these groups aren't penalized; however, the response time of interactive processes with the same priority may be slower. A process's internal priority doesn't change, regardless of the amount of processor time it has consumed.

Why Use Standard Scheduling?

Standard scheduling is meant for general-purpose systems. Since, by default, processes start in group 2, standard scheduling favors highly interactive timesharing processes over compute-bound processes. But this interactive bias can be overcome, if you want to give user processes a priority that makes them group 1 or group 3 processes. Standard scheduling also allows high priority processes (like the PMGR) to get all the time they need without penalty.

Compute-bound processes of the same priority also tend to get equal amounts of processor time, regardless of group. This works well in situations where you want to treat compute-bound processes equally.

Why Use Class Scheduling?

Standard scheduling, based on priority only, has two limitations.

The first limitation of standard scheduling is that you can't give one compute-bound process preference over another without starving the lower priority process.

If you want to give one compute-bound process preference over others, and you give it higher priority, the process can consume all available processor time. AOS/VS gives control to the highest priority, ready process, so when the process is ready, AOS/VS will give it control. If the process is always ready (as it will be if it is always computing and doing little I/O), then it will get all available processor time. AOS/VS will give it subslice after subslice, while lower priority processes get no time.

For example, say two compute-bound processes are running at priority 5. Each process will get approximately 50 percent of the available processor time. If you change the priority of one process to 4 (a higher priority), it will get about 100 percent of available time while the other gets no time. Standard scheduling provides no middle ground between even distribution (here, 50/50) and monopoly (here, 100/0).

The second limitation of standard scheduling appears when you give compute-bound processes a lower priority than interactive ones (you might do this to give users faster response time). With lower priority, the compute-bound processes would get *no* time — there's no way to give them even a tiny percentage of time. Thus compute-bound jobs might take hours or even days to complete.

For example, say a system has 60 timesharing users (which represent interactive processes) and two batch streams (which represent compute-bound processes). By default, the batch streams get a lower priority than the user processes. This increases the turnaround time of each batch job and increases the number of jobs stacked on the queue. Batch processing can occur at night — but for those jobs that require daily turnaround, this isn't acceptable. Standard scheduling offers no good solution to this problem.

With classes, you can specify the percentage of time for processes in a class. AOS/VS will then try to give that class the specified amount of processor time. This overcomes both limitations above.

As an additional benefit, you can force processes to run in a specific class, whether they are compute bound (batch) or interactive (user). Class scheduling extends your control over standard scheduling.

Entering System Manager Mode

To issue the system calls that enable and manage class scheduling, the calling process must be in System Manager mode. To put a process into System Manager mode, you

must first have the System Manager privilege in your user profile. You can then turn on System Manager mode by issuing the ?SYSPRV system call. Once in System Manager mode, you can issue any of the class-scheduling system calls.

Enabling Class Scheduling

To enable class scheduling, you issue the ?CLSCHED system call. If you have not yet specified the classes that you want on the system, or the way in which you want to schedule them (we describe how to do this later in the chapter) the ?CLSCHED system call will enable the system's default class-scheduling arrangement.

Under the default class-scheduling arrangement, all processes are in the same class, to which the system gives 100 percent of the available processor time. When a processor becomes ready, the system gives control of that processor to the highest priority, ready process. The default class-scheduling arrangement is effectively the same as standard scheduling. However, class scheduling — even under the default scheduling arrangement — requires some additional system overhead.

Disabling Class Scheduling

You can disable class scheduling at any time by issuing a second, complementary ?CLSCHED system call. When you disable class scheduling, the system continues to maintain the classes and scheduling arrangements that you previously specified. You can modify these class scheduling specifications — even though class scheduling is *not* enabled — by issuing other class scheduling system calls. When you issue another ?CLSCHED system call, the system will enable class scheduling with any modifications that you have made.

Getting the Status of Class Scheduling

You can also use the ?CLSCHED system call to find out if class scheduling is or is not enabled. To do this, you must set the Get word in the ?CLSCHED system call packet.

Class Scheduling Arrangements — Logical Processors

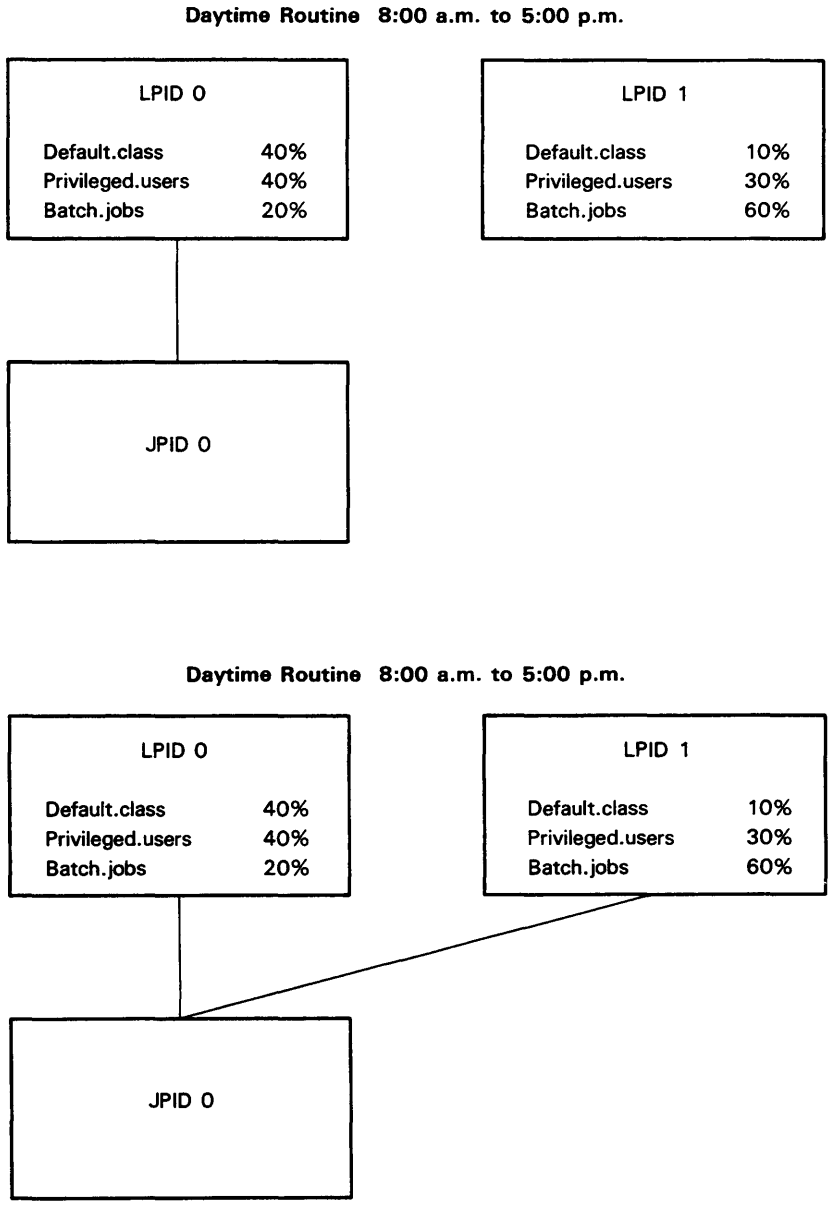
The AOS/VS system allows you to create *logical processors*, which are class-scheduling arrangements. A logical processor, which allocates processor time between several classes (usually), is only active when it is attached to a job processor (physical processor).

At system startup, AOS/VS attaches a default logical processor (LPID 0) to the mother processor (JPID 0, usually). If your computer has multiple processors, you can also attach these processors to the default logical processor. When any one of these attached processes becomes free, it runs the highest priority, ready process. (We describe how to attach job processors to logical processors in Chapter 10.)

In addition to the default class, the AOS/VS system allows you to create up to 16 classes, and to create up to 16 logical processors to schedule these classes. You then can move a job processor from one logical processor to another. This allows you to activate different scheduling arrangements (logical processors) for different environments — perhaps based on time of day.

For example, Figure 11–2 shows one application of class scheduling using two logical processors and only one job processor.

Figure 11–2 shows a logical processor that favors the default and privileged classes (interactive users) during the working day, and another logical processor that favors batch jobs at night.



ID-03315

Figure 11–2. Using Different Logical Processors with One Job Processor

With two job processors, there's even more flexibility. Figure 11–3 shows an application similar to that in Figure 11–2, but with two job processors.

Figure 11–3 shows two job processors connected to logical processor LPID 0 during the day — providing maximum processing power for LPID 0. At night, one of the job processors is moved to another logical processor — giving the processes in classes on both logical processors computing time.

Creating and Changing Process Classes

To create process classes — the first step in creating class scheduling arrangements — or to change one or more existing classes, issue the ?CLASS system call.

The ?CLASS system call creates classes by allowing you to specify which of the class ID values, which range from 0 to 15, are valid.

If you set the Get word in the ?CLASS system call packet, the system will return a list of valid class IDs.

Assigning Classes to User and Program Localities

After you have specified which of the 16 classes are valid, you can use the ?CMATRIX system call to assign those classes to user and program localities in the *class locality matrix*. The class locality matrix, of which there is an example in Figure 11–1, is a 16 by 16 table with 256 cells, in which user and program locality values range from 0 to 15.

You can also use the ?CMATRIX system call to read the current class assignments in the table.

Modifying the Class Locality Matrix

If you change the class locality matrix, the changes that you make will *not* effect the class membership of any existing process. However, any new process will get its class membership from the newly modified class-scheduling matrix.

If, after having changed the class scheduling matrix, it becomes necessary to change the class membership of an existing process, you can change the process's user locality to that of a different class by issuing the ?LOCAITY system call.

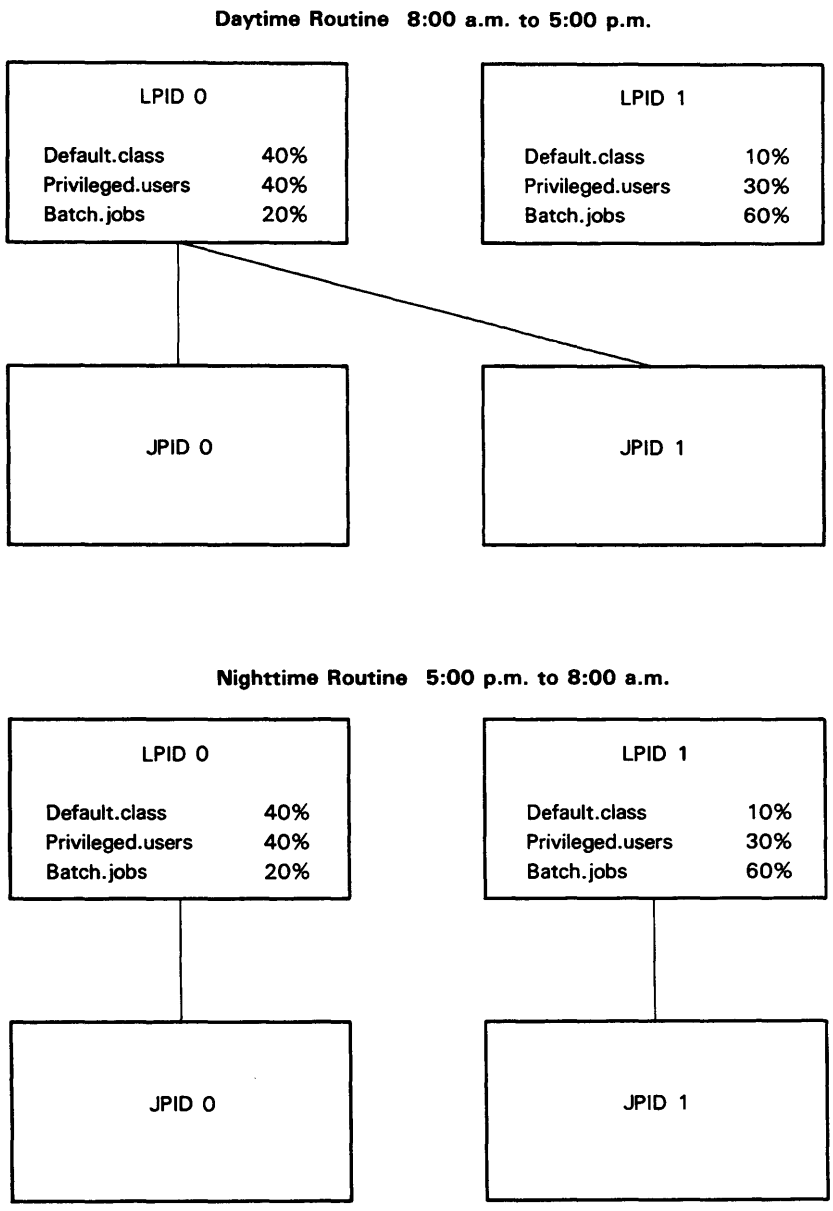
Adding and Deleting Classes

You can add and delete valid class IDs by issuing the ?CLASS system call. However, the system will not delete a class, but return an error, under the following conditions:

- The class appears in the class-scheduling matrix.
- The class is class 0, the default class.

Creating and Managing Logical Processors

To create a logical processor, you issue the ?LPCREA system call. The system will then create the logical processor and return its LPID to you. You can then specify the class assignments for that logical processor and, if you want to activate the logical processor, connect a job processor to it.



ID-03316

Figure 11–3. Using Different Logical Processors with Two Job Processors

Specifying Class Assignments for the Logical Processor

To specify the class assignments for a logical processor, which determine the process classes in the scheduling arrangement and how much time each class gets, issue the ?LPCLASS system call.

The ?LPCLASS system call allows you to include up to 16 classes in the logical processor's scheduling arrangement. You can schedule these classes as

- *Primary Classes*, for which you specify a percentage of processor time. If the primary class has processes ready to run, it can use up to the percentage of processor time that you specify.
- *Secondary Classes*, which use any remaining time not used by the primary classes. Secondary classes compete for processor time by level — the first secondary class that you specify in the ?LPCLASS call has a higher priority than the next secondary class, the second a higher priority than the third, and so on.

A process's class *type* — primary or secondary — can affect the amount of processor time that the process gets. For example, when a processor becomes free, the highest priority, ready process gets control, *unless*

- The process belongs to a primary class that has consumed its percentage and another primary-class process is ready.
- The process belongs to a secondary class and a higher-level secondary class or a primary class is ready.

In either of the previous cases, the process can't get time regardless of its priority.

For a primary class example, take the class Privileged.users, shown in earlier figures. The processes in this class are user processes, so assume they're scheduled heuristically. Normally, heuristic scheduling penalizes processes that use a lot of processor time. But these user processes belong to a primary class allotted 40 percent of processor time. So, these users will be allowed to use up to 40 percent if they can possibly use it. They will get faster response time at the expense of other processes (but if these privileged processes can't use the time, it will be given to other primary– or secondary-class processes).

Or, say there are two compute-bound processes to run. You want one process to get twice as much time as the other. Without classes (the limitation described above), you couldn't do this. *With* classes, you need only set up a primary class for each process (perhaps defined by program locality); then allot the preferred class twice the percentage of processor time as the other. For example — depending on the importance of these processes — you could give the preferred process class 75 percent, and the other process class 25 percent.

Specifying the User Process Interval

The system enforces class percentages over a constant *interval*, measured in tenths of seconds, which you specify in the ?LPCLASS system call. At the end of each interval, AOS/VS reinitializes the percentages, regardless of how much time each class has used.

The default interval is 4 seconds. In the 75/25 arrangement that we described previously, for 4 seconds AOS/VS would enforce a 75 percent limit on one class and a 25 percent limit on the other class. If one class reaches its limit, its processes become ineligible until the next interval.

The default interval of 4 seconds is a good general-purpose value. But if you have a class with interactive processes (like users in text editors) that tends to exhaust its percentage,

all the class's processes may appear blocked for a second or so during each interval. You can eliminate such delays in service by specifying a shorter interval, such as 1 or 2 seconds.

The system also reinitializes the class percentages (freeing classes that consumed their percentages) when the connected job processor(s) become idle. This allows primary-class processes that have exhausted their class percentage to run if no other primary-class process wants time.

Specifying Primary Class Percentages

When the sum of the class percentages that you have specified equals 100 percent, the system allows each class to use *up to* its percentage. Generally, we suggest that you maintain a sum of 100 percent — not less, not more — for primary classes on any logical processor. A sum of less than 100 percent may result in needless scheduling overhead; a sum of more than 100 percent may result in some classes using all of the processor time, starving others.

One exception to the sum 100 percent rule involves high priority server processes, such as EXEC, CEO®, XODIAC™, INFOS® II, and so on. If you create a class that includes these server processes, the total percentage for that logical procesor can exceed 100 percent.

For example, you could give the server class 100 percent, and other class a total of 80 percent. This allows the server processes to get as much time as they need. While servers typically don't use a lot of processor time, and will not ordinarily use much of the 100 percent, if they need the time, they can get it. However, you'd still need to give the server processes high priority, since you want them to get processor time first from among the classes that have time remaining.

Within primary classes whose percentage limit hasn't been reached, AOS/VS uses standard scheduling, as described above. AOS/VS also uses standard scheduling in logical processors for which class scheduling hasn't been enabled.

Generally, classes and logical processors are most useful when you want to deal with one or more exceptional processes. You can leave unexceptional processes alone, as part of the default class (Default.class) supplied with AOS/VS.

Getting Class Scheduling Information

You can use the ?LPCLASS system call to find out which classes are running on a logical processor. If you set the Get word in the system call packet, the ?LPCLASS system call will return information on the classes in that logical processor's scheduling arrangement.

Deleting a Logical Processor

If you want to delete a logical processor — remove it from the AOS/VS system — issue the ?LPDELE system call.

The system does not allow you to delete a logical processor that has a job processor connected to it. To delete a connected logical processor, you must release the job processor(s) from the logical processor before you can issue the ?LPDELE system call. We describe how to attach and release job processors in Chapter 10.

Getting Logical Processor Status

The ?LPSTAT system call allows you to get general information about the logical processors on the system, or information on a particular logical processor. Specifically, the ?LPSTAT system call returns

- The number of logical processors that are currently on the system and their LPIDs.
- The number of job processors attached to a specific logical processor that you specify and the JPIDs of the job processors.

End of Chapter

Chapter 12
Using and Managing AOS/VS System Resources

The system calls that you can use to manage AOS/VS system resources are

?BNAME	Get host location of a process or queue.
?CDAY	Convert a scalar date value.
?CTOD	Convert a scalar time value.
?ENQUE	Queue a file entry.
?ERMSG	Read the error message file.
?EXEC	Request a service from EXEC.
?FDAY	Convert the date to a scalar value.
?FEDFUNC	Interface to AOS/VS File Editor (FED) utility.
?FTOD	Convert the time of day to a scalar value.
?GDAY	Get the current date.
?GHRZ	Get the frequency of the system clock.
?GSID	Get the system identifier.
?GTNAM	Return symbol closest in value to specified input value.
?GTOD	Get the time of day.
?GTSVL	Get the value of a user symbol.
?ITIME	Return the AOS/VS-format internal time.
?LOGCALLS	Log system calls.
?LOGEV	Enter an event in the system log file.
?PROFILE	Create, delete, rename, read, or modify a system profile.
?SDAY	Set the system calendar.
?SINFO	Get selected information about the current operating system.
?STOD	Set the system clock.

This chapter describes how to use and manage various resources of the AOS/VS system. For example, this chapter describes how to set the system's calendar and clock, and how to get the time and date from the clock. It describes how to request service from the system's EXEC utility, how to manage the system's user profiles, how to use and update the system's error message file, how to update the system's log file, and so on.

Using the System's Clock and Calendar

AOS/VS maintains a 24–hour clock and a calendar. During the system-generation procedure, you can set the clock to any one of several real-time frequencies.

Depending on your application, you may need to know the real-time frequency of the system clock while your program is executing. The ?GHRZ system call returns this information to AC0 as a code in the range from 0 through 4. Each digit of the code corresponds to a specific frequency.

The system clock expresses the current time in seconds, minutes, and hours; the values for seconds and minutes range from 0 through 59, and the value for the hour ranges from 0 (midnight) through 23 (11 p.m.). You can issue the ?ITIME system call to get an AOS/VS-format time-stamp.

The system calendar expresses the current date as day, month, and year. To determine the year, the system calendar subtracts the base year 1900 from the current year and converts the result to octal. The notation for 1980, for example, is 120 octal.

The system calls ?STOD and ?SDAY set the system clock and calendar, respectively. The ?GTOD and ?GDAY system calls return the current time and date, respectively.

In some cases, such as in the ?FSTAT packet, AOS/VS returns the time and/or date as a scalar value. In scalar notation, the current time equals the number of biseconds that have elapsed since midnight. The date equals the number of days that have elapsed since 31 December 1967. The ?CTOD system call converts a scalar time to seconds, minutes, and hours. The ?CDAY system call converts a scalar date to month, day, and year. To convert time and date back to scalar values, issue the ?FTOD and ?FDAY system calls, respectively.

Using the EXEC Utility

In general, the EXEC utility manages queues and magnetic tape units. Because the EXEC utility can perform many functions for you, you must issue the ?EXEC system call to tell it what to do. Specifically, the ?EXEC system call directs the EXEC utility to perform one of the following functions on behalf of a calling process:

- Assign or deassign a logical name to a tape unit or an uninitialized disk that you want to treat as a whole unit (i.e., a non–LD disk) and issue an operator mount or dismount message.
- Mount labeled or unlabeled magnetic tapes.
- Dismount labeled or unlabeled magnetic tapes.
- Place a request into a queue.
- Hold, unhold, or cancel a queue request.
- Provide an report on the status of the EXEC utility.

The ?EXEC system call requires a packet: To direct the EXEC utility to perform one of these functions, you must specify in offset ?XRFNC (the first offset) what you want the EXEC utility to do. Although each function requires a unique packet, the ?XRFNC offset is common to all packets.

If your system is not running the EXEC utility, you can still queue files for spooled output by issuing the ?ENQUE system call.

Using the File Editor (FED) Utility

Using the ?FEDFUNC system call, you can direct the File Editor (FED) utility to perform one of the following functions for the calling process:

- Change the radix.
- Open a symbol table file.
- Evaluate a FED string.

The ?FEDFUNC system call requires a unique packet for each function. You must define the function you want the FED utility to perform for you in the first offset, ?FRFNC, which is common to all of the call's packets.

Accessing Symbol Tables

The ?GTNAM system call lets you refer to the system-defined symbol table (.ST) file without knowing its contents. This means that if you do not know the symbol for a particular value, you can issue the ?GTNAM system call to search the .ST file for the symbol that is closest in value to the value you supplied in AC0. The ?GTSVL system call is similar to the ?GTNAM system call, but it allows you to refer to a particular program's user-defined .ST file without knowing its contents, instead of the system-defined .ST file.

Managing User Profiles

The AOS/VS system allows you to create, delete, modify, and examine user profiles by issuing the ?PROFILE system call. To issue the ?PROFILE system call, the calling process must have the Superuser privilege and be in Superuser mode.

When you issue the ?PROFILE system call, you can perform one of the following functions:

Function	Mnemonic	Use
CREATE	?PFCRE	Create a profile and associate a username with it.
DELETE	?PFDEL	Delete a profile and its associated username.
RENAME	?PFREN	Rename the profile by associating a new username with the profile.
INITIATE ACCESS	?PFIAC	Initiate access to an existing profile. *This function must precede the next two functions: READ FIELD and UPDATE FIELD.*

Function	Mnemonic	Use
READ FIELD	?PFRDF	Return the contents of one or more fields in a profile.
UPDATE FIELD	?PFUFD	Change the contents of one or more fields in a profile.
TERMINATE ACCESS	?PFTAC	Terminate access to an existing profile. *This function closes the profile to further READ FIELD and UPDATE FIELD functions until you issue the next INITIATE ACCESS function.*

Reading and Updating the Error Message File (ERMES)

The system file ERMES contains all the error codes, their corresponding mnemonics, and their text messages. There are 20000 (octal) groups of error codes for AOS/VS (including user programs). Data General Corporation reserves code groups 0 through 77 (octal) and 200 through 7777 (octal) for the system. You can define the remaining groups, numbered 100 through 177 (octal) and 10000 through 17777 (octal).

The error codes are 32–bit unsigned values. Each error code contains two fields: a group field, and an error code field. If an error occurs when a system call is executing, AOS/VS returns the error code value to AC0. Each error is associated with a unique text string.

The ?ERMSG system call returns the text string associated with a particular error code. Before you issue the ?ERMSG system call, you must specify the error code in AC0.

To add error codes to the ERMES file, you must obtain its source version, allocate an unused code group (or add to an existing code group), and insert your own series of codes and descriptive messages. You can also create a new error message file that is structured like ERMES, but has different contents.

Writing to the System Log File

AOS/VS maintains a special accounting file, :SYSLOG, with the special file type ?FLOG. You can log messages into :SYSLOG. The ?LOGEV system call accesses this file. The ?LOGEV system call writes an event code and, optionally, a message to the log file. A process must be in Superuser mode to issue the ?LOGEV system call.

For more information about the structure of the system log file, see Appendix A in this book.

Getting System, Process, and Queue Locations

If your system is part of a network and you need to find out on which system you are running, you can issue the ?GSID system call. To determine the location of a process or queue — whether it is on the local host or a remote host — issue the ?BNAME system call.

Getting the System's Revision Number

Between each major release, AOS/VS may undergo several revisions. Therefore, it is important to know your system's revision number and its memory configuration. The ?SINFO system call allows you to get this information.

End of Chapter

Chapter 13
Supporting User Devices

You use the following system calls to support user devices:

?DDIS	Disable access to all devices.
?DEBL	Enable access to all devices.
?IDEF	Define a user device.
?IMSG	Receive an interrupt message.
?IRMV	Remove a user device.
?IXIT	Exit from an interrupt service routine.
?IXMT	Send an interrupt message.
?LEFD	Disable LEF mode.
?LEFE	Enable LEF mode.
?LEFS	Return the current LEF mode status.
?STMAP	Set the data channel map.

This chapter describes how to define user devices to the AOS/VS system, and how to enable them.

Because AOS/VS supports a wide variety of user devices, such as magnetic tape drives, disk drives, and line printers, you usually define these devices during the system-generation procedure. However, a process that has the ?PVDV privilege can define and enable devices at program execution time. For more information on defining devices at system generation, see *How to Generate and Run AOS/VS.*

The devices that you define and enable during the system-generation procedure are *system-defined devices.* The devices that a process with the ?PVDV privilege defines and enables at execution time are *user-defined devices.*

This chapter describes the system calls that allow you to use both system- and user-defined devices.

AOS/VS supports a maximum of 64 system-defined and/or user-defined devices per I/O controller (IOC). You can use any device code in the range from 1 through 63, as long as you do not use codes that are already in use. (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.* Each device has an interrupt service routine that AOS/VS uses to handle the interrupts that the device sends. See Figure 13–1.

The DCT is ?UDLN words long. Note that AOS/VS returns most of the DCT parameters as output to the ?IDEF system call. However, you must perform the following steps:

1. Supply the address of the interrupt service routine (offset ?UDVIS).
2. Supply the address of the power-failure/auto-restart routine or, if you do not want to use such a routine, –1 (offset ?UDDRS).
3. Provide the interrupt service mask (offset ?UDVMS).

To remove a user device, issue the ?IRMV system call.

?IDEF System Call Requirements

To issue the ?IDEF system call, a process *must* be resident. Additionally, all memory used by the interrupt service routine and the power-failure/auto-restart routine must also be resident in memory. Consequently, before a process can issue the ?IDEF system call, the process must

1. Issue the ?CTYPE system call to ensure that it is a resident process.
2. Issue the ?WIRE system call to wire the following into memory:
   - The interrupt service routine.
   - The power-failure/auto-restart routine.
   - All memory that the interrupt service and power-failure/auto-restart routines access.
   - The user stack.
3. Issue the ?AWIRE system call to wire the AOS/VS Agent's code for the ?IXMT system call into memory. (We describe the ?IXMT system call, which transmits an interrupt message, later in this chapter.

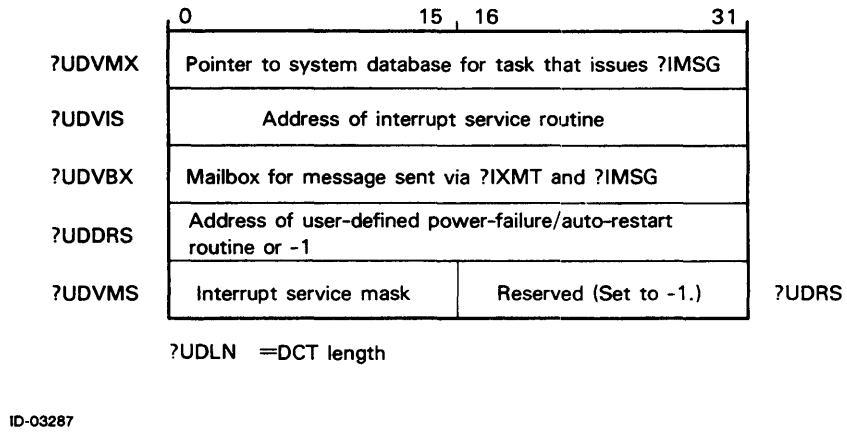


Figure 13–1. Device Control Table (DCT)

?IDEF System Call Options

When you issue the ?IDEF system call, you can select any of the following options:

- Burst multiplexor (BMC) I/O.
- Data channel (DCH) map A.
- DCH map B through P.
- Neither BMC nor 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 the DCH maps (B through P), 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 slot. Instead, you can only specify how many map slots your application needs.

However, if you do not want to use either DCH or BMC I/O, you do not have to use the extended map table; you must specify this option in the accumulators when you issue the ?IDEF system call.

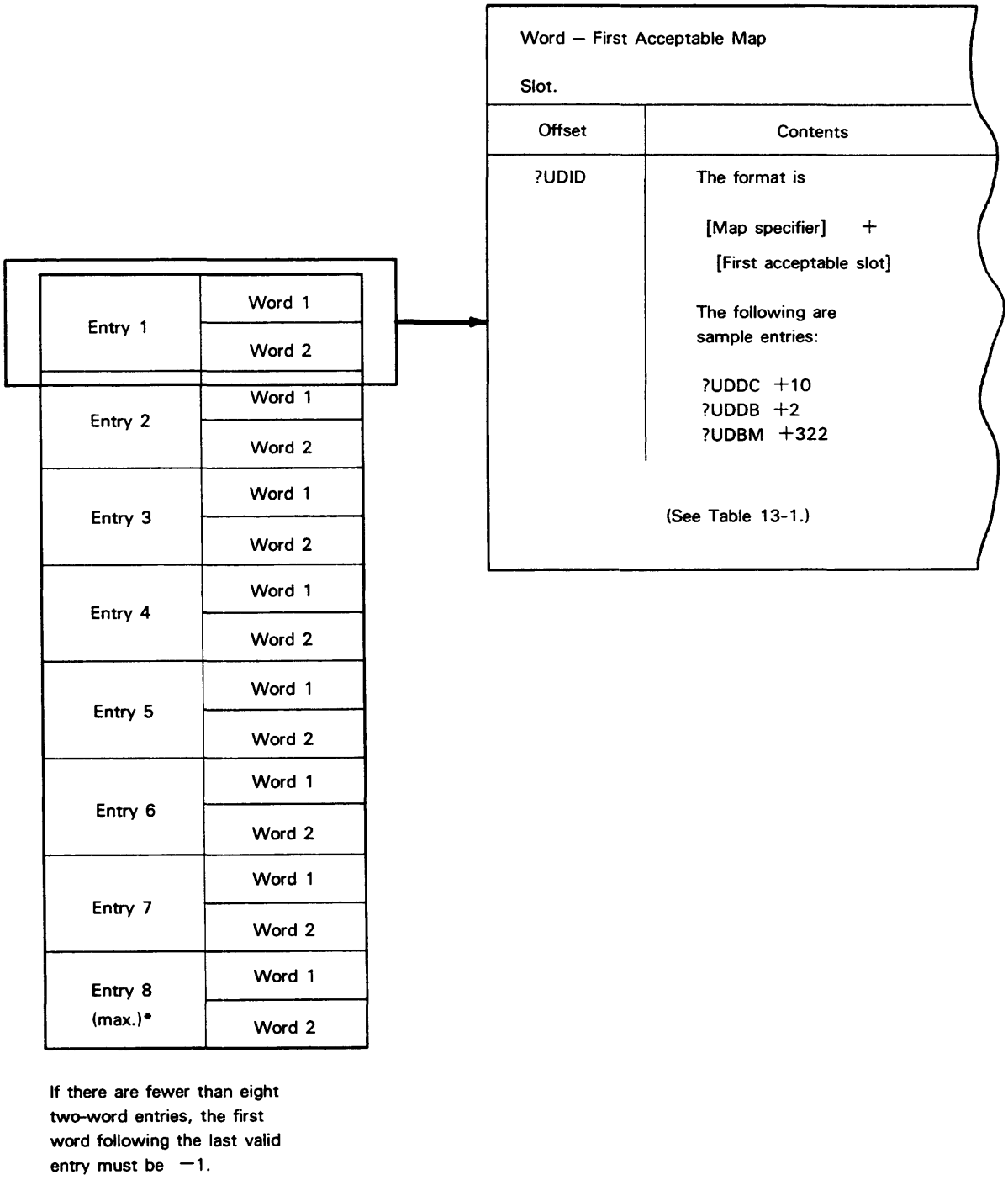
BMC I/O requires program control only at the beginning of each block transfer. Consequently, BMC I/O is generally faster than DCH I/O. Typically, the BMC rate is about half the memory rate, although the exact transfer rate varies from implementation to implementation. (Not all user-defined devices have BMC hardware).

If you use the extended form of ?IDEF, you can select specific DCH or BMC map slots. Each DCH map consists of 32 map slots, numbered 0 through 37 (octal). The BMC map consists of 1024 map slots, numbered 0 through 1777 (octal).

Each map slot (DCH and BMC) addresses 1K memory words. The hardware uses these 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. (Figure 13–2 shows the structure of a map definition table and its entries. See Table 13–1 for a detailed description of the contents of each map definition table entry.
2. Issue the ?IDEF system call.



ID-03320

Figure 13–2. Structure of Map Definition Table

Table 13-1. Contents of Map Definition Table Entry

Word 1 - First Acceptable Map Slot		
Offset	**Contents**	**Comments**
?UDID	The format is [Map specifier] + [First acceptable slot] The following are sample entries: ?UDDC+10 ?UDDB+2 ?UDBM+322	Map specifier must be one of the following: ?UDBM, which selects the BMC map. ?UDDA, which selects the DCH A map. ?UDDB, which selects the DCH B map. ?UDDC, which selects the DCH C map. ?UDDD, which selects the DCH D map. . . . ?UDDP, which selects the DCH P map. First acceptable slot must be • From 0 through 1777 (octal) if your map specifier is ?UDBM. • From 0 through 37 (octal) if your map specifier is ?UDDA, ?UDDB, ?UDDC, and so on through ?UDDP. AOS/VS allocates the first contiguous group of slots on the map, starting with the first acceptable slot on the map that you selected. Then, AOS/VS returns to you the first slot that it allocated in ?UDID.
Word 2 - Number of Map Slots Requested		
Offset	**Contents**	**Comments**
?UDNS	Number of map slots requested in range from 1 through 40 (octal)	The sum of the first acceptable slot plus the number of slots cannot be larger than the size of the map that you requested; that is, 40 (octal) for DCH or 2000 (octal) for BMC.
NOTE: If AOS/VS cannot allocate all entries, then it does not allocate any entries.		

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, a ?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, see 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 processor and AOS/VS ignore that device. As soon as the device completes its job, it signals the processor that it is done. This signal is called an *interrupt*.

When the processor 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 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 inclusively ORs it with the interrupt service mask in the DCT.
4. Saves the current load effective address (LEF) state. LEF mode is processor state that allows AOS/VS to correctly interpret LEF instructions. (See the "LEF Mode" section in this chapter for more information.)
5. Turns LEF mode off.

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 an ?IXIT system call to dismiss your interrupt.

A process in an interrupt service routine can issue only three system calls:

- ?IXMT, 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/Family floating-point registers when a process enters an 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 a 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. Consequently, you *must* define a stack for the interrupt service routine. Both ?IXMT system call, which sends an interrupt message, and the ?IXIT system call, which exits from the interrupt service routine, require a stack.

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 ?IXMT 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.

Defining and Using Devices on More than One I/O Channel

Some MV/Family computers support more than one I/O channel. When you run AOS/VS on these computers, you can define user devices that use I/O channels other than the default I/O channel (channel 0). To define a user device that uses an alternate I/O channel, you must use a three-digit device code to reference that device in the ?IDEF system call. In the three-digit code, the first digit gives the channel number; the last two digits give the code for the user device. For example, a device code of 124 references a user device that uses channel 1 and has a device code of 24.

To perform I/O to a device that runs on an alternate channel, you *must* use the PIO instruction. Unlike the NOVA I/O instructions, in which you can use only two digit device codes, the PIO instruction allows you to use three-digit device codes. For reasons that we described above, you must use a three-digit device code to reference devices on an alternate channel. For more information on the PIO instruction, see the Principles of Operation manual for your computer.

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. Table 13–2 summarizes the functions of the LEF mode and device access calls. (See the LEF Mode section in this chapter for more.)

Table 13–2. LEF Mode and Device Access System Call Functions Summary

System Call	Function
?DEBL	Enables I/O; disables LEF mode.
?DDIS	Disables I/O; but does not re-enable LEF mode.
?LEFE	Enables LEF mode; disables I/O.
?LEFD	Disables LEF mode; but does not enable I/O.
?LEFS	Returns the current LEF mode status.

No device I/O can occur while the processor is in LEF mode, because LEF instructions and I/O instructions use the same bit patterns. Similarly, when LEF mode is disabled, AOS/VS executes LEF instructions as if they were I/O instructions. *The deciding factor for executing LEF and I/O instructions is the state of the processor; that is, whether it is in LEF mode or I/O mode.*

NOTE: If a user task accesses a system device that AOS/VS is currently using, the results could be catastrophic — you should take care to ensure that user tasks *never* access system devices.

Re-enabling LEF Mode

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-effective-address mode) is the processor 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 complementary I/O mode.

LEF mode and I/O mode are mutually exclusive. When the processor is in LEF mode, all I/O instructions execute as LEF instructions; therefore, I/O cannot take place in this state. Conversely, the processor must be in LEF mode to execute LEF instructions properly.

AOS/VS provides the following system calls to check and alter LEF mode:

?LEFD Disables LEF mode.

?LEFE Enables LEF mode.

?LEFS Returns the current LEF mode status.

Each process begins with LEF mode enabled. AOS/VS disables LEF mode when a process enters a user device routine, and restores LEF mode when the process exits from that routine.

The Power-Failure/Auto-Restart Routine

If you give the address of a power-failure/auto-restart routine in the ?IDEF system call — provided you have the necessary battery backup hardware — AOS/VS will call that routine 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. (For more information on enabling auto-restart, see *How to Generate and Run AOS/VS*).

When running a power-failure/auto-restart routine, AOS/VS disables interrupts. The power-failure/auto-restart routine should *not* enable interrupts. After a power failure, the states of both the devices and the data channel map are undetermined — enabling interrupts before the power-failure/auto-restart routine has finished could produce unexpected results.

End of Chapter

Chapter 14
Supporting Binary Synchronous Communications (BSC)

You use the following system calls to support binary synchronous communications (BSC):

?SDBL	Disable a BSC line.
?SDPOL	Define a polling list or a poll address/select address pair.
?SDRT	Disable a relative terminal.
?SEBL	Enable a BSC line.
?SERT	Re-enable a relative terminal.
?SGES	Get BSC error statistics.
?SRCV	Receive data or a control sequence over a BSC line.
?SSND	Send data or a control sequence over a BSC line.

AOS/VS supports binary synchronous communications (BSC) over dedicated or switched communications lines. This chapter describes the system calls that you use to support BSC communications; it assumes that you are familiar with BSC protocol and the rules governing BSC.

To understand this chapter, you should be familiar with the following terms:

- *Station.*

  A station is the origin (sender) or destination (receiver) of data over a BSC line.

- *Dedicated communications line.*

  A dedicated communications line continuously connects two or more stations, regardless of the amount of time the line is actually in use. This type of line is dedicated to serving specific local and remote stations.

- *Switched communications line.*

  A switched communications line is one on which you use dialing procedures to establish a connection between the local and remote stations.

BSC Concepts

Before you use any of the BSC system calls, make sure that your system manager or operator has created a GSMGR (global synchronous manager) process. This process acquaints AOS/VS with the synchronous communications hardware that you specified during the system-generation dialog. If this process does not exist, any BSC system calls you issue will cause an error return. For information on creating the GSMGR process, see *How to Generate and Run AOS/VS*.

AOS/VS recognizes each BSC line by the device name, SLNx, where x represents the line number. When you enable a BSC line (with the ?SEBL system call), you must supply the SLN designator with the correct line number. However, it is not necessary to specify whether the enabled line is dedicated or switched.

AOS/VS assigns a channel number to each enabled BSC line and returns this value in the ?SEBL packet. Unlike disk files, you cannot open a BSC line on more than one channel.

To send data over an enabled BSC line, a station issues the ?SSND system call. To receive data, a station issues the ?SRCV system call. BSC protocol distinguishes between Send Initial and Send Continue system calls, and between Receive Initial and Receive Continue system calls. A system call is an Initial system call if it opens a communications session.

The ?SSND and ?SRCV system calls depend upon timing and upon the interaction of the sending and receiving stations. When AOS/VS encounters timing errors or inappropriate responses to the ?SSND and ?SRCV system calls, it begins error-recovery procedures. You must view these error-recovery procedures in the context of the send and receive system calls. (See the "BSC Error-Recovery Procedures" section in this chapter.)

To disable a BSC line, issue the ?SDBL system call.

BSC Line Configurations

There are two types of BSC line configurations:

- *Point-to-point.*

  On a point-to-point line, each station bids for the line; that is, asks to use it. Only two stations can be on a point-to-point line.

- *Multipoint* (also called *Multidrop).*

  On a multipoint line, stations do not bid for the line. Instead, one station (called the control station) has complete control over the activities of the other stations (called the tributary stations) on the line. Therefore, no contention occurs between stations. Usually, a multipoint line connects one local station with more than one remote station. However, it can connect as few as two stations.

If both stations on a point-to-point line bid for the line at the same time, the line is under contention. Contention occurs when one point-to-point station bids for a line and the other station, in response, also bids for the line. When you enable a point-to-point line, you must designate your computer as either the primary station or the secondary station. AOS/VS favors the primary station over the secondary station in the following way when contention occurs:

- If your station is the primary station, AOS/VS automatically follows your bid with another bid sequence. The secondary station should acknowledge this additional bid sequence.
- If your station is the secondary station, AOS/VS gives you an error return. To receive the primary station's bid sequence, you must issue an ?SRCV Receive Initial system call.

Unlike the secondary stations on a point-to-point line, the tributary stations on a multipoint line are completely subservient to the control station. The following restrictions apply to tributary stations:

- Tributary stations can only send data to and receive data from the control station.
- A particular tributary station can send or receive data over the line only when the control station specifically requests that it do so.
- Tributary stations cannot communicate directly with one another.

When you enable a BSC line with the ?SEBL system call, you must specify whether it is a point-to-point line or a multipoint line. Also, you must use the ?SEBL system call to specify whether your station is a primary station, a secondary station, a control station, or a tributary station.

Figure 14–1 shows the difference between a point-to-point line configuration and a multipoint line configuration.

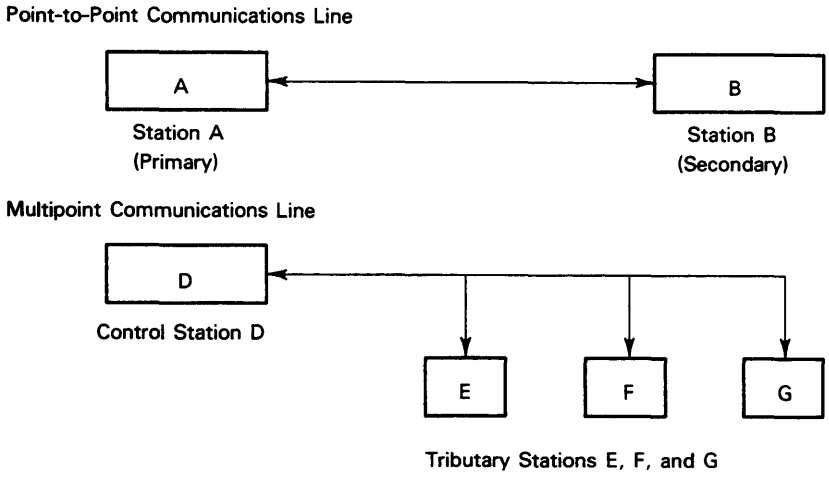


Figure 14–1. Point-to-Point/Multipoint Line Configurations

Multipoint Line Selection and Polling

To manage the activity on a multipoint line, the control station performs two operations:

- *Polls.*

  This means that the control station contacts its tributary stations to see if any of them has data to send to it. There are two types of polls: *general* and *specific.*

  In a general poll, the control station contacts each of its tributaries in round-robin fashion, and accepts the first positive response.

  In a specific poll, the control station contacts a single tributary to solicit data.

- *Selects.*

  The control station selects by contacting a specific tributary to see if it is ready to receive data from the control station.

Each tributary station on a multipoint line must have two unique identifiers for polling and selecting to occur: a poll address and a select address. If your computer is a tributary station, you must define its poll address and select address by issuing the ?SDPOL system call.

If your computer is a control station, you must issue the ?SDPOL system call before polling or selecting to define a polling list. A polling list is a series of contiguous words that contains each tributary station's poll address and device address. The device address points to the peripheral device from which the control station will request data when it polls that tributary.

To perform polling, a control station issues the ?SRCV system call to specify whether the system call is a Receive Initial or a Receive Continue system call and whether the operation is general polling or specific polling.

In its first general poll, the control station starts with the poll and device address entry at the top of the polling list (the lowest relative terminal number), and sends this entry down the BSC line. Each tributary station recognizes its own poll address; it responds to the poll only if the entry matches its poll address. If the poll address sent by the control station does not match a tributary station's poll address, the tributary station ignores it.

A general poll ends when a tributary station responds to its poll address by sending data to the control station. If there is no response to a particular poll address entry, the control station continues to poll until it reaches the end of the polling list. At that point, AOS/VS takes the error return from the control station's ?SRCV system call, and passes error code EREPL (end of polling list) to AC0.

As we mentioned, general polling is a round-robin operation. This means that when a general poll ends in a positive response, the next general poll begins with the next relative terminal on the polling list (that is, the tributary station immediately following the previous respondent). Specific polling, that is, polling one and only one tributary station, is a way to break out of the round-robin method of general polling.

Relative Terminals

AOS/VS assigns a relative terminal number to each tributary station, based on that station's position on the polling list.

The first time you enable a multipoint line (with the ?SEBL system call) and define its polling list (with the ?SDPOL system call), AOS/VS enables all relative terminals on the list for polling. To disable a relative terminal without redefining the polling list, issue the ?SDRT system call. To re–enable a relative terminal, issue the ?SERT system call.

When you disable a relative terminal, it does not affect the corresponding tributary station; it simply means that the control station ignores that tributary station when it performs general polling, until you subsequently re-enable the relative terminal or define a new polling list.

BSC Protocol

The logic behind data transmissions over a BSC line is BSC protocol. Briefly, BSC protocol is a set of rules governing

- The initialization of communications over a BSC line.
- The orderly exchange of data over a BSC line.
- The termination of communications over a BSC line.

These objectives are accomplished in part by the protocol's data–link control characters, which are synchronization characters that both the sending and the receiving stations recognize. Data transmissions over a BSC line typically consist of text, header information (optional), and data-link control characters, which delimit various portions of the data block and control its transmission.

None of the BSC system calls require data-link control characters as input. AOS/VS provides the required control characters when you send text or header information over a BSC line, and removes them when you receive the information. However, because several of the system call descriptions refer to the data-link control characters, Table 14–1 defines the control characters that chapter.

Table 14–1. BSC Protocol Data-Link Control Characters (DLCC)

Character	Description
ACK0 ACK1	Affirmative Acknowledgment Positive replies, sent in alternating sequence, to indicate that the receiver has accepted the previous block without error, and is ready to accept the next block of the transmission. ACK0 is also a positive response to a line bid (ENQ) for a point-to-point line and to a selection sequence on a multipoint line.
BCC	Block Check Character A value generated by the transmitting station and sent with each data block to validate the block's contents. The receiving station generates its own BCC. If the two values match, the block is accepted as error-free. A BCC follows every ITB, ETB, and ETX character. If you transmit in the ASCII code set, the BCC is a longitudinal redundancy check (LRC). For the EBCDIC code set, the BCC is a cyclical redundancy check (CRC).
DLE	Data-Link Escape The first character in a two-character sequence used to signal the beginning or end of Transparent Text mode. The sequence DLE STX signals the beginning of Transparent Text mode. The sequence DLE ETB or DLE ETX signals the end of Transparent Text mode. For more information on Transparent Text mode, see the the description following this table.
DLE EOT	Data-Link Escape, End-of-Transmission Signals a line disconnect for a switched line. The sending or receiving station usually transmits this sequence when all message exchanges are finished.
ENQ	Enquiry Sent by a station on a point-to-point line to bid for the line (for transmission of data). Sent by the control station on a multipoint line to signal the end of a polling or selecting sequence. A transmitting station can also send ENQ to ask the receiver to repeat a response if the original response was garbled or not received when expected.
EOT	End-of-Transmission Signals the end of a message transmission (consisting of one or more separately transmitted data blocks), and resets the receiving station. On a multipoint line, a polled station sends an EOT to indicate that it has nothing to send back to the control station. EOT can also serve as an abort signal to indicate a system or transmission malfunction.
ETB	End-of-Transmission Block

(continues)

Table 14-1. BSC Protocol Data-Link Control Characters (DLCC)

Character	Description
ETX	End-of-Text Signal the end of a data block that began with an SOH or an STX. Both the ETB and the ETX characters reverse the direction of the transmission. When a station receives an ETB or an ETX, it replies with a control character that indicates its status (that is, ACK0, ACK1, NAK, WACK, or RVI). An ETB terminates every text block except the last. An ETX implies an end-of-file condition; thus, it terminates the last block of text in a message.
ITB	End-of-Intermediate-Transmission Block Separates records within a block and/or delimits field boundaries for error checking. ITB does not reverse the direction of the transmission.
NAK	Negative Acknowledgment Sent by the receiving station to indicate that it is not ready to receive, or to request that an erroneous block be transmitted again.
RVI	Reverse Interrupt A positive response used instead of ACK0 or ACK1; signals that the receiver must interrupt the transmission to send the transmitting station a high-priority message. The transmitting station treats an RVI as a positive acknowledgment and, in response, transmits all the data that prevents it from becoming a receiving station. The transmitting station can perform more than one block transmission to empty all its buffers. On a multipoint line, a control station can send an RVI after it receives data from a tributary station, to indicate that it wants to communicate with a different tributary station.
SOH	Start of Header Signals the start of header information (ancillary information within a block).
STX	Start of Text Signals the beginning of the text (and terminates the header information).
SYN	Synchronization Character Establishes and maintains character synchronization; also serves as filler in the absence of data or control characters. Each transmission must begin with at least two contiguous SYN characters.
TTD	Temporary Text Delay A two-character sequence that consists of STX ENQ, which the transmitting station sends to retain the line without immediately sending data. The receiving station responds with a NAK. The TTD/NAK sequence can repeat, if the transmitter needs additional delays.
WACK	Wait-Before-Transmit Positive Acknowledgment A positive acknowledgment that the receiver sends; signals that the receiver is temporarily unable to receive. (A receiver can send WACK as a response to a line bid on a point-to-point line or a selection sequence on a multipoint line, or as a response to data.) A receiving station can send more than one WACK until it is ready to receive. The transmitting station can respond with ENQ, EOT, or DLE EOT.

(concluded)

Note that BSC protocol supports Transparent Text mode. Transparent Text mode causes AOS/VS to treat most control characters as bit patterns without control significance. The exceptions are DLE STX, which signals the beginning of Transparent Text mode, and DLE ETB or DLE ETX, which signal the end of Transparent Text mode. If you are sending data that may match the bit patterns of the control characters, you should send it in Transparent Text mode.

BSC Error-Recovery Procedures

When AOS/VS receives an inappropriate response to an ?SSND or ?SRCV system call, or does not receive a response within the time-out interval that you specify in offset ?STOV, it enters its BSC error-recovery procedures. The error-recovery procedures differ, depending on which operation was underway when the error occurred. In addition, AOS/VS's action within each recovery procedure depends on the cause of the error.

In most cases, AOS/VS responds to a BSC error by trying the particular procedure again, repeatedly if necessary, until its retry count is exhausted. The retry count is a system-maintained variable, which you cannot control.

Table 14–2 describes the error-recovery procedures for the various types of send and receive system calls.

Table 14-2. BSC Error-Recovery Procedures

Call Type	AOS/VS Action
Send Initial	
Time-out NAK or inappropriate response	Resend ENQ, unless retry count exceeded. If retry count exceeded, take error return to ?SSND system call. (Possible errors in AC0 are ERTOF, ERNAK, ERUNI.)
ENQ	If calling station is the primary, resend ENQ. If calling station is the secondary, take error return to ?SSND system call. (Error in AC0 is ERCTN.)
Send Continue	
Time-out or inappripriate response	Send ENQ, unless retry count exceeded. If retry count exceeded, take error return to ?SSND system call. (Possible errors in AC0 are ERTOF, ERUNI.)
NAK	Resend data, unless retry count exceeded. If retry count exceeded, take error return to ?SSND system call. (Error in AC0 is ERNAK.)
Receive Initial (point-to-point and multipoint tributary station)	
Time-out or inappropriate response	Retry receive initial, unless retry count exceeded. Take error return to ?SRCV system call. (Possible errors in AC0 are ERTOF, ERUNI.)

(continues)

Table 14-2. VSC Error-Recovery Procedures

Call Type	AOS/VS Action
Receive Continue	
Time-out or inappropriate response	If retry count exceeded, take error return to ?SRCV (possible errors in AC0 are ERTOF, ERUNI). Otherwise, await ENQ from sender (assuming that the sender will issue a time-out and send an ENQ).
ENQ	Resend last response and attempt receive continue, unless retry count exceeded. If retry count exceeded, take error return to ?SRCV system call. (Error in AC0 is ERENQ.)
CRC (block check) error	Send negative acknowledgement (NAK) and error attempt receive continue unless retry count exceeded. If retry count exceeded, take error return to ?SRCV system call. (Error in AC0 is ERCRC.)
Receive Initial (multipoint control station)	
Time-out or inappropriate response	Retry receive initial, unless retry count for particular relative terminal exceeded. If retry count exceeded, take error return to ?SRCV system call (Possible errors in AC0 are ERTOF, ERUNI.)
EOT	If a polled terminal responds with EOT, step to the next relative terminal on the polling list and continue polling. If end of the polling list is reached, take error return to ?SRCV system call. (Error in AC0 is EREPL.)

(concluded)

To get BSC error-recovery statistics, issue the ?SGES system call. The ?SGES system call returns the number of block-check errors, the number of time-outs, and the total number of negative acknowledgment (NAK) characters received in response to send operations.

BSC Implementation

Figures 14–2 through 14–8 illustrate how AOS/VS implements BSC protocol using the BSC system calls. Before you read this section, refer to the system call descriptions for the ?SEBL, ?SSND, and ?SRCV system calls in the *System Call Dictionary (AOS/VS and AOS/DVS)*, and to the definitions of the BSC data-link control characters in Table 14–1.

Remember, you cannot issue a send initial system call from a multipoint tributary station, and that Receive Continue system calls from multipoint stations and from point-to-point stations are identical.

You will notice that each figure has three columns. The first column represents the system calls that you issue, with their normal and error returns. The second column illustrates AOS/VS's actions, and the third column shows the remote station's actions.

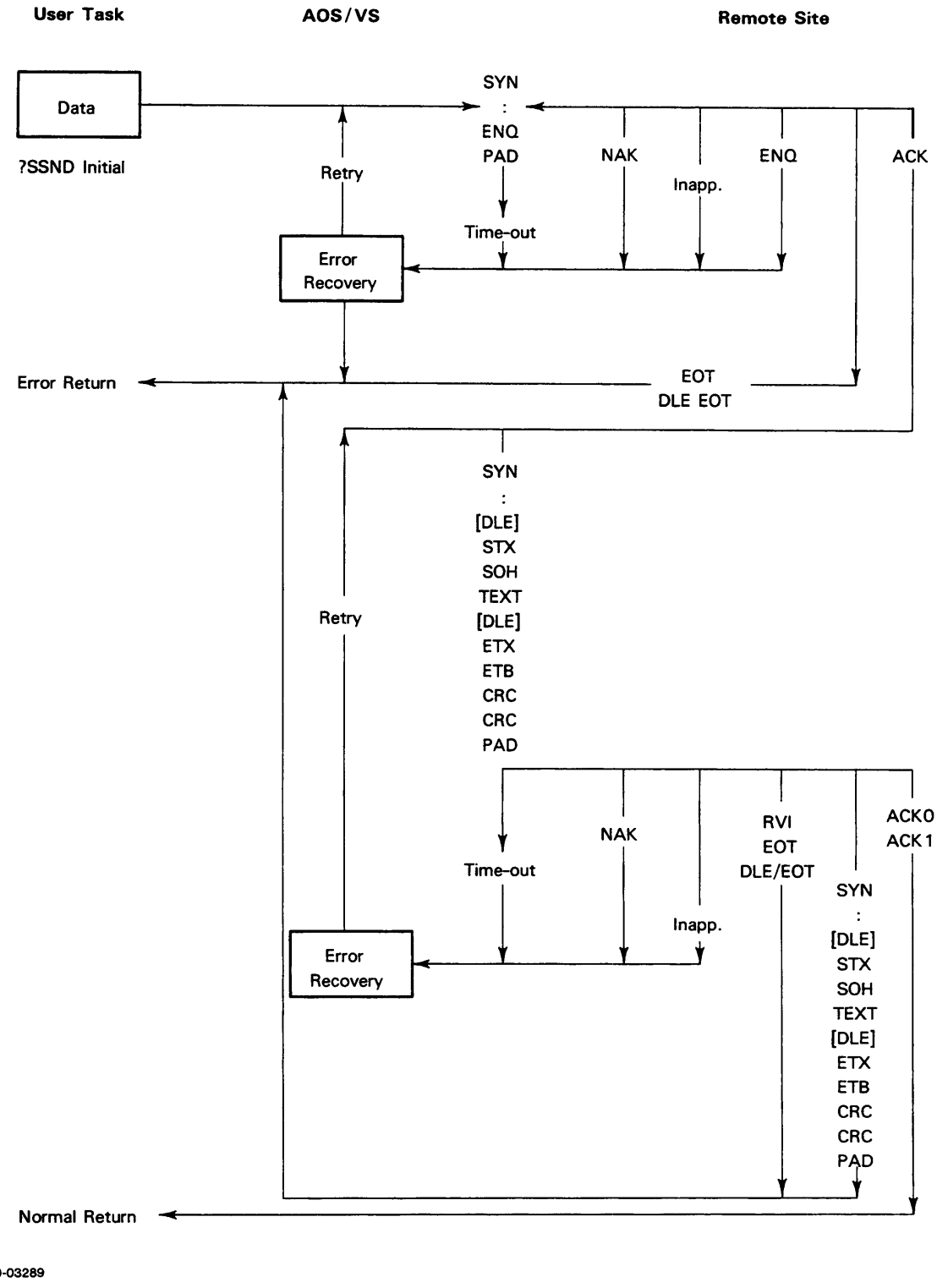
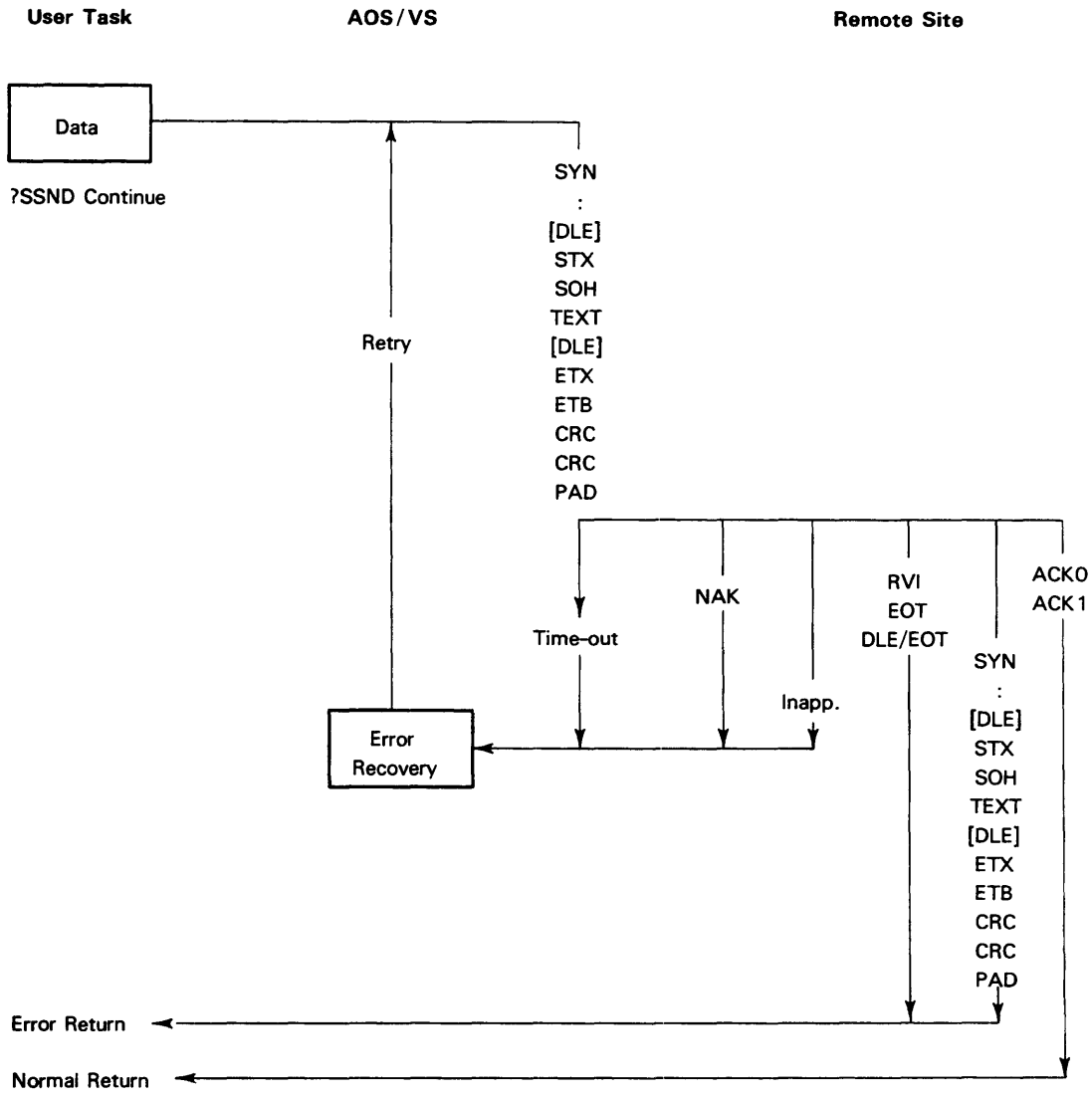


Figure 14–2. ?SSND System Call, Initial, Point-to-Point



ID-03290

Figure 14–3. ?SSND System Call, Continue, Point-to-Point

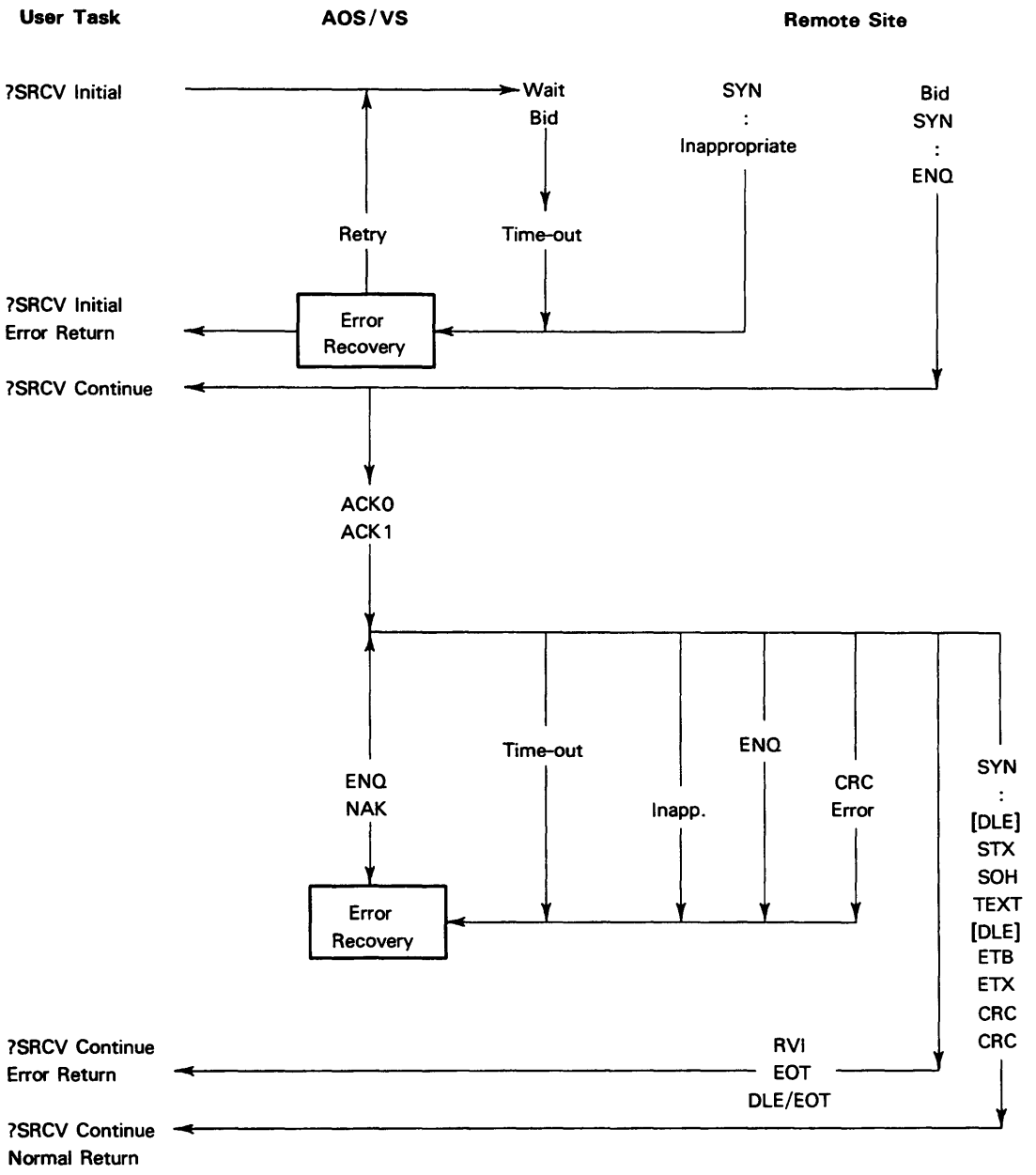


Figure 14–4. ?SRCV System Call, Initial and Continue, Point-to-Point

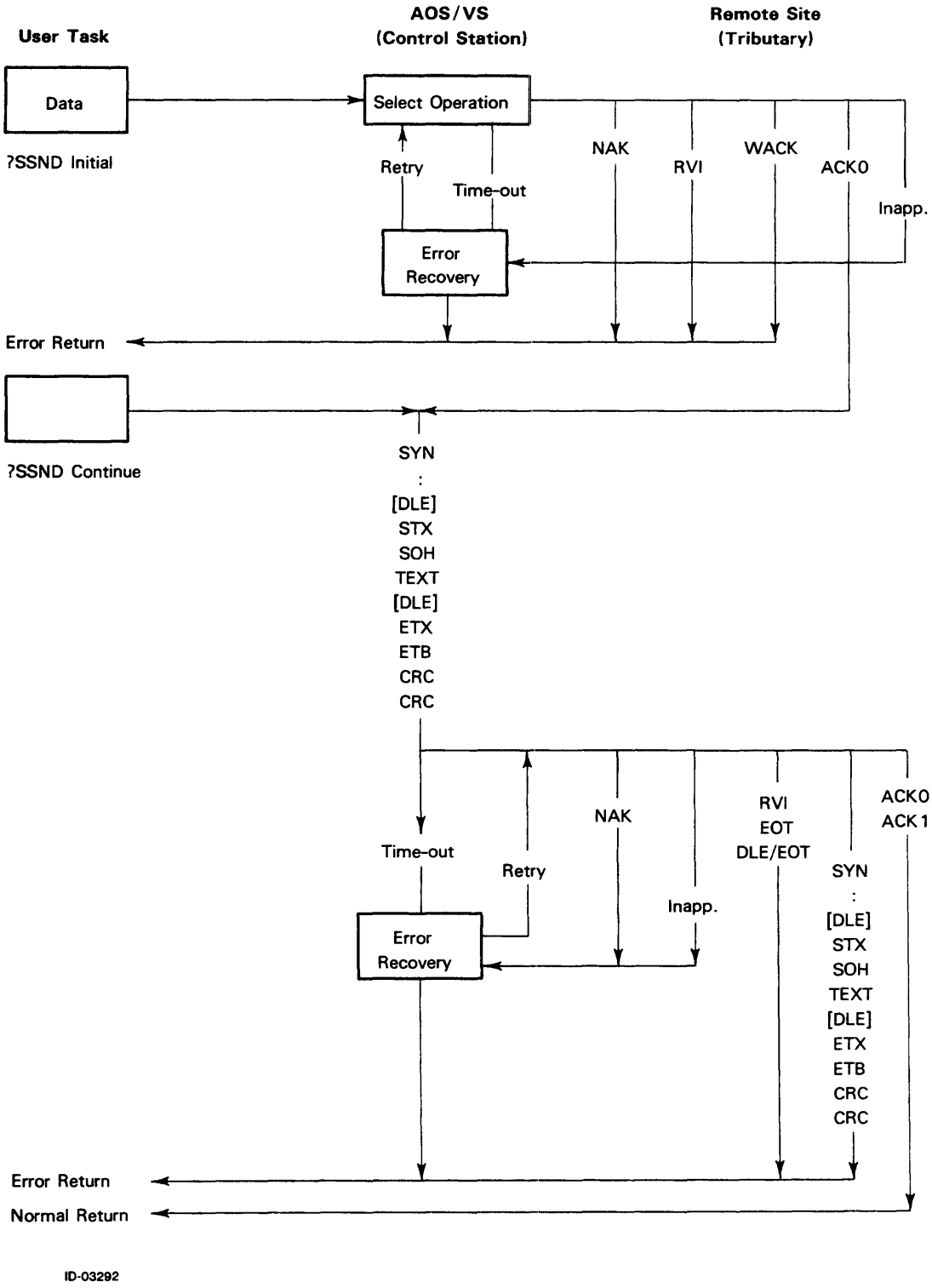


Figure 14–5. ?SSND System Call, Multipoint Control Station

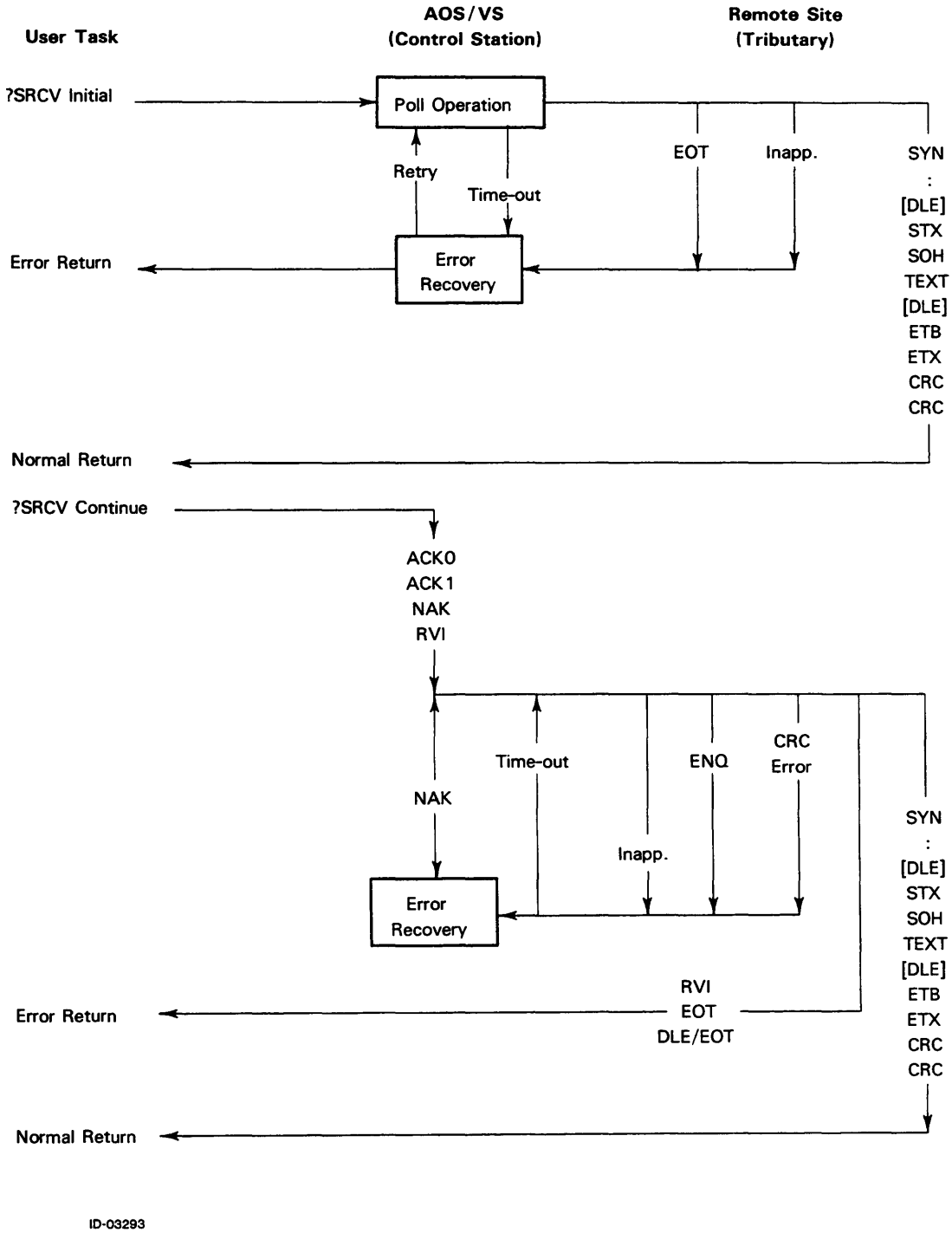
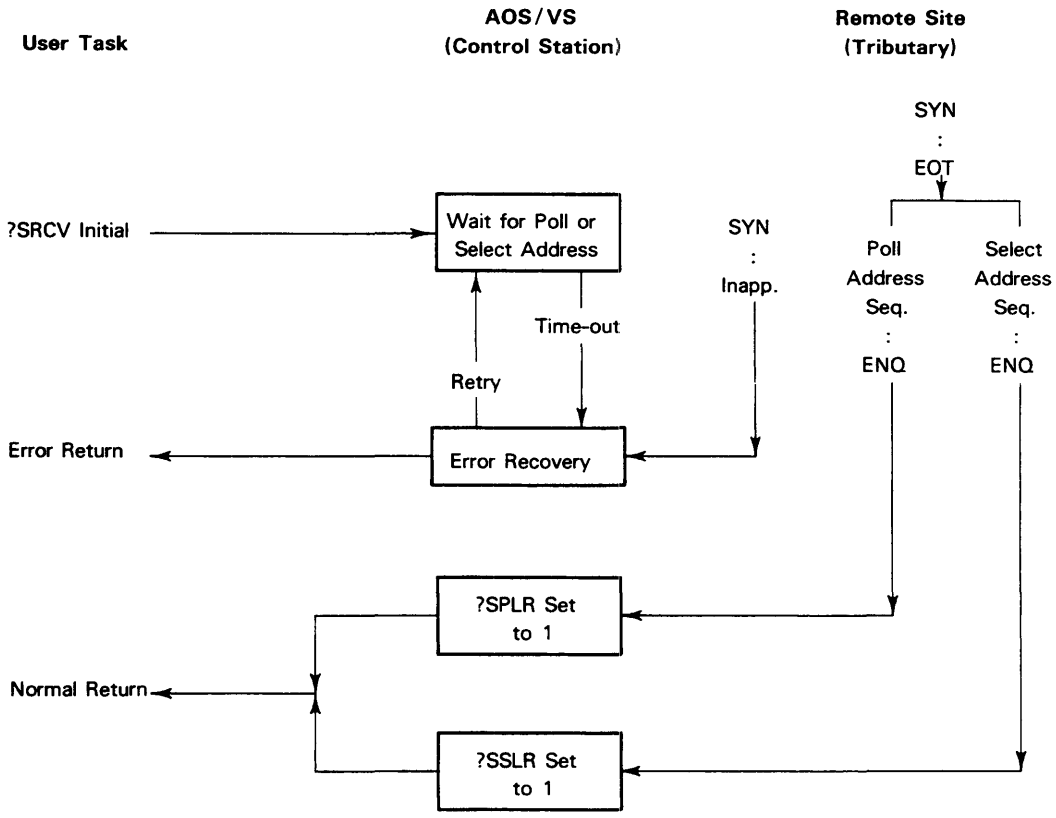


Figure 14–6. ?SRCV System Call, Multipoint Control Station



NOTE: Use ?SRCV continue if the station was selected; use ?SSND if the station was polled.

ID-03294

Figure 14–7. ?SRCV System Call, Multipoint Tributary Station

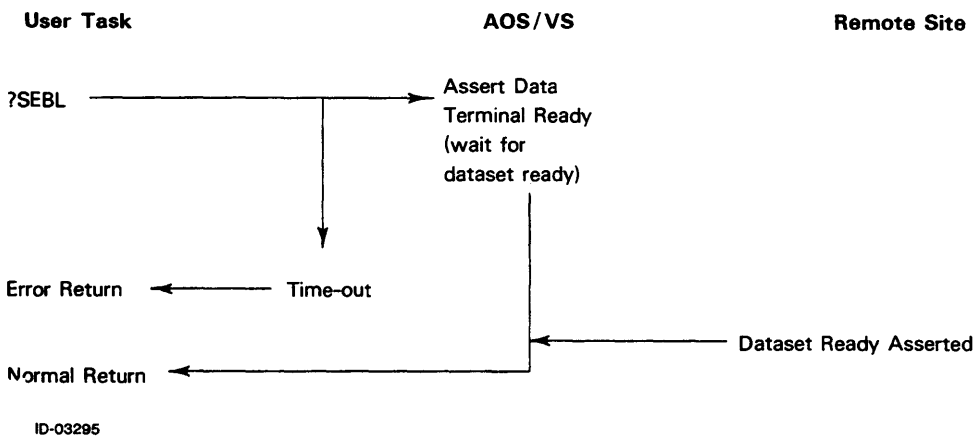


Figure 14–8. ?SEBL System Call, Point-to-Point

End of Chapter

Chapter 15
Managing 16-Bit Processes

You use the following system calls to manage 16–bit processes:

?DELAY	Suspend a 16–bit task for a specified interval.
?GCRB	Get the base of the current resource.
?IDSTAT	Return task status word.
?IESS	Initialize an extended state save (ESS) area.
?IHIST	Start a histogram for a 16-bit process.
?KCALL	Keep the calling resource and acquire a new resource.
?OVEX	Release an overlay and return.
?OVKIL	Exit from n overlay and terminate the calling task.
?OVLOD	Load and go to an overlay.
?OVREL	Release an overlay area.
?RCALL	Release one resource and acquire a new one.
?RCHAIN	Chain to a new procedure.
?SERMSG	Return text for associated error code.
?UNWIND	Unwind the stack and restore the previous environment.
?WALKBACK	Return information about previous frames in the stack.

This chapter describes how to manage 16–bit processes under AOS/VS. AOS/VS allows you to execute 16–bit programs in addition to 32–bit programs. These can be programs you developed under AOS/VS or under the Advanced Operating System (AOS); in the latter case, you must relink to execute the programs under AOS/VS. In some cases, reassembling or recompiling AOS programs may also be necessary.

Memory Modification with Disk Images

Sixteen-bit programs have a more restricted address space than 32–bit programs (32K words or less). Therefore, to augment a 16–bit process's logical address space, you must call in shared or unshared overlays. There are two types of system calls for this purpose: resource system calls, which automatically load and release overlay procedures, and primitive overlay system calls.

Most 16–bit applications use the resource system calls, because they simplify resource management. These system calls let you postpone the decision to include the resources in your root program or in one or more overlay areas until link time.

The primitive overlay system calls give you greater control of overlays, but to use them, you must be willing to explicitly load, release, and control overlays.

Using Overlays

Overlays are useful in a small logical address space because they allow you to reuse the same portion of memory, called an overlay area, for different portions of code. In general, at link time, you define two or more overlays for each overlay area. The Link utility classifies the elements of a 16–bit program into two resource types:

- The root, which is the memory-resident portion of the program.
- Overlays.

Link reserves space in the .PR file for overlays, but diverts the actual overlay code to an overlay (.OL) file. As your program calls overlays during execution, AOS/VS reads them from the overlay areas in the .OL file to the designated overlay areas in memory.

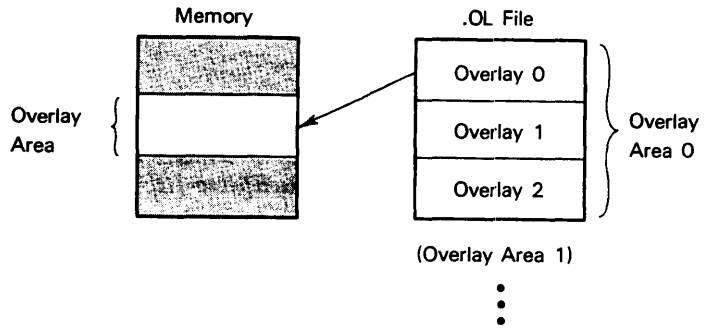
You can define as many as 63 overlay areas per program. Each overlay area can accommodate a maximum of 511 separate overlays. An overlay area can consist of either shared or unshared overlays, but not both. Link builds shared overlay areas in multiples of 1K words and builds unshared overlays in multiples of 256 words.

Normally, the basic size of an overlay area equals the size of its largest overlay, plus any padding Link provides to fill out the area to a multiple of 1K or 256 words. As a result, AOS/VS reads only one overlay into the overlay area at that time. (See Figure 15–1.)

You can increase an overlay area to a multiple of its basic size at link time. This results in a total overlay area that can simultaneously accommodate more than one overlay of the basic size. During execution, AOS/VS can place these overlays into any of the basic areas within the total overlay area.

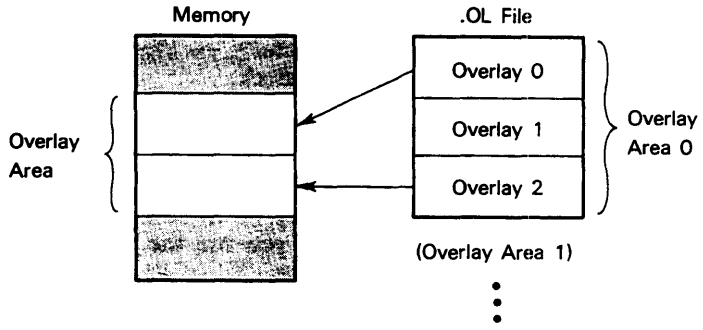
Therefore, overlays destined for a multiple-overlay area must be position-independent; that is, you must write them so that all internal procedure references are relative to some point in the same overlay. Figure 15–2 shows an overlay area with a total size that is double its basic size.

As Figure 15–2 shows, doubling the basic size of the overlay area in memory allows AOS/VS to simultaneously read in two overlays of the basic size (Overlay 0 and Overlay



ID-03296

Figure 15–1. Basic Overlay Area Equals Size of Largest Overlay



ID-03297

*Figure 15–2. Multiple Overlay Area (Total Area = Basic Size * 2)*

2). Note that AOS/VS could also fit both Overlay 1 (which is smaller than the basic size) and Overlay 2, or both Overlay 0 and Overlay 1 in the total overlay area.

Usually, you use special overlay designators to define object modules as overlays in the Link command line. Link assigns a number to each overlay area and to each of the overlays that make up that overlay area. Link bases these numbers on the order in which the overlay areas and the individual overlays appear in the Link command line.

For example, to link six object modules, A, B, C, D, E, and F, to form the program file A.PR, the command line is

```
X LINK A B !*C DIE F*!
```

The overlay designators (!*, !, and *!) define one overlay area (Overlay Area 0) with two distinct overlays: modules C and D make up Overlay 0, while modules E and F make up Overlay 1.

You can use the pseudo-ops .ENTO and .EXTN to refer symbolically to overlay areas and overlays. (For more information on .ENTO and .EXTN, refer to the *AOS/VS Macroassembler Reference Manual.* Also, for more information on how Link handles overlays, refer to the *AOS/VS Link and Library File Editor (LFE) User's Manual.*)

Resource System Calls

AOS/VS provides access to overlays and other procedures through a generalized procedure/system call mechanism implemented by the ?RCALL, ?KCALL, and ?RCHAIN system calls. You must define the procedure you want to call with the .PENT (procedure entry) pseudo–op. For the ?RCALL and ?KCALL system calls, the calling procedure must begin with an ?RSAVE macro instruction and end with RTN. (The ?RCHAIN caller must begin with ?RSAVE. Only the last procedure in the chain should end with a RTN.)

The ?RCALL system call releases the calling resource, and then loads the new resource. Because the calling resource is released on an ?RCALL, you can load the new resource into the caller's memory area.

AOS/VS preserves the state of the ?RCALL caller so that it can reload the caller, if necessary, after it executes the new procedure.

The ?KCALL system call loads a new resource and transfers control to its entry point. Unlike the ?RCALL system call, the ?KCALL system call does not release the calling resource. You must use the ?KCALL system call carefully — the indiscriminate use of the ?KCALL system call can result in a resource deadlock.

A resource deadlock occurs when every task that requires overlays is suspended waiting for overlay areas to become available. (If the overlays have issued ?KCALL system calls, AOS/VS cannot release their overlay areas.) A resource deadlock can occur if an overlay issues a ?KCALL system call that loads another overlay to the same basic (non-multiple) overlay area. To load procedures, we recommend that you use the ?RCALL system call instead of the ?KCALL system call to load procedures.

The ?RCHAIN system call releases the calling resource and acquires the new resource before it leaves the calling procedure. Typically, you use the ?RCHAIN system call to join resources that you split into small sequential pieces. Only procedures within resources that you have loaded with the ?RCALL or ?KCALL system call can issue the ?RCHAIN system call.

The ?RCHAIN system call allows you to chain from an a procedure that you have loaded with an ?RCALL or ?KCALL system call to a new procedure. After AOS/VS executes the new procedure, it returns control to the original procedure, not to the ?RCHAIN caller.

Link resolves each resource system call and, if necessary, binds the appropriate resource handler routines into the program file. These routines, which are part of the runtime library URT16.LB, load the called procedures as they are needed at execution time.

If a resource deadlock or error occurs while AOS/VS is executing a resource system call, it transfers control to an error-processing module with the entry ?BOMB. Unless you write

your own error-handling routine with a ?BOMB entry, AOS/VS uses the default routine in URT16.LB. The default routine terminates the calling process and passes the appropriate error code to the caller's father.

If you use your own ?BOMB routine, AOS/VS transfers control to that routine, and supplies the following error-handling information:

- An error code in AC0.
- The procedure descriptor entry in AC1.
- The fault address on the stack.

Procedure Entries

Usually, you pass procedure entries as arguments to the resource system calls; for example, ?RCALL procedure entry. As an alternative, you can pass procedure entry descriptors on the stack. AOS/VS then pops the procedure entry descriptor off the stack before you execute the resource system call.

The .PTARG pseudo-op translates the name of a procedure to a procedure entry descriptor. Figure 15–3 shows a sample ?RCALL sequence that uses the descriptor method.

```
NAME:   .PTARG  FIRST           ;Define a procedure entry
              .                 ;descriptor for FIRST.
              .
              .
        LDA     0,NAME          ;Load AC0 with the descriptor.
        PSH     0,0             ;Push the descriptor onto the
                                ;stack.
        LDA     0,ARG1          ;Pass ARG1,
        LDA     1,ARG2          ;ARG2, and
        LDA     2,ARG3          ;ARG3 to FIRST.
        ?RCALL                  ;Pop procedure entry descriptor
                                ;off the stack, release calling
                                ;resource, and acquire resource
                                ;that contains FIRST.
```

Figure 15–3. Passing a Procedure Entry Descriptor via the Stack

Alternate Return from Resources

After AOS/VS executes a new procedure, it returns control to the word that immediately follows the resource system call. To return control to the second word after the resource system call, issue the following instruction sequence from the calling procedure:

```
ISZ ?ORTN,3
RTN
```

The first statement (ISZ ?ORTN,3) increments the caller's return address. The RTN instruction returns to the incremented address.

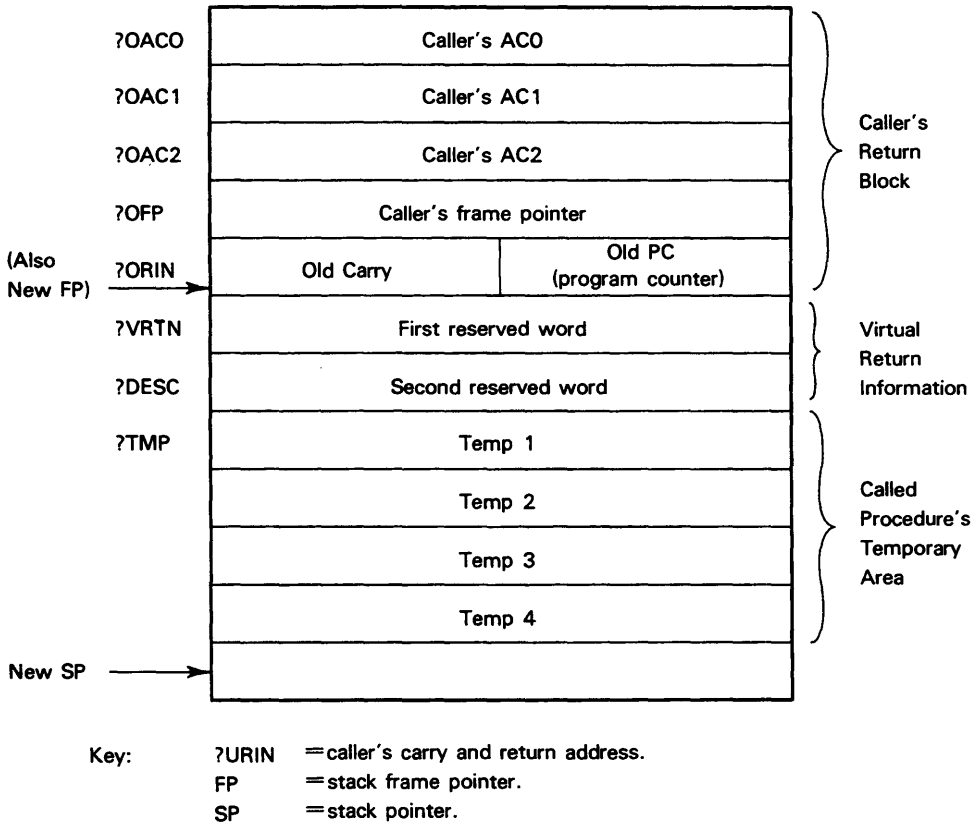


Figure 15–4. Resource System Call Stack after ?RSAVE System Call

System Management of Resource System Calls

The ?RCALL, ?KCALL, and ?RCHAIN system calls require two extra words on the user stack. These words must be located between the caller's return block and the called procedure's temporary variables. Do not alter these words, because AOS/VS uses them to store information from the called procedure.

The ?RSAVE macro instruction reserves these two extra words. Therefore, all procedures that issue ?RCALL or ?KCALL system calls must begin with an ?RSAVE instruction and end with a RTN instruction. Every procedure that the ?RCHAIN system call acquired must also begin with an ?RSAVE instruction, but only the last procedure in the chain should end with RTN.

Figure 15–4 shows the contents of the stack after the execution of an ?RSAVE instruction. Figure 15–4 also lists the parametric names of the stack locations. Note that these parameters

apply to the stack for 16–bit programs, not the stack for 32–bit programs. The user parameter file, PARU16, defines these and the other 16–bit parameters. For more information on stacks, refer to the appropriate "Principles of Operation" manual for your Data General computer.

Runtime Relocatability Requirements

If they are part of multiple overlay areas, the overlays you call with the ?RCALL, ?KCALL, and ?RCHAIN system calls must be position-independent, because AOS/VS may reload them into different portions of the overlay area after it executes the resource system call.

Moreover, overlays within multiple areas must be runtime relocatable to issue ?RCALL system calls. This means that you cannot issue any assembly language reference to a fixed address before the ?RCALL system call, because the address could be invalid when the calling overlay is reloaded. The ECLIPSE instructions subject to this restriction are JSR, EJSR, LEF, ELEF, PSHJ, POPJ, XOP, and PSHR. Figure 15-5 shows a JSR instruction whose return value will be invalid if the procedure that issues the ?RCALL system call is reloaded into a different memory area.

```
                .
                .
                .

C:      JSR A           ;Jump to Subroutine A.
                .       ;Return address is C+1.
                .
                .
A:      ?RSAVE          ;Save the return address of C.
                .
                .
                .
        ?RCALL B        Release C, acquire B, and go to
                .       ;target procedure in B.
                .
                .
        RTN             ;Return to C+1.
```

Figure 15–5. Invalid Return Address from ?RCALL System Call

The return address from the ?RCALL system call in Figure 15–5 is C+1, which is the first word that follows the JSR A instruction. The return address will be invalid, however, if AOS/VS relocated procedure A after it executed procedure B. If procedure A issued a ?KCALL system call to B instead, the return would be valid, because the ?KCALL system call keeps the calling resource (A) before it acquires the new resource and transfers control to its correct entry point (B).

Primitive Overlay System Calls

As an alternative to the resource system calls, you can use the primitive overlay system calls, ?OVLOD, ?OVREL, ?OVEX, and ?OVKIL, to call and release overlays. These system calls give you greater control over the overlay environment, but require you to explicitly load and release the overlays. Because the resource system calls manage overlays automatically, you should use them rather than using the primitive overlay system calls.

To use the primitive overlay system calls, you must define each overlay with the .ENTO (overlay entry) pseudo–op. The system maintains an overlay use count (OUC) for every memory-resident overlay. The OUC specifies the number of tasks currently using the overlay. When the OUC value reaches 0, the overlay area is freed for use by another overlay.

As long as any task is using an overlay (that is, OUC is not 0), no other overlay can be loaded into the same basic overlay area. This is true even if another high-priority task issues an overlay load request in the meantime. If the overlay area is a multiple of its basic size, however, another task can use any free basic area in the total area.

The ?OVLOD system call loads an overlay and passes control either to the beginning of that overlay or to some offset within it. In addition, your input to the ?OVLOD system call determines whether the loading is conditional or unconditional.

If you specify unconditional loading, AOS/VS loads the overlay that you request, even if it is already resident. If you specify conditional loading, AOS/VS first checks whether the target overlay is already in the overlay area. If it is, AOS/VS does not load the overlay, but simply increments the overlay's OUC. If the overlay is not resident, AOS/VS loads it into the overlay area and sets its OUC to 1.

To release an overlay that was loaded with the ?OVLOD system call, you must use one of the following release system calls:

- ?OVREL

  The ?OVREL system call decrements the overlay's OUC and frees the overlay area if the OUC equals 0. Note that you cannot issue ?OVREL from the overlay you want to release. Instead, issue ?OVREL from some point outside that overlay.

- ?OVEX

  The ?OVEX system call decrements the overlay's OUC, frees the overlay area if the OUC equals 0 and transfers control to a specific nonoverlay address. You can use the ?OVEX system call to return from a subroutine within an overlay.

- ?OVKIL

  The ?OVKIL system call decrements the overlay's OUC, releases the overlay area if the OUC equals 0 and kills the calling task.

Extended State Save Area

AOS/VS allows each 16-bit process to set up an extended state save area (ESS) for each task in the unshared portion of the logical address space. The ESS area holds task-specific information, such as the value of the program counter and its carry bit, and the current contents of the accumulators. However, you can use the ESS area to store any information you feel is relevant to a task.

Before you can use an ESS area, you must initialize it with the ?IESS system call. Input to the ?IESS system call includes the starting address of the ESS (in the unshared area of your logical address space), and a pointer to a block of page 0 locations in your logical address space. When AOS/VS schedules a 16–bit task, it copies the ESS information to the designated page 0 area. When rescheduling occurs, AOS/VS transfers the ESS information back to the ESS block in the unshared area of your logical address space.

End of Chapter

Appendix A
System Log Record Format

This appendix describes the format of the *system log (SYSLOG) file*, into which both AOS/VS and privileged processes can write records that log the occurrence of certain events.

Each record in the SYSLOG file contains an *event code* that identifies the event that it is logging. The event code can be either one of the standard AOS/VS event codes, or a special event code that AOS/VS allows you to define within your programs.

To write a record into the SYSLOG file, a process must issue the ?LOGEV system call; to issue the ?LOGEV call, the process must have Superuser mode turned on.

This appendix lists the standard AOS/VS event codes and what they represent. It also describes the format of the records that log these events.

Reporting the Contents of the SYSLOG File

You can report the contents of the SYSLOG file by either using the CLI REPORT command, or by writing a program that reads and reports on the contents of the SYSLOG file.

If you use the CLI REPORT command, it will report on *only* those records that contain the standard AOS/VS event codes. Since you define the meaning of any special event codes, *and* specify the contents of the records that describe those events, the REPORT command cannot report on the events you log with those records.

If you have defined special event codes and you want to read and report on the records that contain them, or if you want to report on only certain standard AOS/VS events, you must write a program to do so.

Reading the SYSLOG File

To read the SYSLOG file, you should open it for dynamic reads. You can declare an integer*2 array for the record header using a 0 base (e.g., in FORTRAN 77, INTEGER*2 HEADER(0:7)). This allows you to use the 0-base subscripts that we show. You can also declare an integer*2 array for each different record length, less 8 elements. At runtime, you can read the header array, check the length from words 0 and 1. Then, if the record has information other than the header, you can read the remainder of the record into the array of its length. Then, in the header array, you can check word 5 for the message code. If this is a code you want, you can break down the header and array, and format them for output, and then read the next record. If you don't care about the code, you can simply read the next record.

You can get the record length from the two-word length descriptor by — in FORTRAN 77 — equivalencing element 0 of the descriptor to a 4–byte integer.

Most record formats have a specific length. Formats for events like process creations and file opens, however, don't have a specific length. Instead, their length varies with the length

of the username and/or pathname stored in them. These flexible format records are recorded *only* when logging was turned on with /DETAIL=FULL. The flexible record lengths help conserve disk space (since otherwise every record would have to be as long as the longest one — and these events occur quite often). The *only* records with flexible length are

- Process events. These include process creation (code 910), chaining to another process (917), and loading a program into a ring (ring load, ?RINGLD system call, 916).
- File access events. Events with flexible-length records are file create (code 929); file open (code 920); file delete (code 924), file rename (codes 938 and 942); initialize LDU (code 937); release LDU (code 928); change file ACL (codes 939 and 943); read user data area, UDA (code 925); and write UDA (code 926).

In each of these flexible length records — as in all records — the record length appears in the first two 16–bit words in each record.

Record Header Format

Figure A–1 shows the header that begins each log record. The sixth 16–bit word in the header is the event code. In this and the following figures, all subscript/offsets and event codes are decimal. In the records themselves, all numeric values are octal.

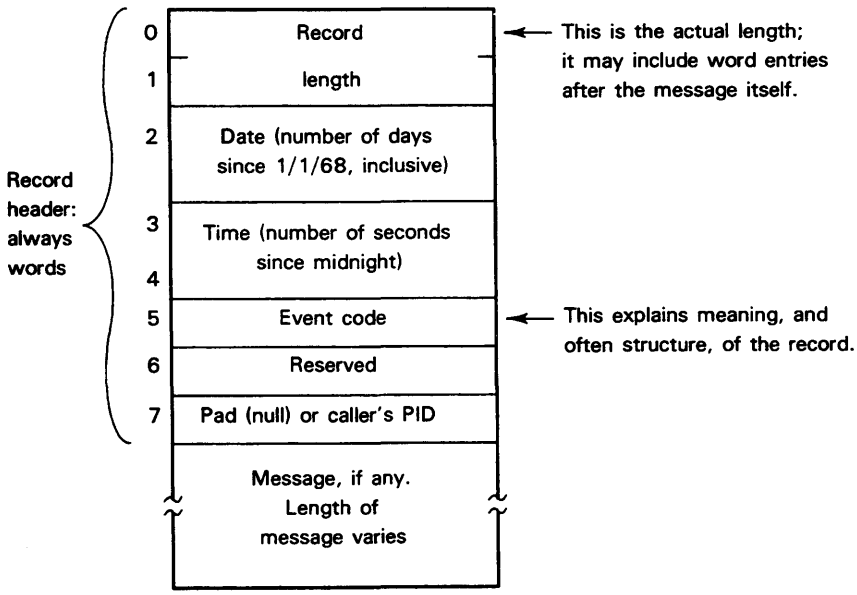


Figure A–1. Log Record Header

SYSLOG Record Formats

The SYSLOG file stores the following standard record types. AOS/VS writes these records into the SYSLOG file; however, AOS/VS writes those record types for which we note "Error Log" into the system's error log file.

Table A–1 summarizes all record types by event code, including their record length. Records with length noted as 0 consist only of the header.

Figure A–2 describes records that are longer than the header. The SYSLOG file stores numeric values in octal and ASCII characters as ASCII. Where padding is needed, SYSLOG uses nulls (ASCII 000). The symbol # means "number."

Event codes 900–999 and 1200–1299 are logged only when system logging was turned on with the /DETAIL=FULL switch. *Do not use codes 900–999 and 1200–1299 for your special event codes.*

Table A–1. SYSLOG Event Codes and Record Lengths

Event Code (decimal)	Meaning	Message Length (Words 8 through n)
0	Unused	
1	SYSLOG logging turned on (see also error code 901.	0
2	SYSLOG logging turned off (see also error code 901.	8 through 1016 (varies).
3	Process termination.	24
4	Device error — Error log.	32
5	Pad record (padding only).	8 through 148 (varies).
6	Unused by MV/Family processors.	3
7	Power failure — Error log.	3
8	AOS/VS and log file revision (follows *logging started* record, 1).	4
40	Power restored — Error log.	0
41	Fatal AOS/VS error — Error log.	33
42	AOS/VS hang — Error log.	0
43	Single-bit ERCC error, MV/4000 — Error log.	16
44	Multibit ERCC errors, MV/4000 — Error log.	16

Table A-1. SYSLOG Event Codes and Record Lengths

Event Code (decimal)	Meaning	Message Length (Words 8 through n)
45	ERCC Sniff I/O errors, MV/4000 — Error log.	16
73	SCP reset (not recorded).	0
74	SCP request completed (not recorded).	0
75	Host-SCP error (not recorded).	0
76	Host-SCP buffer full error — Error log.	0
77	SCP time-out — Error log.	0
78	SCP interface degrade — Error log.	0
79	SCP-request-to-host error — Error log.	0
80	SCP buffer not cleared — Error log.	0
81	Host-request-to-SCP error — Error log.	0
96	SCP logging enabled — Error log.	0
97	SCP logging disabled — Error log.	0
98	Main processor halt — Error log.	0
99	BOOT issued (MV/8000) — Error log.	0
100	Power failure — Error log.	0
101	Power restore — Error log.	0
102	Air flow fault — Error log.	0
103	Overtemp fault (not recorded).	0
104	Transfer to battery backup — Error log.	0
105	Reserved.	—
106	ERCC error, MV/8000 and MV/6000 — Error log.	16
107	Microsequencer parity error — Error log.	0
108	System cache parity error — Error log.	0
109	Cache to Bank controller parity error — Error log.	0

Table A-1. SYSLOG Event Codes and Record Lengths

Event Code (decimal)	Meaning	Message Length (Words 8 through n)
110	IOC bus parity error — Error log.	0
111	S-bus timeout — Error log.	0
112	S-bus parity error — Error log.	0
113	Operating system error (unused).	0
114	Diskette log error (MV/8000).	0
115	Infinite protection fault — Error log.	0
116	Infinite page fault — Error log.	0
117	Instruction cache enabled — Error log.	0
118	Instruction cache disabled — Error log.	0
119	Reserved.	—
120	Reserved.	—
121	System reset (not recorded).	0
122	ATU accelerator enabled — Error log.	0
123	ATU accelerator disabled — Error log.	0
125	XEQ DTOS command (MV/8000).	0
126	Bad return from DTOS (MV/8000).	0
127	HALT command (MV/8000) — Error log.	0
128	CONTINUE command (MV/8000) — Error log.	0
129	START command (MV/8000) — Error log.	0
130	INIT command (MV/8000) — Error log.	0
131	Bank controller ERCC report disable — Error log.	0
132	Good return from DTOS (MV/8000).	0
133	Hard interrupt (not recorded).	0
901	Reserved.	0
910	Process created.	Varies

Table A-1. SYSLOG Event Codes and Record Lengths

Event Code (decimal)	Meaning	Message Length (Words 8 through n)
911	Reserved.	0
912	Process terminated by superior process.	4
913	Superuser turned on or off.	4
914	Superprocess turned on or off.	4
915	Access devices turned on or off (?IDEF turns on; ?IRMV turns off).	4
916	Process loaded program into ring (used ?RINGLD system call).	Varies
917	Process chained to another process.	Varies
918	PMGR assigned a console to a process.	5
919	PMGR revoked a process's console assignment.	5
920	File opened.	Varies
921	Reserved.	—
922	File closed.	4
923	Reserved.	—
924	File deleted, access by pathname see also error code 931).	Varies
925	File user data area (UDA) read, access by pathname (also see 932).	Varies
926	File UDA written, access by pathname (also see 933).	Varies
927	File UDA created, access by pathname (see also error code 934).	Varies
928	Logical disk unit (LDU) released, access by pathnname.	Varies
929	File created, access by pathname.	Varies
930	Reserved.	—
931	File deleted, access by channel number.	4

Table A-1. SYSLOG Event Codes and Record Lengths

Event Code (decimal)	Meaning	Message Length (Words 8 through n)
932	File user data area (UDA) read, access by channel number.	4
933	File UDA written, access by channel number.	4
934	File UDA created, access by channel number.	4
935	Reserved.	—
936	Reserved.	—
937	Logical disk unit (LDU) initialized.	Varies
938	File renamed, access by pathname (see also error code 942)	Varies
939	File ACL changed, access by pathname (see also error code 943)	Varies
940	Reserved.	—
941	Reserved.	—
942	File renamed, access by channel number.	Varies
943	File ACL changed, access by channel number.	Varies
944	Reserved.	—
945	Shared file opened, first open.	Varies
946	Shared file opened, subsequent open.	7
947	Permit access to protected file call (used to modify shared file) occurred.	9
948	Job processor initialized (?JPINIT system call issued).	17
949	Job processor released (?JPREL system call issued).	17
950	Job processor moved to another logical processor (?JPMO system call issued).	17
951	Job processor status requested (?JPSTAT system call issued).	17

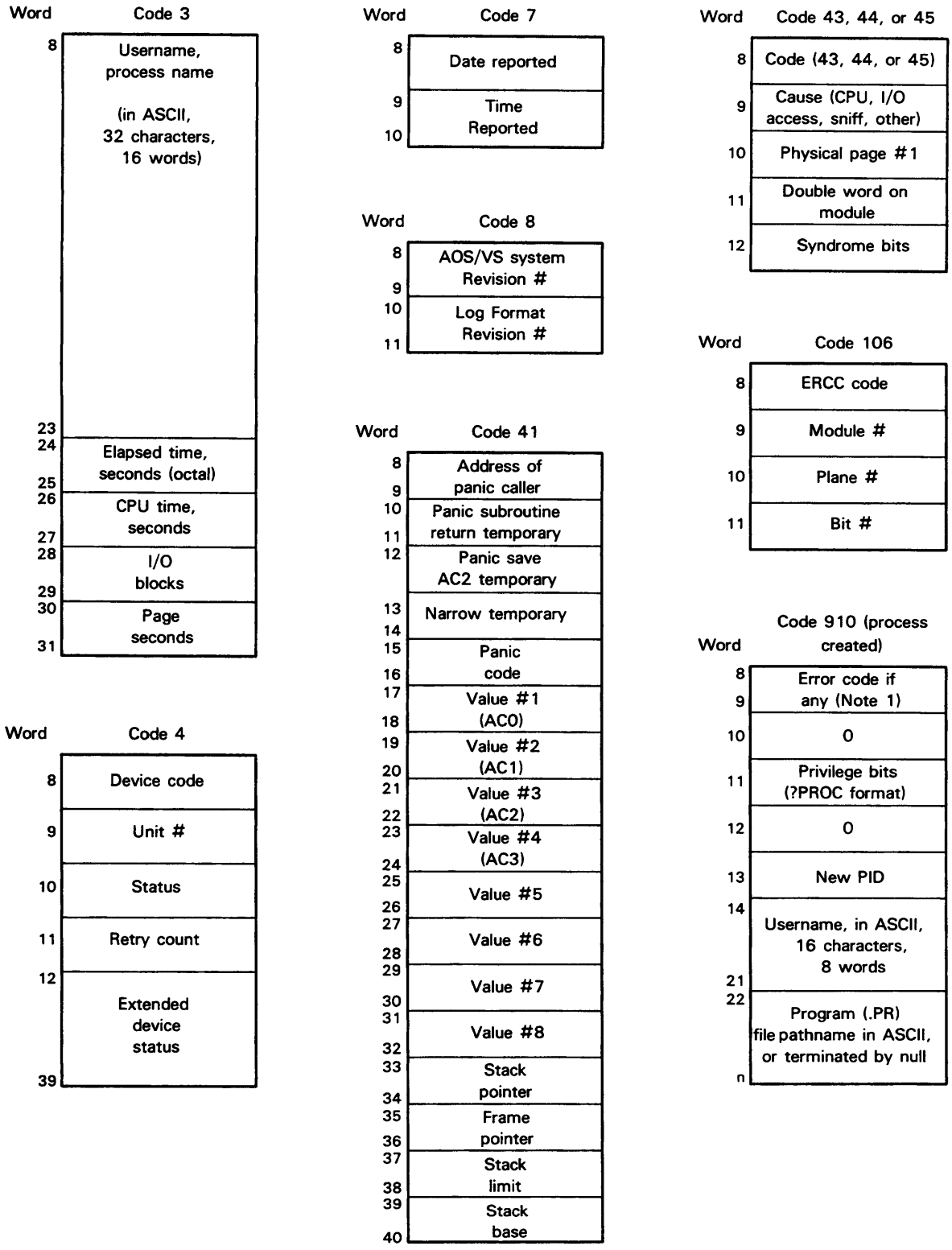
Table A-1. SYSLOG Event Codes and Record Lengths

Event Code (decimal)	Meaning	Message Length (Words 8 through n)
952	Logical processor created (?LPCREA system call issued).	17
953	Logical processor deleted (?LPDELE system call issued).	17
954	Logical processor status requested (?LPSTAT system call issued).	17
955	Class assignments requested or set (?LPCLASS system call issued).	17
956	Class IDs requested or set (?CLASS system call issued).	17
957	Class scheduling matrix requested or set (?CMATRIX system call issued).	17
958	Class scheduling status requested (?CLSTAT system call issued).	17
959	Class scheduling enabled or disabled (?CLSCHED system call issued).	17
960	Process's user locality changed (?LOCALITY system call issued).	17
961	?MIRROR system call issued successfully.	17
962	?MIRROR system call issued unsuccessfully.	17
963	LDU mirror image released by system.	17
964	System Manager privilege (SYSMGR) turned on or off.	17
1023	UPSC power supply fault.	25
1024	Console connect time.	25
1025	Unit mount time.	25
1026	Privileged user logon.	9
1027	Pages printed.	25
1028	Reserved.	—
1029	Reserved.	—
1030	RMA accounting.	12

Table A-1. SYSLOG Event Codes and Record Lengths

Event Code (decimal)	Meaning	Message Length (Words 8 through n)
1031	Reserved.	—
1064	FTA accounting.	21
1065	General event (LOGEVENT).	Varies
1066	DG/SNA accounting.	21
1067	X.25 Accounting.	38
1068	X.25 Error.	14
1069	Message Transfer Service (MTA) accounting.	47
1200	Reserved.	0
1201	Reserved.	0
1211	Labeled medium (tape) mounted.	28
1212	Labeled medium (tape) dismounted.	28
1213	User logon.	26
1214	File printed.	154
1215	Invalid logon attempt.	25
1220	User profile created.	17
1221	User profile deleted.	17
1222	User profile renamed.	33
1223	User profile opened (e.g., by EXEC when a user starts to log on).	17
1224	User profile read (follows event code 1223).	18
1225	User profile written to (follows event 1223).	18
1226	User profile closed (follows event 1223).	17
1227	Reserved.	—
1228	Reserved.	—
1229	LOCK_CLI locked or unlocked.	—

NOTE: Codes 900-999 and 1200-1299 are logged only if logging was enabled with /DETAIL=FULL.



ID-03322

Figure A–2. Log Record Codes, Events, and Message Lengths, Excluding Header (continues)

Word Code 912
(process terminated)

8	Error code if any (Note 1)
9	
10	Target PID
11	

Word Code 913, 914, 915
(Super mode activity)

8	Error code if any (Note 1)
9	
10	n (0 means no change, 1 means off, 2 means on)
11	

Word Code 916 (?RINGLD)

8	Error code if any (Note 1)
9	
10	0
11	Ring number
12	Program (.PR) file pathname loaded, in ASCII, terminated by null
n	

Word Code 917 (chain)

8	Error code if any (Note 1)
9	
10	Program (.PR) file pathname, in ASCII, terminated by null
n	

Codes 918, 919
(Assign/revoke assignment
Word of terminal by PMGR)

8	Error Code, if any.
9	
10	PID of process receiving/losing terminal assignment.
11	
	Device code of IAC or system console.
	Secondary code (terminal line #).

Code 920
Word (file opened)

8	Error code if any (Note 1)
9	
10	Flag word (Note 2)
11	Channel number
12	Pathname, in ASCII, terminated by null
n	

Codes 922, 931, 932, 933,
and 934 (file accessed by
Word channel number)

8	Error code if any (Note 1)
9	
10	Flag word (Note 2)
11	Channel number

Codes 924, 925, 926, 927,
928, 929 (file deleted, UDA
accessed, LDU released,
Word file created)

8	Error code if any (Note 1)
9	
10	Flag word (Note 2)
11	Pathname, in ASCII, terminated by null (for code 927, pathname is the LDU name)
n	

Code 937
Word (LDU initialized)

8	Error code if any (Note 1)
9	
10	Flag word (Note 2)
11	Disk unit name(s) in LDU (e.g., @DPJ10), in ASCII, separated by null (if more than one), ends
n	with two nulls.
n + 1	LDU name, in ASCII,
n + m	terminated by null

Code 938
Word (file renamed)

8	Error code if any (Note 1)
9	
10	Flag word (Note 2)
11	New filename, in ASCII, terminated by null
n + 1	Original pathname,
n + m	in ASCII, terminated by null

Code 939
Word (file ACL changed)

8	Error code if any (Note 1)
9	
10	Flag word (Note 2)
11	New ACL in call ?SACL format: user, null, value, user, null, value, and so on (user is the username, in ASCII; value is the access value, explained in Note 3); a null terminates the ACL.
n + 1	Pathname, in ASCII,
n + m	terminated by null.

ID-03322

Figure A–2. Log Record Codes, Events, and Message Lengths, Excluding Header (continued)

Code 942 (file renamed by channel #)

Word	
8	Error code if any (Note 1)
9	
10	Flag word (Note 2)
11	Channel number
12	New filename, in ASCII, terminated by null
n	

Code 943 (file ACL changed by channel #)

Word	
8	Error code if any (Note 1)
9	
10	Flag word (Note 2)
11	Channel number
12	New ACL in call ?SACL format, as described in 939 above; ends with a null.
n	

Code 945 (shared file opened, first open)

Word	
8	Error code if any (Note 1)
9	
10	Flag word (Note 2)
11	Channel number
12	File identifier
13	
14	Caller's ring
15	Pathname, in ASCII, terminated by null

Code 946 (shared file opened, subsequent open)

Word	
8	Error code if any (Note 1)
9	
10	Flag word (Note 2)
11	Channel number
12	File identifier
13	
14	Caller's ring

Code 947 (permitted access to, protected word file)

Word	
8	Error code if any (Note 1)
9	
10	ACL value (Note 3)
11	File identifier
12	
13	Caller's ring
14	Target ring
15	Target PID

Codes 948 through 960

Word	
10	Error code, if any.
11	
12	Reserved.
17	

Code 1023

Word	
8	UPSC power supply fault code

Code 1024 or 1025

Word	
8	Username (in ASCII, 16 characters, 8 words)
15	
16	Device name (in ASCII, 32 characters, 16 words)
31	
32	Minutes used

Code 1026

Word	
8	Username (in ASCII, 16 characters, 8 words)
15	
16	Privilege bits (?PROC format)

Code 1027

Word	
8	Username (in ASCII, 16 characters, 8 words)
15	
16	Device name (in ASCII, 32 characters, 16 words)
31	
32	Pages printed

ID-03322

Figure A–2. Log Record Codes, Events, and Message Lengths, Excluding Header (continued)

Word Code 1030

Word	
8	Remote host ID
9	Virtual Circuit #1
10	AOS/VS error code
11	Username (in ASCII, 16 characters, 8 words)
18	
19	Not used
20	
21	Connect time
22	
22	Request count
23	

Word Code 1031

Word	
8	Username (in ASCII, 16 characters, 8 words)
15	
16	Remote host #
17	Virtual Circuit #1
18	Termination code

Word Code 1064

Word	
8	Remote host ID
9	Virtual Circuit #
10	AOS/VS error code
11	Username (in ASCII, 16 characters, 8 words)
18	
19	Not used
20	
21	Connect time
22	
23	Service type
24	I/O block count
25	
26	# of packets sent
27	
28	# of packets received
29	
30	# of bytes sent
31	
32	# of bytes received
33	

Word Code 1065

Word	
8	Message (in ASCII, padded to multiple of 8, max is 48 characters, 24 words)
31	

Word Code 1066

Word	
8	Logical unit (LU) address
9	Time LU closed
10	
11	Time LU opened
12	
13	# of request units received
14	
15	# of bytes received
16	
17	# of Request units sent
18	
19	# of bytes sent
20	
21	SNA customer name (username: processname, in ASCII. 18 characters, 9 words)
29	

Word Code 1067

Word	
8	Linkname (in ASCII, 16 characters, 8 words)
15	
16	Channel #
17	Virtual Circuit #
18	Connect type
19	Reason code
20	Username: processname (in ASCII, 32 characters, 16 words)
35	
36	Connection address (in ASCII, 15 characters, 8 words)
43	
44	Connect time
45	
46	# of packets sent
47	# of packets received
48	# of bytes sent
49	
50	# of bytes received
51	

ID-03322

Figure A–2. Log Record Codes, Events, and Message Lengths, Excluding Header (continued)

Code 1068

Word	
8	Linkname (in ASCII, 16 characters, 8 words)
15	
16	Channel #
17	Virtual Circuit #
18	Flag word
19	XODIAC error code
20	Diagnostic error code
21	AOS/VS error code

Code 1069 (MTA Accounting)

Word	
8	Remote host ID
9	Virtual Circuit ID
10	Error Code
11 . . . 18	Username (16 characters)
19 20	Reserved
21 22	Connect time
23	Service type
24 25	I/O block count
26 27	# of packets sent
28 29	# of packets received
30 31	# of bytes sent
32 33	# of bytes received
34 . . . 47	Reserved (14 words).

Codes 1211 and 1212 (labeled medium mounted and dismounted)

Word	
8	Error code if any (Note 1)
9	
10	Username (in ASCII, 16 characters, 8 words)
17	
18	Device name (in ASCII, 32 characters, 16 words)
33	
34	Volume ID (volid), 6 characters, 3 words
36	

Code 1213 (user log on)

Word	
8	Error code if any (Note 1)
9	
10	Username (in ASCII, 16 characters, 8 words)
17	
18	Console name (in ASCII, 32 characters, 16 words)
33	
34	Privileges (?PROFILE call format)

Code 1214 (file printed)

Word	
8	Error code if any (Note 1)
9	
10	Username (in ASCII, 16 characters, 8 words)
17	
18	Device name (in ASCII, 32 characters, 16 words)
33	
34	# of pages printed
35	Pathname of file (in ASCII, 256 characters, 128 words)
162	

Code 1215 (invalid log on attempt)

Word	
8	Error code if any (Note 1)
9	
10	Username (in ASCII, 16 characters, 8 words)
17	
18	Console name (in ASCII, 32 characters, 16 words)
33	

Codes 1220, 1221, 1223, and 1226 (user profile created, deleted, opened, and closed)

Word	
8	Error code if any (Note 1)
9	
10	Profile name (in ASCII, 32 characters, 16 words; terminated by null)
25	

ID-03322

Figure A–2. Log Record Codes, Events, and Message Lengths, Excluding Header (continued)

Code 1222 (USER profile renamed)

Word	
8	Error code if any (Note 1)
9	
10	Original profile name (in ASCII, 32 characters, 16 words; terminated by null)
25	
26	New profile name (in ASCII, 32 characters, 16 words)
41	

Codes 1224 and 1225 (user profile, field read or written)

Word	
8	Error code if any (Note 1)
9	
10	profile name (in ASCII, 32 characters, 16 words; terminated by null)
25	
26	Field number in descriptior packet (?profile call format)
27	

Code 1229 (locCLI locked or unlocked.)

Word	
8	Error code if any (Note 1)
9	
10	n (0 means no change, 1 means unlocked, 2 means locked.)
11	

Note 1. If an error occurred on this event, the AOS/VS error code is stored, right-justified, in these two 16-bit words (this 32-bit double word). For example, the the value 242 (octal) in this double word means the user's access was rejected a *EXECUTE ACCESS DENIED* error message. If no error occurred on the event, the value of the double word is 0.

Note 2. The flag word normally contains 0, unless something extraordinary happened to prevent the complete record from being written (for example, a hard error on disk). the error code field within the record will probably not show an error, since it records things like user access privilege errors. The flag word codes (numeric values) are

040000 Pathname field not available. This means a problem (like a hard error on disk, also shown in ERROR_LOG) prevented the entire pathname from being recorded. It can also occur on an attempt to initialize an LDU if the user lacked E access to the PER entry for the unit.

020000 This means the first variable field was incomplete (perhaps) for the same reasons as flag code 40000). The first variable field occurs only in records with codes 937, 938, 939, 942, and 943 (LDU initialize, file rename, and ACL change).

010000 This means that one or more fixed fields (like channel number) were not available. It can happen on ?OPEN or ?SOPPF if the pathname resolution fails.

Note 3. The file ACL value is a composite of the value of bit settings. The structure of log records longer than 8 words is as follows:

Bit	11	12	13	14	15
	O	W	A	R	E
	digit		digit		

Thus OWARE access has the value 37, WARE 17, ARE 7, RE 3, and E 1.

ID-03322

Figure A–2. Log Record Codes, Events, and Message Lengths, Excluding Header (concluded)

Anatomy of a System Log File Record

An easy way to examine a system log file is to use the DISPLAY utility. You can apply the /DECIMAL switch to get a decimal display. Figure A–3 shows both an octal and decimal display of a logfile. You can get similar displays via X DISPLAY :SYSLOG and X DISPLAY/DECIMAL :SYSLOG.

```
  0 000000 000010 011635 000000 101443 000001 000000 000000  .........#......
 10 000000 000040 011635 000000 101475 000003 000000 000000  ... .....=......
 20 047520 035060 030063 000000 000000 000000 000000 000000  OP:003..........
 30 000000 000000 000000 000000 000000 000000 000000 000000  ................
 40 000000 000002 000000 000320 000000 000011 000000 000012  ................
  .
400 000000 000050 011635 000000 102146 002001 000000 000000  ...(.....f......
410 045101 041513 000000 000000 000000 000000 000000 000000  JACK............
420 046524 041060 000000 000000 000000 000000 000000 000000  MTBO............
430 000000 000000 000000 000000 000000 000000 000000 000000  ................
440 000003 000000 000000 000000 000000 000000 000000 000001  ................
  .
1410 000000 000020 011635 000000 120060 002051 000000 000000  .........0.)....
1420 051165 067156 064556 063440 051145 073040 031056 030060  Running Rev 2.00
1430 000000 000020 011635 000000 120102 002051 000000 000000  .........B.)....
1440 046117 043505 053105 047124 020114 064566 062563 020441  LOGEVENT Lives!
  .


   0     0     8  5021     0 33571     1     0     0  .........#......
   8     0    32  5021     0 33597     3     0     0  ... .....=......
  16 20304 14896 12339     0     0     0     0     0  OP:003..........
  24     0     0     0     0     0     0     0     0  ................
  32     0     2     0   208     0     9     0    10  ................
  .
 256     0    40  5021     0 33894  1025     0     0  ...(.....f......
 264 19009 17227     0     0     0     0     0     0  JACK............
 272 19796 16944     0     0     0     0     0     0  MTBO............
 280     0     0     0     0     0     0     0     0  ................
 288     3     0     0     0     0     0     0     1  ................
  .

 776     0    16  5021     0 41008  1065     0     0  .........0.)....
 784 21109 28270 26990 26400 21093 30240 12846 12336  Running Rev 2.00
 792     0    16  5021     0 41026  1065     0     0  .........B.)....
 800 19535 18245 22085 20052  8268 26998 25971  8481  LOGEVENT Lives!!
  .
```

Figure A–3. An Octal and Decimal DISPLAY of a System Log File

The first record starts with number 0 in both the octal and decimal display. The second word in this record appears as 000010 in the octal display, 8 in the decimal display. This word tells the length. For this record it is 10 octal (8) words. The code, in the fifth word of the record, is 1. From Figure A–2, you can tell that code 1 means that system logging was on.

Figure A–4 shows a breakdown of the first record in octal and decimal.

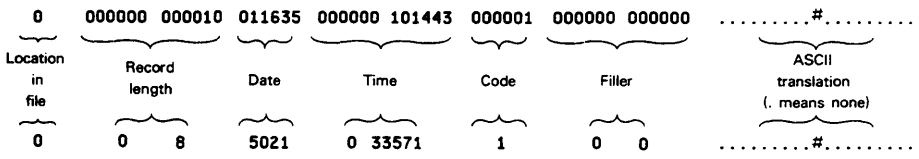


Figure A–4. Octal and Decimal Versions of a SYSLOG Record

The second record starts at location 10 octal (8). It is 40 octal (32) words long. The code in the fifth word is 3, which means that the record is a process termination message. The ASCII appears in the field off to the right (this field is not in the record, but is a convenience supplied by DISPLAY). Looking at the fields of this record, you can see (in the last line) the elapsed time, processor time, I/O blocks, and page-seconds.

The third record shown appears later on in the file. It has a record length of 50 octal (40). The fifth word, 2001 octal (1025), describes it as a unit mount. Again, you can see the ASCII off to the right. JACK used the mounted tape for 3 minutes.

The fourth and fifth records appear still later in the file. Each is 24 octal (16) words long. The fifth word in each header is 2051 (1065). This means it is an event record, written by the LOGEVENT command. Once more, you can see the ASCII on the right.

The report that the REPORT program would generate from this log file would have a lot of columnar information, and its numbers would be decimal — as shown in the earlier REPORT figures.

End of Appendix

Glossary

A–type process
A process that has a PID between 1 and 255. This process can't execute any program if PIDs 1–255 are in use. Error conditions may result if a process with a PID over 255 tries to communicate with an A–type process. A is the PID-size type of all processes before AOS/VS revision 7.00 (except the CLI and EXEC).

anyPID program
A program that can run at any PID up to the maximum specified at VSGEN. The system will run it above 255 if possible. An anyPID program's program file has been edited with the SPRED editor and its PID-size type made anyPID.

AOS/DVS
Advanced Operating System/Distributed Virtual System, DG's 32–bit operating system for distributed computing.

B-type process
A B-type process has a PID between 1 and 255. It can't run if PIDs 1–255 are in use, but it can create and communicate with a process of any PID-size type. Most DG programs, including the CLI and EXEC, run as B–type processes. By default, the CLI run for each user is a B–type process. Thus, by default, a user CLI must run in the range the range 1–255 but can execute any PID-type program. Most processes from programs supplied with AOS/VS are B–type processes.

Block I/O
One of two input/output modes in which you can access a file. Information is transferred in 512–byte disk blocks, magnetic tape blocks, MCA blocks.

AOS/VS always performs I/O in block units, whether you employ block or record I/O.

Block length
The number of bytes per block. (See also, *Block.*)

Break file
A status file in which AOS/VS, under certain conditions, saves the state of a terminated process.

C–type process
A C–type process has a PID above 255 or below 255, depending on what PIDs are free. A C–type process can execute any PID-size type program. But error conditions may arise after it executes a smallPID program (since the father process has a PID the son can't understand).

Character device
A device that performs I/O in byte units. CRT terminals and hard-copy terminals are typical character devices.

Child processor
A job processor (physical processor) other than the primary (mother) processor. Mother and child processors exist only in computers with more than one job processor. (See also, *mother processor.*)

Class
A set of processes that receive special scheduling treatment.

CLI
See *Command line interpreter (CLI).*

Command line interpreter (CLI)
A utility that is the main interface between you and the system. The CLI accepts your command lines and (among other functions) translates this input into commands for other utilities, or into commands that directly perform functions such as file maintenance.

Compute-bound process
A process that — within a given interval — issues many processor instructions. These instructions might involve computations, or data comparisons and sorts. Conversely, a process that demands more I/O than processor attention is *I/O bound.* Multiple job processors (and classes and logical processors) are most efficient when a system runs compute-bound (not I/O bound) processes.

Connection table
A table in which AOS/VS writes an entry to manage exchanges between customers and servers.

Control character
A keyboard character that you type while you press the CTRL key.

Control point directory (CPD)
A directory in an LD that contains two variables: CS, the amount of space currently allocated; and MS, the maximum amount of space available in the directory. CPDs allow you to control the system's disk space allocation.

Control sequence
A CTRL–C followed by any control character. (See also, *Control character.*)

CPD
See *Control point directory (CPD).*

Critical region
A procedure or database shared by all tasks, but available to only one task at a time.

CS (current space)
The amount of space currently allocated in a CPD. (See also, *Control point directory (CPD).*)

Data-link control character
A synchronization character recognized by both sending and receiving BSC stations.

Data-sensitive record type
A record type whose records consist of character strings terminated by one of the default delimiters, NEW LINE, carriage return, null, or form feed, or terminated by a user-defined delimiter.

Dedicated line
A communications line that continuously connects two or more stations, regardless of the amount of time the line is actually in use.

Dedicated pages
Memory pages that AOS/VS reserves for specific purposes, including physical pages occupied by the resident portion of the operating system and pages wired to a resident process.

Demand paging
Moving logical pages from the disk to memory as a process refers to (demands) those pages.

Device
A hardware peripheral component; each type of device has unique operating characteristics. Devices are either character-oriented (send or receive single bytes of data) or block-oriented (send or receive data in multibyte blocks).

Device independence
The ability of a process to communicate with a device without regard to the unique nature of the device.

Directory
A file that catalogs files and allows qualified users to access them. Directories are connected in a structure that resembles an inverted tree. On this tree, the lower directories are inferior to the higher directories. Each directory contains an entry for any directory that is immediately inferior to itself.

Directory entry
A unit of information contained in a directory; a directory can contain multiple entries. A common type of entry is that which lists certain information about a file in the directory. Examples of other types of entries are IPC entries and links. (See also, *File status*.)

Disk
A magnetic recording medium (for example, disk pack, disk cartridge, diskette, fixed-head disk).

Disk address
The location of a block on a disk. (See also, *Disk block*.)

Disk block
The smallest allocatable unit of disk memory, standardized as 512 bytes.

Disk controller
A mechanism that directs the operator of one or more disk units. A program can direct the operation of a disk controller.

Disk controller name
The name of a disk controller, consisting of three letters and possibly one decimal digit; for example, DPE and DPE1.

Disk drive
See *Disk unit*.

Disk unit
A mechanism that physically reads from and writes to disk.

Disk unit name
The name of a disk unit, consisting of the name of a disk controller followed by a decimal digit; for example, DPE0 and DPE10.

Dormant state
One of four task states, in which a task exists that has not yet been initiated (made known to the operating system) or that has terminated execution.

Double connection
A connection in which each process can act as either the customer or the server of the other, depending on the action to be performed.

Dynamic record type
A record type in which you specify the record length when you read or write.

Eligible process
A process that has been allocated main memory, which allows it to compete for control of the processor with other such processes, based on its proces type and its priority. (This is one of three process states.)

Error code
A 32–bit unsigned value that AOS/VS returns in AC0 to indicate an exceptional condition. (This exceptional condition may or may not indicate an actual error.) Each error code has a text string associated with it. (See the description of the ?ERMSG system call for information on getting the text string associated with a particular error code.)

Exceptional condition code
See *Error code.*

Executable file
A binary memory-image file that you can read into main memory from a peripheral storage device for exection; a program that can run.

Executable task
A task that has control of a processor. Only one task at a time can be executing. (This is one of four task states.)

File
A collection of related data treated as a unit. A file can contain up to 2**32 bytes of data. Disk and magnetic tape can contain one or more files.

File element
The basic unit of storage in the AOS/VS disk file organization. Each file element consists of one or more contiguous blocks. You specify file element size when a file is first created. If a file grows, it grows in units of the file element size.

File status
A collection of information about each file. This information includes the file size, time of creation, and other details.

File system
See *Hierarchical file system.*

Filename
An alphanumeric file identifier. All filenames in a single directory must be unique, and each can contain no more than 31 characters.

Fixed-length record type
A record type in which you specify a predefined, common record length.

Form name
The name of a file in the :UTIL:FORMS directory, which was created with the CLI Forms Control Utility (FCU). The form name must contain from 1 through 31 legal filename characters.

Gate
An entry point to code in an inner ring.

Global port number
A number made up of a port's PID, ring number, and local port number, which uniquely identifies that port system-wide.

Global server
A separate process that performs functions on behalf of a customer process. (The servers that are described in Chapter 9 are global servers.)

Hierarchical file system
The inverted tree structure in which AOS/VS organizes files and directories. The highest directory in the hierarchy is the system root, which points to inferior directories; these, in turn, point to inferior directories. Any process with proper privileges can access any file within any directory.

High-order bits
The 16 most significant bits in a 32–bit value; that is, Bits 0 through 15.

Histogram
A data array that provides a global view of processor activity.

Hybrid program
A program that cannot run if PIDs 1–255 are in use, like a smallPID program. A hybrid program, however, can communicate with processes with PIDs above 255. A hybrid program's program file has been edited with the SPRED editor and its PID-size type made hybrid. Most programs shipped with AOS/VS Revision 7.00 are hybrid programs.

Image
See *LDU image.*

Index
A single block that lists the address of each file element.

Ineligible process
An process that has not been allocated main memory, but in all other ways is ready to run. (This is one of three process states.)

Initial task
The first task that executes in a process. AOS/VS assigns the initial task TID 1, priority 0.

Interprocess communication facility (IPC)
A generalized AOS/VS facility that sends free-format messages of any length between any two processes. IPC messages are sent between ports. (See also, *Port.*)

IPC
See *Interprocess communication facility (IPC).*

Job processor
A hardware entity that computes and interprets program instructions. The term includes and extends the standard definition of central processing unit (CPU). When AOS/VS starts up, it recognizes only the default processor, JP0. If your computer has more than one job processor, you must initialize the additional job processors. In a system with multiple job processors, the default processor, JP0, is called the mother. Each additional processor is called a child.

Jobname
A name that identifies a batch job. A jobname must contain from 1 through 31 legal filename characters.

JPID
The ID number that AOS/VS uses to identify job processors.

K
An abbreviation for the decimal number 1024. Thus, 32 Kbytes of memory are 32,768 bytes.

Kernel
The part of the operating system that contains device drivers, system parameter tables, and other things. It talks to the hardware, PMGR, and — through the Agent — to users.

Keyword switch
A two-part switch of the following form: /keyword=value. For example, /L=filename is a keyword switch.

LD
See *Logical disk (LD).*

LDU image
If two LDUs have the same name but different LDU unique IDs, they are said to be images of each other. An image may be synchronized (data is the same) or unsynchronized (data is different). (See also, *mirroring).*

LDU unique ID
The LDU unique ID is a six-character field in the Disk Information Block (DIB) you create with the Disk Formatter. The system and system utilities can tell from the LDU unique ID whether the LDU can be part of a mirrored LDU.

LEF mode
The CPU state that protects the system's I/O devices from unauthorized access. I/O instructions and LEF instructions use the same bit patterns. AOS/VS determines how to interpret these instructions by checking the state of LEF mode and the state of I/O mode. (LEF mode and I/O mode are mutually exclusive states.)

Link entry
A file that contains a pathname to another file.

Link-to-link reference
A link entry that is another link entry.

Load effective address mode
See *LEF mode.*

Local root
A single directory that acts as the foundation for a directory structure on a logical disk.

Local server
A server that shares the same logical address space as its customer. (Local servers can be loaded into the inner rings of your process.)

Locality
A number, or group of numbers, that determines the class of a process. There are two kinds of locality: user locality, (defined by PREDITOR in a user profile), and program locality (defined in a program file preamble by the SPRED utility).

Locality of reference
Clustering instructions and data by writing code in modular pieces.

Logical address space
The entire range of locations that a process can address. A process's user-visible logical address space can be up to 512 Mbytes for each ring.

Logical context
The total pages available to you (the user), including shared, unshared, and unused pages.

Logical disk (LD)
One or more physical disk units that you want to consider as a single logical unit.

Logical disk address
The location of a logical block on a logical disk. The address must include a disk pointer and a disk address to access the block.

Logical disk mirroring
See *mirroring.*

Logical disk name
The filename of a logical disk's root directory.

Logical processor
A special scheduling arrangement that allocates processor time to — usually — several process classes. You can create up to 16 logical processors.

Low-order bits
The 16 least significant bits in a 32–bit value; that is, Bits 16 through 31.

LP
See *logical processor.*

LPID
The ID number by which AOS/VS identifies a logical processor.

LRU chain
A list of released shared pages arranged in least recently used (LRU) order.

Main memory (physical)
Core or semiconductor storage, which contains computer instructions or data.

Master LD
A logical disk (LD) whose root becomes the system root (identified by a colon (:)). You must select the master LD.

MCA
See *Multiprocessor communications adaptor (MCA).*

Mirroring
Mirroring (also called logical disk mirroring) means having the operating system maintain two logically identical LDU images. (Mirroring implemented through the disk controller is also called hardware mirroring.) Mirrored LDUs provide high data availability since the system can continue to function on one image if the other image is taken out of service (as for backup) or if there is a hard disk error.

Modem
A communications device that translates analog signals to digital signals, and vice versa, over telephone lines.

Mother processor
The default job processor, JP0. If your computer has other job processors, each one is called a child processor.

MS (maximum space)
The maximum amount of space avilable in a CPD. (See also, *Control point directory (CPD).*)

Multidrop line
See *Multipoint line.*

Multilevel connection
A process that acts as both a server and a customer in a customer/server relationship.

Multipoint line
One of the two types of BSC line configurations (the other type is point-to-point). There is no contention between stations on a multipoint line.

Multiprocessor communications adaptor (MCA)
A device that permits communication between two Data General processors using the processors's data channels.

Multiprogramming
The ability to run an arbitrary number of independent processes. The system allocates its resources among these processes based on their priorities, types, or certain software events.

Multitasking process
A process in which more than one task is currently active.

Obituary message
A zero-length IPC message that is sent to a process when a customer or a server disconnects. (Obituary messages use the IPC system calls.) (See also, *Obituary notice.*)

Obituary notice
A signal that is sent when a customer or a server disconnects. (Obituary notices use the ?SIGNL, ?WTSIG, and SIGWT system calls.) (See also, *Obituary message.*)

Object code
Code, consisting of 32–bit instruction words and data words, which has been assembled or compiled from a source code file but not yet bound with other modules by the Link utility to make an executable program.

Object code file
A file containing object code, usually created by the Macroassembler or one of several high-level language compilers and having a filename ending in ".OB."

Overlay
A portion of a larger program that can be brought into main memory when it is needed (for only 16–bit processes).

Overlay area
A fixed-length storage area in a program in which different overlays can be read at different times while a program is executing (for only 16–bit processes).

Packet
A group of words in your address space that AOS/VS uses to get your input specifications and/or return output values. Many system calls require a packet.

Page
Memory storage area of 2 Kbytes (2048 bytes), starting on a 2 Kbyte boundary.

Page fault
A reference to a page that is not currently in the working set.

Parameter packet
See *Packet.*

Parametric code
Code in which system call packet offsets are cited by their mnemonic names, regardless of how the offsets are ordered in the packet figures.

Pathname
A name that identifies the location of a file within the system's files. A pathname may be a filename, or an optional list of directories followed by the file's name.

Physical disk
Same as disk.

PID
See *Process identifier (PID).*

PID 2
The initial operator process. (See also, *Process identifier (PID).*)

PID/ring tandem
Process identifier (PID)/ring-within-PID ordered pair. The connection management facility uses PID/ring tandems to identify all connections.

Point-to-point line
One of the two types of BSC line configurations (the other type is multipoint). Each station must bid for a point-to-point line.

Polling list
A series of contiguous words that contains each BSC tributary's poll address and device address.

Port
A data path to or from a process. The IPC facility sends messages between ports, which are full-duplex and can therefore send and receive data simultaneously. Each port is assigned a unique number (see also, *Interprocess communication facility (IPC).*)

Port numbers
The identification mechanism that allows two processes to send and receive messages via the IPC facility. The system maintains a directory of process numbers and associated port numbers.

Pre-emptible process
A process that the scheduler treats as a high-priority swappable process. (See also, *Swappable process.*)

Priority numbers
Values in the range from 0 (the highest priority) to 255 (the lowest priority) that determine the order in which tasks or processes execute.

Process
An executing set of instructions.

Process identifier (PID)
A number from 1 through 32 that you assign to identify each process.

Process name
A character string consisting of a username and a simple process name, with a colon (:) separating the two elements. AOS/VS uses process names and PIDs to identify each process.

Process priority
One of the factors that governs how the system allocates CPU time to a process. More than one process can have the same priority. (See also, *Priority number.*)

Process state
One of the factors that AOS/VS considers to determine the order in which it executes processes.

Process type
Process type governs when and for how long a process acquires main memory. The three process types are: resident, pre-emptible, and swappable.

Program
The current executable contents of a process's address space. A program contains the code paths executed by tasks. A process contains only one program at any given time; but during the execution of a process the current program may change many times.

Program file
A segment image linked for any one ring. (See also, *Segment image.*)

Resident process
A process that always remains in memory somewhere. (See also, *Swappable process* and *Pre-emptible process.*)

Ring
A physical barrier separating 512–Mbyte segments of main memory. AOS/VS protects system and sensitive user data by enforcing ring crossing protocols.

Ring maximization
A protection scheme in which 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. AOS/VS uses ring maximization to validate user-supplied channels, word pointers, or byte pointers for system calls. (See also, *Ring specification.*)

Ring specification
A protection scheme in which AOS/VS protects tasks executing in one user ring from interference by tasks executing in any other user rings. The connection management and IPC facilities use ring specification as their protection scheme.

Root process
The most superior process in the system hierarchy. All system processes and the initial process are sons of the root process.

Scalar notation
A time or date notation in which the current time equals the number of biseconds that have elapsed since midnight, and in which the date equal the number of days that have elapsed since 31 December 1967.

Search list
A list of directories that AOS/VS searches if it fails to find a specified file in your working directory. Each process has its own search list.

Segment
One of eight independent 512–Mbyte units connected by strict protocols that make up your logical address space. (See also, *Logical address space.*)

Segment image
A .PR file that AOS/VS has made part of a process's logical address space. (A segment image is a static entity.)

Shared library
See *Shared routine facility.*

Shared page
A page in your logical address space that more than one process can access. Shared pages are usually write-protected to prevent overwriting.

Shared routine facility
The facility whereby AOS/VS implicitly calls one or more library routines on disk into main memory areas in page increments; processes share these.

smallPID program
A program that cannot run if PIDs 1–255 are in use. SmallPID is the PID-size type of all programs before AOS/VS Revision 7.00 (except the CLI and EXEC, which were hybrid in Revision 6.00). The Link program creates programs of smallPID-size by default.

Source code
Code, consisting of byte-packed words of ASCII characters, which can be converted by an assembler or compiler into object code. Usually, you compose source code.

Source code file
A file that contains source code. Usually you use the CLI or a text editor under AOS/VS to create source code.

Spooling
A method of storing information on disk temporarily for later processing. AOS/VS uses spooling when processes (which run fast) have to use slow devices, like printers. (Spooling stands for Simultaneous Peripheral Operation On Line.)

Stack
A block of consecutive memory locations set aside for task-specific information. Every task that uses system calls must have a unique stack. (See also, *Wide stack.*)

Stack base
The starting address of a stack.

Stack fault handler
A routine that gains control when there is a stack error.

Station
The origin (sender) or destination (receiver) of data over a BSC line.

Swappable process
A process that is swapped into memory and written out to disk at the discretion of the scheduler. Swappable processes have the lowest priority of the three process types; they acquire memory only after the scheduler has satisfied all resident and pre-emptible processes.

Swapping
A procedure whereby AOS/VS writes a process out to disk, and then reassigns the main memory occupied by that process to another process that is waiting to run. This procedure is invisible to the process.

Switched line
A communications line on which you use a dialing procedure to establish a connection between local and remote stations.

System call
A request to the operating system to act on your behalf.

System generation
The process of tailoring AOS/VS to the particular hardware configuration and application environment at your installation.

Task
A path through a process. A task is an asynchronously controllable entity to which the processor is allocated for a specific time. A task can only execute code within the bounds of the address space allocated to its process.

Task call
See *System call.*

Task control block (TCB)
A block of data maintained by AOS/VS that contains a memory image of the processor registers and other context data for each task.

Task identifier (TID)
A user-specified number in the range from 1 through 32 that identifies a task within a particular process. (See also, *Unique task identifier (TID).*)

Task priority
Governs which is the executing task within a process. The executing task is always the highest priority task ready to run in the process with control of the central processor.

Task states
A task in a process exists in one of four states: dormant, ready, suspended, or executing.

TCB
See *Task control block (TCB).*

Template
Certain charaters to be matched, plus one or more expansion operator characters that allow specified parts of the template to accept any character as legal.

Tick
A real-time clock pulse.

TID
See *Task identifier (TID).*

Time-out value
The length of time AOS/VS will wait for a response from the target device before it takes an error return or begins error-recovery procedures. The shortest possible time-out value is 2 seconds.

Timesharing
A multiprogramming scheme in which processes share the processor on a timed basis; that is, a process takes control of the processor for a unit of time called a time slice. When this time slice expires, control goes to the next process that is waiting. Consequently, no process monopolizes the processor.

Trapping
Encountering a hardware fault.

Undedicated pages
Pages that AOS/VS can assign to a process as it requires them.

Unique task identifier (TID)
A system-assigned number that uniquely identifies each task, system-wide. (See also, *Task identifier (TID).*)

Unshared page
A page in your logical address space that only one process can access. Unshared pages cannot be write-protected.

Unused page
A page in your logical address space that is neither shared nor unshared. (See Chapter 3 for information on the relationships among shared, unshared, and unused pages.)

Variable-length record type
A record type whose records have a 4–byte ASCII header that specifies their byte length. Files that contains records of varying lengths have the variable-length record type.

Wide stack
A 32–bit stack. (See also, *Stack.*)

Wired pages
Pages that are permanently bound to the working set.

Word
A 32–bit (2–byte) location of memory.

Working directory
A process's reference point in the overall directory structure and its starting point for file access. Any directory can be a working directory, as long as you have proper access to it.

Working set
The subset of each process's logical address space that is memory resident. The working set of a process changes in size and content as the process references pages and then stops referencing them.

End of Glossary

Index

C

D

F

G

M

Q

R

S

T

U

X

Data General Corporation, Westboro, MA 01580

093-000335-01