VOS Reference Manual

Stratus Computer, Inc.

Notice

The information contained in this document is subject to change without notice.

UNLESS EXPRESSLY SET FORTH IN A WRITTEN AGREEMENT SIGNED BY AN AUTHORIZED REPRESENTATIVE OF STRATUS COMPUTER, INC., STRATUS MAKES NO WARRANTY OR REPRESENTATION OF ANY KIND WITH RESPECT TO THE INFORMATION CONTAINED HEREIN, INCLUDING WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PURPOSE. Stratus Computer, Inc., assumes no responsibility or obligation of any kind for any errors contained herein or in connection with the furnishing, performance, or use of this document.

Software described in Stratus documents (a) is the property of Stratus Computer, Inc., or the third party, (b) is furnished only under license, and (c) may be copied or used only as expressly permitted under the terms of the license.

Stratus manuals document all of the subroutines and commands of the user interface. Any other operating-system commands and subroutines are intended solely for use by Stratus personnel and are subject to change without warning.

Manual Name: *VOS Reference Manual*

Part Number: R002
Revision Number: 01
Printing Date: April 1990

Stratus Computer, Inc.
55 Fairbanks Blvd.
Marlboro, Massachusetts 01752

Preface

The Purpose of This Manual

The *VOS Reference Manual (R002)* documents the VOS operating system. It provides information on the operating system, processes, the file system, commands and command execution, programming, and networking.

Audience

This manual is intended for systems and application programmers. It is also useful for system administrators, operators, and (to a limited extent) advanced end-users.

Before using the *VOS Reference Manual (R002)*, you should be familiar with the following Stratus manuals.

- *Introduction to VOS (R001)*

Revision Information

This manual is a revision. For information on which release of the software this manual documents, see the Notice page.

This manual has been extensively revised.

Manual Organization

Chapter 1, "Hardware Overview," provides an overview of Stratus hardware. It also describes basic concepts such as fault tolerance and continuous processing.

Chapter 2, "The Operating System," is an overview of the operating system itself. It describes memory management, locking, process scheduling, transaction protection, and the I/O system.

Chapter 3, "Processes and Interprocess Communication," describes processes, process types, process execution, and process resources. It also describes methods of interprocess communication, including events, queues, pipe files, and shared virtual memory.

Chapter 4, "The VOS Command Language," describes commands and how they are executed. It explains the types of commands, command arguments, and library paths.

Chapter 5, "The File System," describes file and index formats, file I/O, directories, the cache manager, and disks.

Chapter 6, "Programs and Program Execution," describes how programs are written and executed, object and program module structure, stack frames and the procedure calling sequence, memory allocation, error handling, and debugging.

Chapter 7, "Networking Support," provides an overview of the StrataLINK local network and the StrataNET wide-area network.

Chapter 8, "Tape Processing," provides an overview of tapes and tape processing. It describes how tapes are used from command level and how they are used from within programs.

Notation

Stratus documentation uses *italics* to introduce or define new terms. For example:

> The action of removing one process from the executing state and placing another process into execution is called a *process switch*.

Computer font is used to represent text that would appear on your display screen or on a printer. (Such text is referred to as *literal* text.) For example:

> Use the `analyze_system dump_lock` request to display more information about wait locks.

Slanted font is used to represent general terms that are to be replaced by literal values. In the following example, the user must supply an actual value to replace the slanted font term.

> When the generalized user name is in the form `*.`*`group_name`*, the individual's user name must have the same group name.

Boldface is used to emphasize words within the text. For example:

> If more than one process is running the same program on a module, separate copies of the program do **not** have to be brought into memory.

Related Manuals

Refer to the following Stratus manuals for related documentation.

- *Introduction to VOS (R001)*
- *VOS Hardware Reference Manual (R008)*
- *VOS PL/I Language Manual (R009)*
- *VOS System Administrator's Guide (R012)*
- *VOS C Language Manual (R040)*
- *VOS Commands User's Guide (R089)*
- *VOS Commands Reference Manual (R098)*
- *System Configuration Manual (R099)*
- VOS Subroutines Manuals
- VOS Transaction Processing Facility Manuals

Online Documentation

You can find additional information by viewing the system's online documentation in `>system>doc`. The online documentation contains the latest information available, including updates and corrections to Stratus manuals.

A Note on the Contents of Stratus Manuals

Stratus manuals document all the subroutines and commands of the user interface. Any other commands and subroutines contained in the operating system are intended solely for use by Stratus personnel and are subject to change without warning.

How to Comment on This Manual

You can comment on this manual by using the command `comment_on_manual`, described in the *VOS System Administrator's Guide (R012)*. Type `comment_on_manual`, press RETURN, and then complete the form that appears on your screen. You must fill in this manual's part number, `R002`. When you have completed the form, press ENTER. Your comments are sent to Stratus over the Remote Service Network. Note that the operating system includes your name with your comments.

Stratus welcomes any corrections and suggestions for improving this manual.

Contents

Tables

Figures

Chapter 1:
Hardware Architecture Overview

This chapter provides an overview of Stratus hardware architecture. For more information, see the the *System Configuration Manual (R099)* and "Stratus Technical Report: The Stratus Architecture" (TR-1).

Fault Tolerance

The Stratus approach to fault tolerance has three basic elements:

- failure detection, so that hardware failures are detected when they occur.
- redundant hardware, so that failure of a single hardware component does not interrupt service to users. This is called *continuous processing*.
- recovery from the failure without (or with minimal) impact on users.

By providing hardware-based implementations of these functions, the Stratus approach has several advantages over software-based solutions. Hardware-based solutions are more robust. In addition, users need not take special steps or invoke special function calls to enable fault tolerance.

Failure Detection

Failure detection is accomplished by making system elements self-checking. This is accomplished in two ways.

- Each of the component circuit boards that make up the system has duplicated circuitry. Operations are performed in parallel and the results compared for consistency.
- All words written to memory contain error detection and correction information.

To provide failure detection, each board in the system has duplicate circuitry. For example, microprocessors on each CPU board are in pairs. This pair is referred to as a *logical microprocessor*. Both microprocessors execute the same instruction/data stream simultaneously, in lockstep (see Figure 1-1). On-board error-checking logic compares the output of both microprocessors at each machine cycle. An equal comparison implies that everything on the board is working correctly; the gate is enabled, and one of the outputs is sent to memory over the system bus. An unequal comparison indicates that something on the board is not operating correctly (one of the microprocessors, or some other component). This failure detection triggers the immediate removal of the board from service by disabling the gate, effectively disconnecting it from the bus.

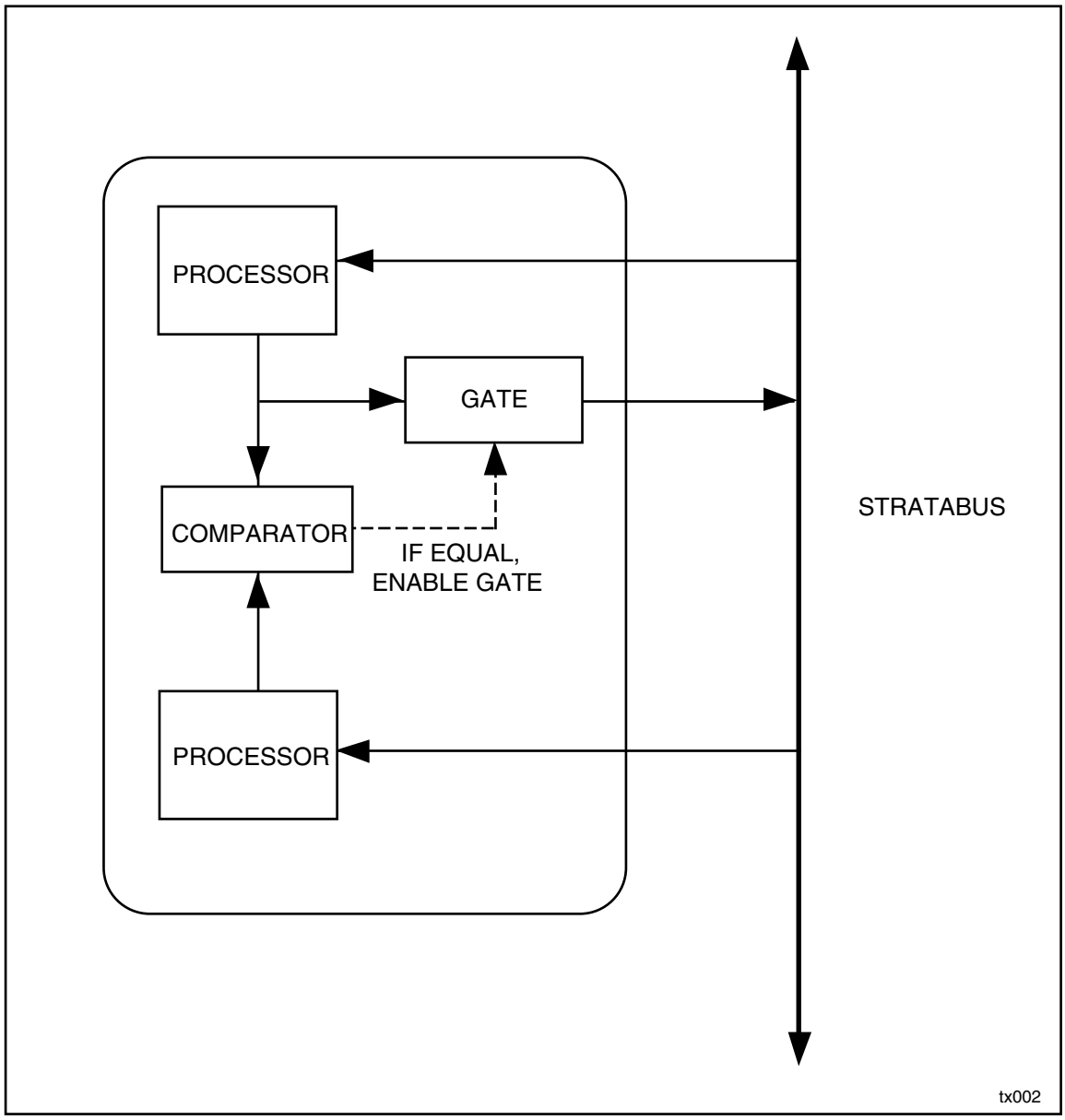

Figure 1-1. Processor Board

The memory controllers on each memory board are self-checking and duplexed. They can detect and correct single bit memory errors, and in case of uncorrectable errors, disconnect the segment of memory that they control. The controllers also monitor the buses for errors. They detect parity errors, purge bad data, and can shut down a bad bus by commanding all the controller boards to ignore that bus and feed both microprocessors on each board from the other one.

Continuous Processing

To assure that a single hardware failure does not interrupt system operation, each board in the system (with the exception of tape controller boards) is paired, with each board in the pair performing the same function in lock-step. For example, identical processor boards execute the same program concurrently. If one board fails and is removed from service, the remaining board continues operating normally, so the end-users and their programs are not affected (see Figure 1-2).

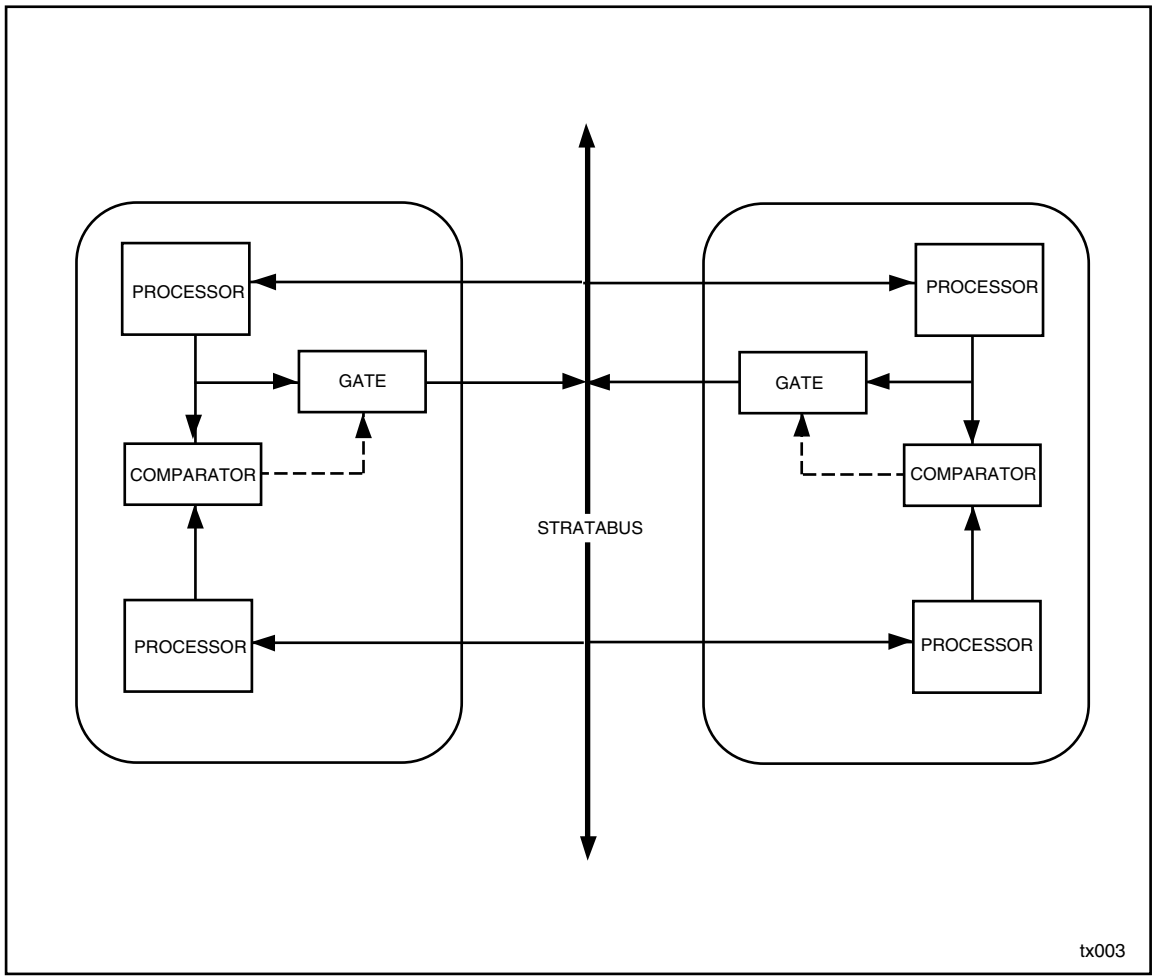

Figure 1-2. Duplexed Boards

This mechanism provides operational continuity even if a failure occurs. The failed board can be removed and replaced without affecting end users. All that is lost until the board is replaced is fault tolerance, not availability or performance. The probability of the simultaneous failure of both independent processors is very low.

In the case of processors, memory and memory controllers, and communication controllers, both board partners execute their functions in lock-step for operational continuity during a single component failure. The system bus, power supplies, and cooling fans are also duplexed, and the system can continue operation with only one of each.

Disk and StrataLINK controllers also have dual components and on-board compare logic, but the partner boards are independent of each other: they do not perform the same functions in lockstep. Synchronization between partner boards is provided by the operating system for the dual StrataLINK and disk controllers.

The disks themselves are duplexed. Information is written to both disks, and read from one. At bootload, the disks are checked for consistency and recovery takes place if necessary.

Failure Recovery

When one board in a duplexed pair fails, the operating system is notified of the failure by a special red light interrupt. The event is logged in the system for later reference, a message is displayed on the monitor terminal, and red lights are lit on the failed board and on the system control panel. The operating system initiates the execution of diagnostics on the now off-line board, compares the results with stored historical data, and uses an incidence-of-failure analysis to assess the situation. If the board fails the diagnostic tests, or if it has exceeded acceptable transient error thresholds, it is not returned to service.

When the board is removed from service, the system automatically dials the Stratus Customer Assistance Center (CAC) and enters a service call. These calls are queued up and handled by CAC technical support personnel, who investigate the problem further and can ship appropriate replacements for failed parts (with installation instructions) via overnight mail, if necessary The user replaces the part, and returns the failed one in the same package. This design is used for all of the main hardware components of the Stratus system (except the tape controller).

Components of a Stratus System

A Stratus System is made up of one or more modules. A module is a single computer and is the smallest hardware unit of a system capable of executing a user's process. Each module contains processor boards, memory boards, communications boards, disks, and, generally, tape and link boards. These components are shown in Figure 1-3 and are described briefly in the sections that follow. Multiple modules can be linked into a system using the StrataLINK local network. Systems can be connected by StrataNET wide-area network.

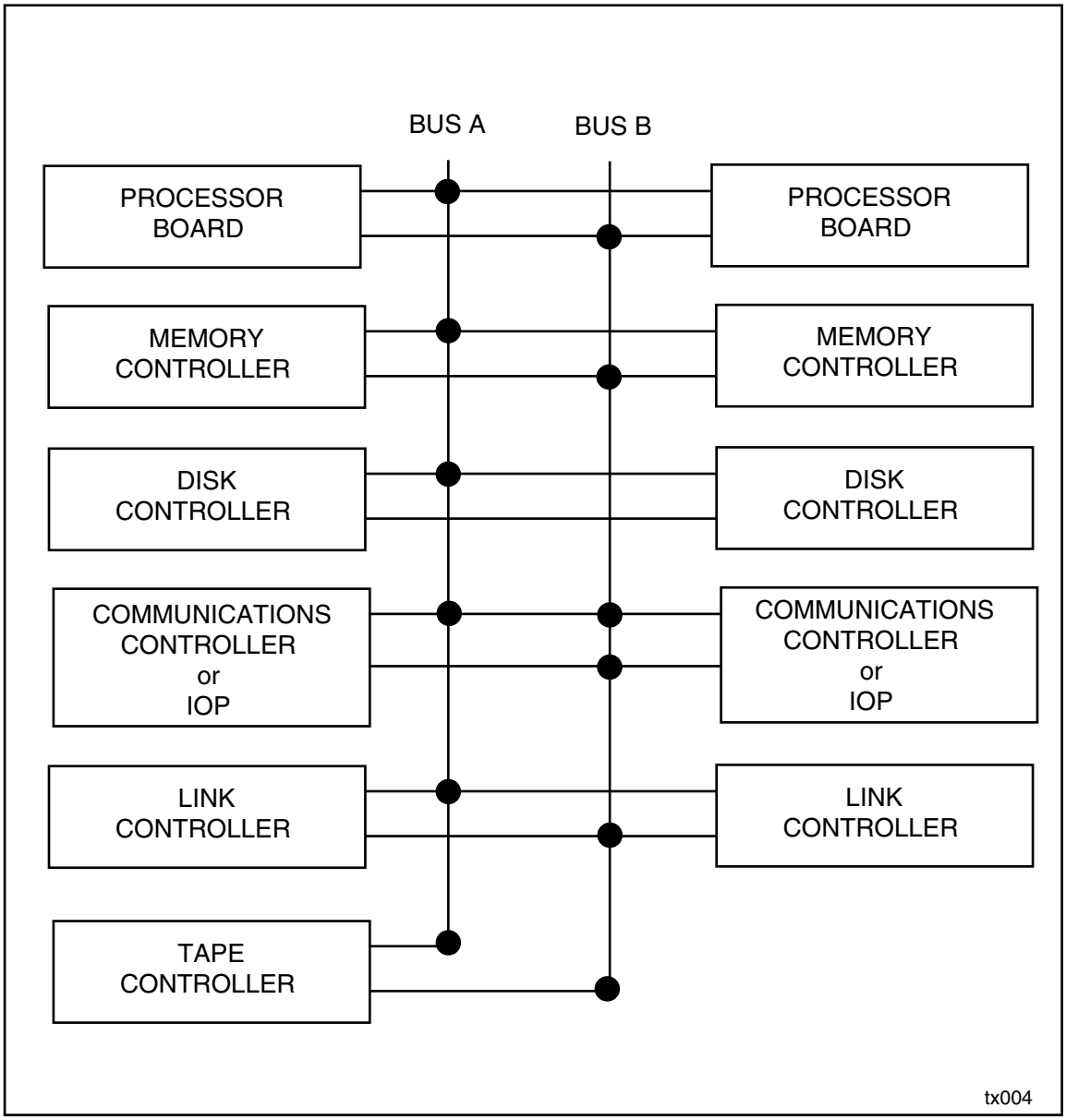

Figure 1-3. Components of a Stratus Module

CPU Boards

The CPU boards use one or more duplexed microprocessors, the number and type depending upon the model. The microprocessors belong to the Motorola 68000 family. Models in the FT line use the original 68000 processor. The XA400 and XA600 models use the 68010 microprocessor. Models in the XA2000-series use the 68020 or 68030 microprocessor, depending upon the model.

The CPU boards also contain cache (to speed up memory access) and address translation hardware. See the section "Virtual-to-Physical Address Translation" in Chapter 2 for more information.

Memory Boards

The memory controllers, which are self-checking and duplexed, monitor all traffic on the buses. They detect parity errors, purge bad data, and can shut down a bad bus by instructing all the controller boards to ignore that bus and feed both microprocessors on each board from the other one. (In full duplex mode, each microprocessor board is fed by a separate bus; see Figure 1-3 for a view of a processor board and its connection to the dual buses.) The memory controllers also manage memory access and refreshing; they provide four-way interleaved memory access (this allows overlapped access to four consecutive words in little more time than it takes to read just one word), detect and correct single bit memory errors, and in case of uncorrectable errors, disconnect the segment of memory that they control (4, 8, 16 or 32 MB, depending on the model). Operations continue on the partner board, and after replacement of the failed board, the controllers automatically bring up the pair to full duplexed operations, transparently to the end user and to application programs.

The operating system allows memory to be run in simplex mode, thus doubling the physical memory size. This processes is referred to as *unfolding* memory. This is convenient for testing and evaluating the effect of increased memory on application program performance, but makes memory a single point of failure, and therefore cannot be done during fault-tolerant operations. Unfolding memoryMemory, simplex

Disk Controllers and Disks

Several disk controller models are available. All are self checking and are configured in one or more pairs, where each pair can attach four or eight duplexed disk drives, depending on the disk model. The software-supported disk duplexing function requires at least one pair of disks of the same type on a pair of disk controllers. See Chapter 5 for more information on disks.

On modules using I/O processors, disk I/O can also be handled using an I/O processor and a disk adapter, rather than a disk controller.

Tape Controllers and Tape Drives

Up to four tape controllers, each supporting a single tape drive, can be configured on a single module. Each drive requires a self-checking, non-duplexed tape controller. On modules using I/O processors, tape I/O can also be handled using an I/O processor and a tape adapter, rather than a tape controller. In these configurations, streaming cartridge tapes are used as well as reel-to-reel tapes.

Communications Boards

Communications boards can be divided into two groups.

- Communications controllers and I/O processors
- Line adapters and I/O adapters

Communications Controllers

Communications controllers are used to transfer information between the operating system and the line adapters that communicate with the actual devices. The C200 is a self-checking board which contains its own microprocessor. Communications controllers are configured in one or more pairs. They execute the same instruction stream in lockstep, and are therefore fully redundant. Each communications controller scans its attached line adapters and transfers messages between them and operating system memory when required. Each pair of communications controllers attaches up to two communications chassis, and each chassis has slots for eight communications adapter cards.

Line Adapters

Line adapters provide the physical connection between the communications controller and the physical line attached to the device.

There are two basic types of line adapters: asynchronous and synchronous. The asynchronous adapters can be hard-wired or programmable, wired for modem or for local use; they all support two asynchronous lines each. The synchronous adapters are programmable, and support one line each. There are also specialized asynchronous adapters for line printer connection and for access to the Remote Service Network. Different types of adapters can be intermixed in the same communications chassis, as long as the maximum aggregate data rate of its controller pair is not exceeded; this is about 150-200 Kbps, and depends on the number and type of lines, message characteristics, distribution, etc. See *VOS Communications Software: Device Configuration Guide (R139)* for more information on the models of line adapters.

I/O Processors (IOPs)

Like the Communications Controllers, IOPs provide a communications path between the operating system and the adapters that communicate with the physical devices. However, the IOP provides a wider range of communications support. The I/O adapters used with it can communicate with disks and with tape drives in addition to being able to communicate with communications devices. (See the next section for more details.)

I/O Adapters (IOAs)

I/O adapters serve the same basic purpose as line adapters. In addition, I/O adapters can communicate with disks and tape drives. The communications adapters include adapters for both synchronous and asynchronous communications, ranging from terminal I/O to Ethernet and Token Ring local area networks. There are also user-programmable I/O adapters. See the *VOS Communications Software: Device Configuration Guide (R139)* for a more complete list.

Communications Devices

Asynchronous communications adapters support terminals and printers, allowing for either local or remote connection. Any kind of ASCII terminal can be used. Some types such as the V101 and V102 are already defined to the system. Others can be defined (see *VOS Communications Software: Defining a Terminal Type (R096)*). Personal Computers with asynchronous terminal emulation, or other asynchronous RS-232C character terminals can also be attached. Printers include daisy-wheel printers and high-speed line printers.

StrataLINK Local Network

The StrataLINK local network provides a 12.5 MHz (1.4 MBytes/sec) data path between multiple modules. The StrataLINK local network allows for horizontal growth: when the capacity of a single module is exceeded, additional modules can be added nondisruptively, without any impact on end users. The interconnected modules offer a single system image to the end user, who needs to do nothing different whether using a single or multimodule system. Fault tolerance is provided by having two separate links.

For more information on StrataLINK, see Chapter 7.

StrataNET Wide Area Network

The StrataNET wide-area network facility is used to connect systems. It uses the Stratus-to-Stratus (STS) interfaces of the Virtual Circuit Facility to enable transparent VOS system calls among multiple Stratus systems. Using StrataNET, the operating system can access remote resources (files, programs, devices) in a manner transparent to users. Thus, a user can simply request a resource by name (such as the path name of a file or program), and the operating system uses StrataNET to respond to those requests involving resources located on other systems.

For information about StrataNET, see Chapter 7 of this manual and *VOS Communications Software: X.25 and StrataNET Administration (R091).*

Chapter 2:
The Operating System

This chapter provides an overview of the VOS operating system.

Major Features of the Operating System

The VOS operating system is a multiuser, multiprogrammed, multiprocessor system. *Multiuser* means that multiple users can perform work simultaneously on the system. Multiprocessor means that a single computer can have more than one central processor, with different processes running on the different processors. *Multiprogrammed* means that users can be executing different programs. Each process has a single image of the operating system as part of its memory space (see Chapter 3). For systems made up of multiple modules, all modules can have knowledge of each others' peripherals and system resources.

The operating system is *procedure call based*. Major system functions are requested by procedure (subroutine) calls. (This contrasts, for example, with systems that are message-based, in which a process requests service by sending a message to another process.) Many of these procedures are also available to users as operating system subroutines.

The operating system is a *virtual memory system*. This means that a process can have a memory space larger than the physical memory of the module. (See Chapter 3 for information on a process's virtual memory space.) New pages are brought into main memory from disk when they are needed, while other pages are removed from memory, allowing programs to be bigger than available physical memory. This is referred to as *demand paging*. For more information on virtual memory management, see the section "Memory Management" below.

The *kernel* of the operating system, the part of the system that performs many of the most basic functions, is part of the address space of all processes running on a module. The VOS internal commands, which are executed in the operating system portion of the processes's virtual memory space, are also part of the address space of all processes on the module. The operating system also includes external commands, found in various system libraries, such as `(master_disk)>system>command_library`. These commands are executed in the user portion of a process's virtual address space.

The operating system manages a distributed, hierarchical file system. For more information, see Chapter 5. Access control is managed as part of the file system.

Process Management

A process has a virtual address space with separate areas for code, dynamic data, and static data. The operating system supports both interactive and noninteractive (batch) processes. Chapter 3 describes processes in detail.

The operating system creates processes, manages process resources, and destroys and cleans up terminated processes.

Operating System Processes

Various system processes perform important functions for the operating system. Several of these processes are described in the following sections.

The Overseer Process

The overseer process monitors the entire system. It waits on a series of events and responds to them when they occur. It response to the events, it performs the following actions:

- starting processes when a terminal connects to the system
- cleaning up terminated processes
- processing messages to the system error log
- broadcasting messages
- shutting down the module
- managing remote logins
- deactivating unused directories.

When any of these events occur, the Overseer process wakes up, performs the appropriate actions, then goes back to waiting. For example, when a process terminates, the Overseer cleans up that process.

The Command Processor

The command processor (also called the *Listener*), processes all commands that are issued by a process. It is responsible for reading the command line, expanding any abbreviations, passing any arguments to the command, processing any command functions on the command line, and beginning the process of loading external commands. Command line editing requests--deleting characters, moving to the beginning or of a line, and so forth--are handled by the asynchronous communications software, not the command processor. The command processor only receives the command after the user enters it.

The command processor is called by the operating system to process a single command line. After the command has been executed, control returns to the command processor, which in turn exits to the operating system. If the process still exists, the operating system calls the command processor again.

The TP Overseer Process

If transaction protection has been enabled on a module, the Transaction Processing Overseer (TP Overseer) process commits and aborts transactions. The TP Overseer also maintains a log of all transactions in their various states. It writes one record for each state of each transaction. Thus, for aborted transactions, the log contains a record of the states the transaction passed

through before the abort. This detailed log is used to restore files to a known state if it ever becomes necessary to do so. The TP Overseer performs the recovery process. (See the section "Transaction Protection" for more information.)

Only one TP Overseer process can run on a module.

Other System Processes

A number of other system processes control processing within the operating system. These include:

- the load control process. This process monitors system load by examining how busy the CPUs are. It then adjusts the batch queues based on this. The user can specify the meaning of busy in the load configuration table (`load_configuration.tin`) and the batch queues adjustment in the load control table (`load_control.tin`). For more information, see the *VOS System Administrator's Guide (R012)*.
- the batch overseer processes. Batch overseer processes manage a module's batch queues.
- spooler processes. Spooler processes manage a module's print queues.
- link server processes. For modules connected using StrataLINK communications, link server processes handle link traffic.
- processes related to communications products (such as X.25) or database products.

Process Switches

The operating system is a multiuser system. A number of processes can run at the same time. However, the number of processes that can physically be running at the exact same instant is limited by the number of CPUs on the system: only one process can be using any one processor at any one time. To enable more processes to run at the same time, active processes have to be switched on and off the CPU. The action of removing one process from the executing state and placing another process into execution is called a *process switch*.

A process switch can occur for several reasons.

- An executing process performs an operation for which a resource is currently unavailable or for which it has to wait. For example, if data must be read in from disk, a process switch occurs.
- An executing process performs an explicit wait.
- The time slice (see the next section) for an executing process expires.
- A process with a greater quantum type (see the next section) is present in the ready queue.

The scheduler process determines which process is to run, based on the process's quantum type. This is described in the next section.

The Scheduler

The scheduler schedules the execution of processes on a module. It must allocate processes on each CPU on the module.

As described in Chapter 3, a process can be in one of three basic states: executing, ready to execute, or waiting. Only one process can be in the execution state for each CPU. A process that is ready to run but not yet executing waits on the *ready queue*. The process of assigning processes to the execution state from the ready queue is called *scheduling*.

The scheduler is invoked in the following situations:

- An executing process waits, either explicitly or because a resource it needs is not currently available.
- An executing process performs a notify that moves a new process into the ready queue.
- A scheduler interrupt occurs. Scheduler interrupts occur on each CPU every 100 milliseconds.

When the scheduler is invoked, it can move processes from the ready queue into the executing state. This is described in the section "Process Switches."

The scheduler uses priority and time slices for process scheduling. This is described in the next section.

Priority Levels and Time Slices

Each process has a priority that is assigned to it when it is started. This priority is used to determine which processes will run next on the CPU. For interactive processes, the default priority is set as part of the user's registration information. For started or batch processes, you can specify the priority of the process as an argument to the `start_process` or `batch` command. (See the *VOS Commands User's Guide (R089)*.) Generally, the higher the value of the assigned priority, the more likely the process is to get CPU time (although this can be changed as described in the following paragraphs). User processes usually have a priority of 5. System process usually have a priority of 7, 8, or 9.

A *quantum* or *time slice* is an interval of CPU time that the scheduler allocates to a process. Quantums have attributes of type, count, and time. The quantum *type* defines its priority: 1 is the highest priority and 255 is the lowest. The *count* attribute determines the number of time slices of the corresponding type that a process can receive. The range is from 1 to 16. The *time* is the number of milliseconds that the scheduler allocates for each time slice of the corresponding type. Possible values are from 100 to 32,000 milliseconds.

The priority is an assigned set of quantum types, counts, and times. Priorities range from 0 to 9. Generally, the higher the priority, the lower the assigned quantum type and the longer the time slice. For example, interactive processes of priority five may have the following quantum attributes assigned:

Priority	Quantum Type	Count	Time
5	2	1	0.50
	4	5	1.00
	6	5	1.00
	10	1	0.50

This means that a process with priority 5 first runs with a quantum of type 2. This quantum is 0.5 milliseconds long. It can run under these parameters once. The next time this process runs, it has a quantum type of 4, with a time of 1.00 milliseconds. It can run at this level 5 times before dropping down to the next level, where the quantum type is 6, and so forth. A process returns to the top of the quantum type list when it waits for terminal input (waits on the event associated with port 5, the terminal port).

Note that while increasing priority (as defined here) generally results in increasing real priority, this does not have to be the case. A system administrator could, for example, assign the lowest quantum type (that is, the highest real priority) to priority 5 and assign higher quantum types (lower real priorities) to priorities 0 and 9. This is apt to cause confusion, however.

The quantum attributes associated with priority levels can displayed with the `display_scheduler_info` command and can be changed using the `set_scheduler_info` command.

Memory Management

Memory management includes the following functions:

- translating the virtual addresses supplied to the operating system by processes into physical addresses in main memory.
- keeping track of process memory, and maintaining the data structures needed to monitor all pages in every process's virtual address space.
- moving virtual pages into and out of physical memory.
- enabling sharing of code between multiple processes executing the same program.
- maintaining the various areas of memory needed by the operating system and by processes. This includes knowing which pages must always remain in memory and which can be paged out of memory, and allocating dynamic memory for processes.

Memory management is discussed in more detail in the sections that follow. All processors are connected and view the same logical memory space. In performing any of its functions, memory management must also coordinate the sharing of memory between multiple processors.

Virtual-to-Physical Address Translation

The memory addresses referenced by a process (for both code and data) are addresses in that process's virtual memory space. For the process to access this code or data, however, it must be in physical memory. When a process references an address, the virtual address must be translated into a physical address. (If the desired page is not in memory, it must be first brought into memory, as described in the section "Page Control.") Memory management maps virtual memory into physical memory. Because physical memory is used by multiple processors, the memory management software must also track which processors are using which areas of memory.

Physical or main memory is divided into 4096-byte pages (frames). Virtual memory pages are put in available main memory pages. Virtual to physical memory address translation is performed by hardware on all models except the FT.

Essentially, when the processor accesses an address, it supplies a virtual page number and the address within that page. Hardware on the processor board, using cache/tables on the board and tables stored in main memory, then determines if that page number corresponds to a physical page in memory. If so, it performs the virtual to physical translation. If not, a page fault occurs, and the page must be brought from disk into main memory.

Page Control

All memory paging is handled by the VOS page control software. Page control monitors all pages in physical memory that can be paged.

The operating system tracks physical memory using arrays of memory map entries (MMEs). Each page of physical memory has one MME associated with it. These MMEs are kept in two lists, which are maintained by page control.

- The *free list* contains all the physical memory pages that do not belong to any process.
- The *used list* contains all the currently in-use pages in physical memory.

When a process needs a new page, page control checks the free list. If it is empty, it then checks up to five pages on the used list for a page that it can use. If none are found, it rechecks the free list. It repeats this cycle until it finds a page it can use.

The page control software checks two bits when it is searching the used list: the *used bit*, which indicates that a page has been used recently, and the *modified bit*, which indicates that a page has been modified. If both the used bit and the modified bit for a page are off, it takes the page. When page control looks at a page, it turns the used bit off. If it finds a page with the used bit off and the modified bit on, it writes that page to disk. When a process uses a page, it turns the used bit on. Therefore, if the page is not touched by a process when page control again looks at it, the used bit will still be off, and page control can take the page if the modified bit is also off.

Not all page faults result in disk I/O. An allocate statement in a program results in a page fault. If page control finds a page on the free list, it satisfies the page fault without having to page anything to disk.

The code and symbol table regions of programs (see Chapter 6) are *pure*. That is, they cannot be modified by the program. Thus, if page control needs to take a page containing one of these regions, it need not write it to the paging partition of the disk. When the page is needed again, it is read from the program module in the file partition. (See Chapter 5 for information on disk partitions.)

Wired Memory

Wired memory is memory that is always in the same location in physical memory. It is not managed by page control. Wired pages are used for communications buffers, cache manager buffers, and certain parts of VOS that cannot be paged (such as the code that implements paging). A page can be added to wired memory by removing it from the used and free lists.

Sharing of Program Regions

If more than one process is running the same program on a module, separate copies of the program do **not** have to be brought into memory. Instead, the processes can share copies of the same program.

As described in the section "Page Control," the code and symbol table regions of programs cannot be modified by the program. Because of this, these areas of programs can be shared if more than one process runs the same program. Two different copies of these program regions are **not** brought into memory. The data (static and external static) regions of programs can be modified and thus cannot be shared, since each process can modify them in different ways.

The active page table tracks active virtual pages on the module. There is one active page table entry (APTE) for each active virtual page. The APTE contains the memory address of the page if it is in memory, or the disk address if it has been paged out of memory. If the page is not shared, the APTE includes a pointer to the process that is using the page. If the page is shared, the APTE includes a pointer to the active page table trailer, which manages memory usage for each program running on the module.

For more information on how the operating system loads programs, see the section "Program Execution" in this chapter. For more information on the structure of programs, see Chapter 6, "Programs and Program Execution."

Heap Management

Heaps are areas in the virtual address space where the system allocates operational and dynamic variables and system information including pages to be executed. Space in a heap must be allocated and freed dynamically, either using a language statement (such as `allocate` and `free` in PL/I and `malloc` and `free` in C) or using the VOS subroutines `s$allocate` and `s$free`.

The different kinds of heaps include the following:

- paged heap (kernel)
- wired heap (kernel)
- user heap (per process)
- PDR heap (per process) process data region
- process heap (per process)
- communications heap (per device).

The *paged heap* is in the VOS section of a process's virtual memory (see Chapter 3). It is used by the system to allocate pages of memory that can be paged in and out of memory.

The *wired heap* is also in the VOS section of a process's virtual memory. It is used by the system to allocate memory pages that **cannot** be paged out of memory, but must remain in physical memory.

The *user heap* is in the user's section of virtual memory. Memory allocated by a process is allocated in the user heap. Initially, it is eight pages. It is separated from the user stack by the *fence*. The fence is an area of memory that separates the stack from the heap. The fence prevents the stack from growing into the heap, or the heap from being allocated in stack space. If the user heap grows to the fence, subsequent allocations will cause a small process to grow into a big process (see Chapter 3).

The *PDR heap* (process data region heap) is used to store per-process information that is writable by the kernel and readable by the user.

The *process heap* does not resemble other heaps in structure or use. Control information for data in the process heap is stored in the PDR heap. It is used for I/O buffers such as fast file I/O, tape buffers, and so forth.

The *communications heap* is part of the wired heap, and is used for communications-related storage. Comm heapCommunications heap

Heaps are extensible in a noncontiguous fashion, and can thus grow dynamically. The operating system enables this by making it appear to the heap management software that the heaps are contiguous by creating dummy in-use blocks that form a bridge between the real in-use blocks.

Obtaining Information About Memory Management

The `analyze_system` facility can be used to obtain information about memory. Several of the requests you can use are listed below. Refer to the *VOS System Analysis Manual (R073)* for more information.

```
check_area
display_extensible_heap
display_memory_usage
dump_area
dump_eit
```

Program Execution

When a user invokes an internal command, no program loading is necessary since internal programs are part of each process's memory space.

When an external command is invoked, the command processor must call the routines needed to load the program into memory and begin execution. The exact steps followed depend upon whether the program is already in use and whether it is on the same module as the process invoking it.

The program loader first attaches a port to the program module. For a local (non-network) program, it then checks for an entry for the program in the executable image table (EIT) to see if the program is already active. The EIT maintains information about each program in execution on a module.

If the program is already active, the program loader connects the process to the program and resolves any references to external routines or data that were unresolved in the program module. Also, if the first page of the program is not currently in physical memory, it loads the first page into memory.

If the program is not active, the program loader creates an entry for it in the EIT. It then resolves unresolved references, and loads the first page of the program into memory.

If the program is on another module, a copy must be read across the network for each process invoking it. It cannot be shared by processes on the local module. (That is, if two processes on module 1 execute a program on module 2, two local copies of the program are made.) Thus only programs on the same module can be shared.

Operating System Locks

On a multiprocessor machine, two or more processes may need to modify the same data structure or file block at the same time. To prevent this from happening, things that can be altered by more than one process are protected by a *lock*. A lock is a software object (one or more bits), associated with a data structure or system resource that any process must check before accessing that resource. The process checks the state of the lock. If it is locked, the process cannot access the resource until whatever process that holds the lock unlocks it. If it is not locked, the process locks it, accesses the resource it needs, and unlocks the lock when it is complete. These locking operations are performed by the operating system, and are invisible to an application program.

Note that the system locks discussed here are different from the file locks discussed in Chapter 5.

When a process tries to get a lock but cannot, it can do several things depending upon the type of lock. There are three types of system locks:

- *Spin locks.* A process spinning for the lock holds the CPU and repeatedly tries to get the lock. This mechanism is usually used for locks that stay locked for short times, so that it is not worth the cost of a process switch to wait for the lock. Spin locks are also called *loop locks*.

- *Wait locks.* A process that waits for a lock releases the CPU and waits on an event. The process that is currently holding the lock notifies the event when it releases the lock. (See Chapter 3 for information on events.)

- *Spin/Wait locks.* If a process tries to get a spin/wait lock but cannot, it first spins for a short time, then waits on the lock.

Deadlocks are prevented by locking hierarchy conventions. That is, if a process must obtain more than one lock, it must get and release these locks in a prescribed order.

Use the `analyze_system dump_lock` request to display more information about wait locks. Use the `dump_et -loop_lock` request to display information about spin locks.

Transaction Protection

Transaction protection is the method which, when used, guarantees transactions to be either complete or not complete, but not be in some undefined middle state. The programming steps necessary to implement transaction protection are listed in Chapter 6 and described in detail in *VOS Transaction Processing Facility Guide (R215)*. This section provides a brief overview of the steps involved in transaction protection.

When processing a transaction, the TP Overseer obtains locks on files, records, keys, and the end-of-file, where appropriate. It performs lock arbitration to resolve lock contention among transactions.

In multimodule applications where the transaction originates on one module and the data files reside on other modules, the TP Overseer process must be running on all the modules involved. The TP Overseers on all the modules have a part in the commit process. The TP Overseer on the module where the transaction started coordinates the commit.

As the last step in the commit process, the TP Overseer uses the data in the log files to update the actual files.

The I/O System

The I/O system is the hardware and software that manages all the input, storage, and output of data. It moves data between the central processors and main memory of a processing module and the module's peripheral I/O devices or secondary storage.

The I/O system software is designed so that, from the applications viewpoint, reading and writing a disk file appear to be the same operations as reading and writing an I/O device. In particular, the rules for naming files are the same as the rules for naming I/O devices. The instructions to open or to close a file are the same as the instructions to initialize or to clear a device. Furthermore, the instructions to access data located on any module in the system, or on another system altogether, are the same as the instructions when the data is on your local module; the local and remote network operations are invisible to users. A program is able to read or write any file or I/O device in any processing module that the StrataLINK and StrataNET communications facilities can access.

When requesting any I/O operation, you need to specify only simple and general information about a file or I/O device. For example, you are not required to define the physical block size of a file on a disk, to allocate storage on the disk explicitly, or to initialize disk storage.

Objects

The I/O system manages several kinds of data structures and I/O devices. The term *object* is used for any data structure or device in the I/O system that you can refer to by name or some other identifier. The most important kinds of objects are files, file indexes, file records, directories, links, I/O devices (terminals, printers, tape drives, communications lines, and so forth), modules, systems, and I/O ports. Most objects have names. The next section presents the rules for forming names.

Names, Suffixes, and Star Names

A name is an ASCII character string that contains no more than 32 characters. The characters must be chosen from the following set of 81 characters:

- the upper-case letters
- the lower-case letters
- the decimal digits
- the ASCII national use characters:
  @ [\ ] ^ ‘ { | } ~
- the following additional ASCII characters:
  " $ + , - . / : _

The following ASCII characters are not allowed in names:

- the nonprinting (control) characters
- the SP (space) character
- the DEL character
- the remaining ASCII characters:
  ! # % & ‘ ( ) * ; < = > ?

In addition to these restrictions, a name must not begin with a hyphen, since a leading hyphen indicates a command argument.

A *suffix* is a character string beginning with a period. In a name, all the characters starting with a period and going to the end of the name make up a suffix. You can use suffixes to distinguish among several related objects. The names of files that the operating system creates often are formed from the name of a source file by replacing or adding a suffix. Note that the length of the suffix is only limited by the maximum length of the object name itself. This contrasts with file names under some operating systems where suffixes (called *extensions* by some other systems) are limited to 3 characters.

As noted, an asterisk is not permitted in an object name. The reason is that an asterisk in a name makes it a star name. A *star name* is a name that contains one or more asterisks or consists solely of an asterisk. A star name can be used to specify a set of objects. Star names function in the following manner:

- An asterisk can be in any position in a star name.

- In a path name, a star name can be in the final object name position only. (See Chapter 5, "The File System," for more information on path names.)
- When the operating system matches non-star names to a star name, each asterisk represents zero or more characters.
- No name can contain consecutive asterisks; there must always be an intervening character.

A star name matches an object name if the result of replacing the asterisks in the star name with definite strings is identical to the object name. For example, the star name `update_*.cobol` matches the object names `update_.cobol`, `update_accts.cobol`, `update_reports.cobol`, and `update_accts.old.cobol`. The strings replacing the asterisks are, respectively, the empty string, `accts`, `reports`, and `accts.old`.

Ports

An I/O *port* is an object that represents an I/O source or destination. You create and name an I/O port and associate it with an actual source or destination, such as a file or an I/O device. When you need to refer to the file or device for reading or writing data, you refer to the port by name or numeric identifier (port ID).

A port is created during a process and, unlike other I/O system objects, exists only for the lifetime of the process. Associating a port with a file or I/O device is called *attaching* the port; you attach a port with the `attach_port` command. The reverse operation is detaching the port; the command is `detach_port`.

When you attach a port with the `attach_port` command, you supply a port name as one of the arguments and the name of a file or I/O device as the other. When a process ends, VOS detaches all its ports.

A port that you explicitly attach with an `attach_port` command remains attached until you explicitly detach it with a `detach_port` command or until your process ends. This allows you to execute a sequence of programs with a common set of attachments.

Most of the time, it is unnecessary to use the `attach_port` and `detach_port` commands, because the programming language run-time support routines automatically attach a port when a program opens a file that is not already connected to a port. A port attached in this manner is automatically detached when the process accessing the attachment returns to command level, so there is no interference with the subsequent execution of other programs by the process.

Multiple ports from multiple processes can be attached to the same file. More than one port can be attached to some devices. For example, several ports can be attached to your terminal at the same time. However, some types of devices can only have one port at a time attached to them (tape drives, for example).

Every process has four ports that the operating system attaches when it creates the process. The names are `command_input`, `default_output`, `default_input`, and `terminal_output`. If the process is your log-in process, the operating system attaches all four ports to your terminal. If the process is created by a `start_process` command, then

VOS initially attaches `terminal_output` and `default_output` to the file or device you specify in the `start_process` command and leaves `command_input` and `default_input` unattached.

The maximum number of ports per process is 255.

Each of the programming languages has rules for associating the file names used in a program with I/O port names. You can write a program using the language's standard I/O statements and have the I/O directed to the file or device associated with a named port. Before running the program, you can associate the port with any file or device name consistent with the program. The language default rules for associating ports with files come into play. You can use the program in several different contexts.

Network and Intermodule I/O

Network I/O operations are I/O operations performed on devices and files that are in a Stratus system other than your system. The communication with another system is with the StrataNET wide-area network.

Intermodule I/O operations are I/O operations performed on devices and files that are in your system but in a processing module other than your module. The connection to another processing module is over the StrataLINK local network.

There is no restriction on the kinds of I/O operations you request over either a StrataNET connection or a StrataLINK connection. The capacity of the physical lines in StrataLINK is great enough that you usually do not notice any reduction in speed when processing data on another module in your system. The capacity of any public communications lines used by StrataNET, however, may slow down communication between systems.

Activity Logging

The operating system enables you to log all activity related to a port. For example, you may want to record in a file all the activity on your terminal or you may want to keep a list of the records that are accessed through a port that is attached to a particular file. Such record keeping is called activity logging.

A log is most commonly kept in a disk file, called a log file. But a log can be written to a tape drive, to a terminal, or to a line printer. The name of the command to start logging is `start_logging`. You give the name of the port to be logged and the path name of the log file or device when you issue the command. The port you will most often log is your own `default_output` port.

When the operating system writes a record to a log file, it includes a copy of the record that is read or written through the port. At your option, the log record can also contain the path name of the file or device that was the source or destination of the copied record, the record number or the index name and key value used to access the copied record, and a code for the I/O operation, such as sequential write, performed on the copied record.

A file or I/O device can concurrently be the log file of several ports defined in your process.

National Language Support

In operating system releases prior to release 6, all text data was assumed to be ASCII. The ASCII character set (defined in American National Standards Institute standard ANSI X3.4-1977) consists of characters represented by 7-bit values in the range 00x to 7Fx. The first 32 characters (00x to 1Fx) are called *control characters* and have no graphic representation; characters from 21x to 7Ex are called *graphic characters*. (Characters 20x (SP) and 7Fx (DEL) are not generally considered graphic characters.) All of the ASCII characters are referred to as *left* control characters and *left* graphic characters, because they occupy the left half of the matrix commonly used to associate character meaning with 8-bit hexadecimal representation (see Figure 2-1). When accessing characters in this matrix, read the column index first followed by the row index. For example, the value 20x falls in the left half of the matrix while A0x falls in the right half.

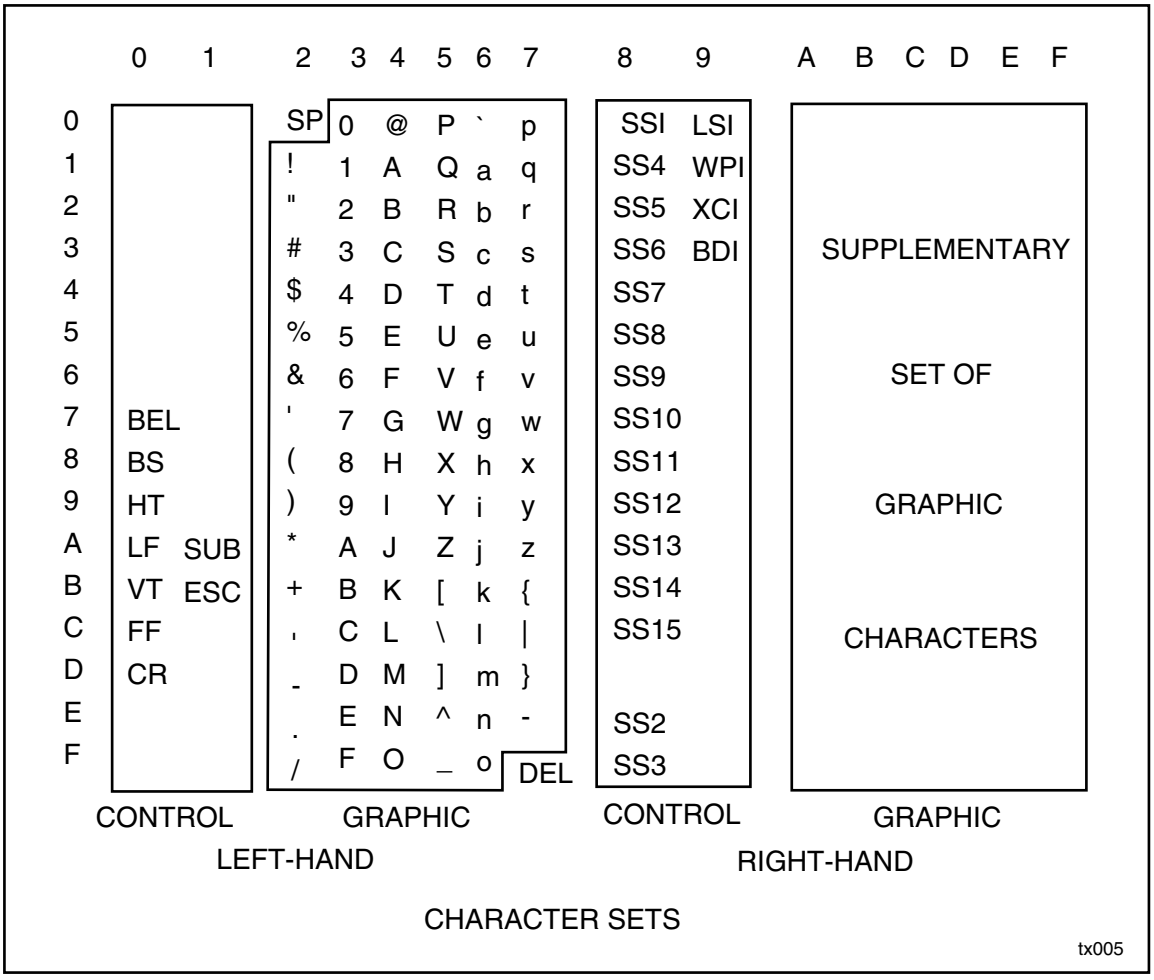

Figure 2-1. The Operating System Code Page

The operating system also supports other character sets by mapping these character sets into the right side of the table. Strings can contain characters from multiple character sets. This is

referred to as National Language Support (NLS). By default, the graphic representations for characters in the range A0 to FF are defined as Latin alphabet No. 1.

Other character sets including the kanji and katakana character sets are defined and supported. These character sets assign alternate meaning to the right graphic characters.

An NLS string is an ordered sequence of 8-bit bytes which represent a sequence of logical characters. Logical characters may occupy more than one byte. Thus, the characters in an NLS string may occupy more than one byte each. For the remainder of this chapter, *character* is used to refer to a one or two-byte character.

Right graphic characters can represent different character sets, including those requiring two bytes to represent single characters. *Shift characters,* analogous in conception and function to the shift key on a typewriter keyboard, are used to tell the operating system which character set is actually being used for a right graphics character. NLS shifts change the character set a byte belongs to. A text file can have only one default character set attribute; to change from the default to another character set in a text file it is necessary to embed shift characters in the file.

There are two types of shift characters:

- *Single shift characters* only affect the following character.
- *Locking shift characters* affect all characters until the next locking shift is encountered.

There are different shift characters for each character set supported. The compilers provide a built-in function which yields single and locking shift characters for any character set and an inverse function which yields the character set number for a particular single shift.

Canonical Form and Common Form

Using combinations of single shift and locking shift characters, there are a number of ways to represent the same sequence of international characters. One form of an NLS string is defined to be *canonical*. A canonical string is one in which every right graphic character is preceded by a single shift character.

There are many advantages in using a canonical string. An obvious disadvantage is that, in most cases, a canonical string requires the maximum number of bytes. If you use canonical strings, every character has a unique and thus unambiguous representation. Furthermore, a simple algorithm can be developed to examine international characters proceeding either forward or backwards from any point within such a string. On the other hand, canonical strings do not take advantage of locking shifts, and are thus generally longer than non-canonical strings. The compilers provide a built-in function, `shift`, which takes an NLS string as an argument and yields the equivalent canonical string.

The operating system supports text files containing characters from any supported international character set and the terminals to which these characters are input or output. When a program reads data from such a file or terminal, it will always be in NLS *common form*. This form is identical to canonical form except that all Latin alphabet No. 1 shift characters are omitted. If the string in question contains no characters from an international

character set other than Latin alphabet No. 1, then a common string is identical to a simple string.

If right graphic characters appear in a common string, they are assumed to be Latin alphabet No. 1 characters unless they are explicitly associated with another character set. This convention is followed when the application program communicates NLS strings to the file system. It is as though every string read from the terminal or a text file by the file system is preceded by an implicit locking shift to Latin alphabet No. 1. The same guarantee holds when the file system writes to a terminal or text file. If ambiguous right graphics are contained in the string being written, they are implicitly shifted to Latin alphabet No. 1. Any valid common or canonical NLS string may be written. The file system will diagnose an attempt to write an invalid NLS string or one containing locking shifts, and return an error code to the program trying to do so.

System Start-Up and Configuration

The operating system is self-initializing. No system generation phase is needed to control hardware or software configuration. Instead, the operating system surveys the hardware that is available at startup and configures what can be configured.

The operating system also surveys the disks to be sure they have the standard format and that data is consistent. If inconsistencies are found, it salvages the disk to correct the inconsistencies.

When a module goes through bootload, the operating system sets up several tables, then begins execution of the module startup macro (`module_start_up.cm`).

At first, the system is aware of only one communications device: the monitor terminal. When module startup executes the `configure_devices` command, the operating system reads the devices table, after which it can communicate with any configured device. Similarly, it executes other configuration commands such as `configure_modules`, to give it information about other modules in the system, and `configure_disks`, to give it information about other disks on the system.

Module startup also mounts disks, sets up the module's default library paths, starts the StrataLINK local network and the StrataNET wide-area network, configures communications protocols, loads any kernel-loadable programs or device drivers, and starts the TP Overseer.

For more information on the steps a module goes through during bootload, see *VOS System Administrator's Guide (R012)*.

Chapter 3:
Processes and Interprocess Communications

A *process* is a collection of system resources, including memory, processor access, and access to the operating system. A VOS process has virtual address space with separate areas for code, dynamic data, and static data.

There are two process types: interactive and batch. An *interactive process* is a process started with the prelogin `login` command. Any subsequent processes started with the `login` command are called *subprocesses*. A *batch process* is started with the `batch` command, the `start_process` command, or the `s$start_process` subroutine.

The operating system can theoretically support up to 255 processes on a module; the table for process IDs supports 255. However, in a normal mix of processes, the system is limited to a smaller number of processes. This process limit is a result of exhausting one of the fixed resources (for example, wired heap). Typically, between 150 and 200 is the maximum number found on a module.

Process Virtual Address Space

A process's virtual address space is divided between the user address space and VOS address space (see Figure 3-1). The first eight megabytes are the operating system's address space, and are shared by all processes. The next eight megabytes is process-specific. The size and address range depends upon the type of process. For a small *user process*, the two megabytes from address E00000 to FFFFFF (hexadecimal) are the user space, with the area between 800000 and E00000 available for heap expansion if the process grows. If this extra heap is used, the process is referred to as a *big process*. For a process with a 128-megabyte virtual address space, the user area is 64 megabytes. VOS has an additional 56 megabytes after this space. By default, the binder creates program modules with two megabytes of user space, and most user processes on they system are thus small processes. These processes can grow dynamically if they need more heap space. You can bind a bigger process using the `-size` argument of the `bind` command. See Chapter 6 of this manual and the *VOS Commands Reference Manual (R098)* for more information on `bind`.

Each process address space contains a separate stack and separate heap or heap area for each of these execution environments. Figure 3-2 show more details of the virtual address space for a process.

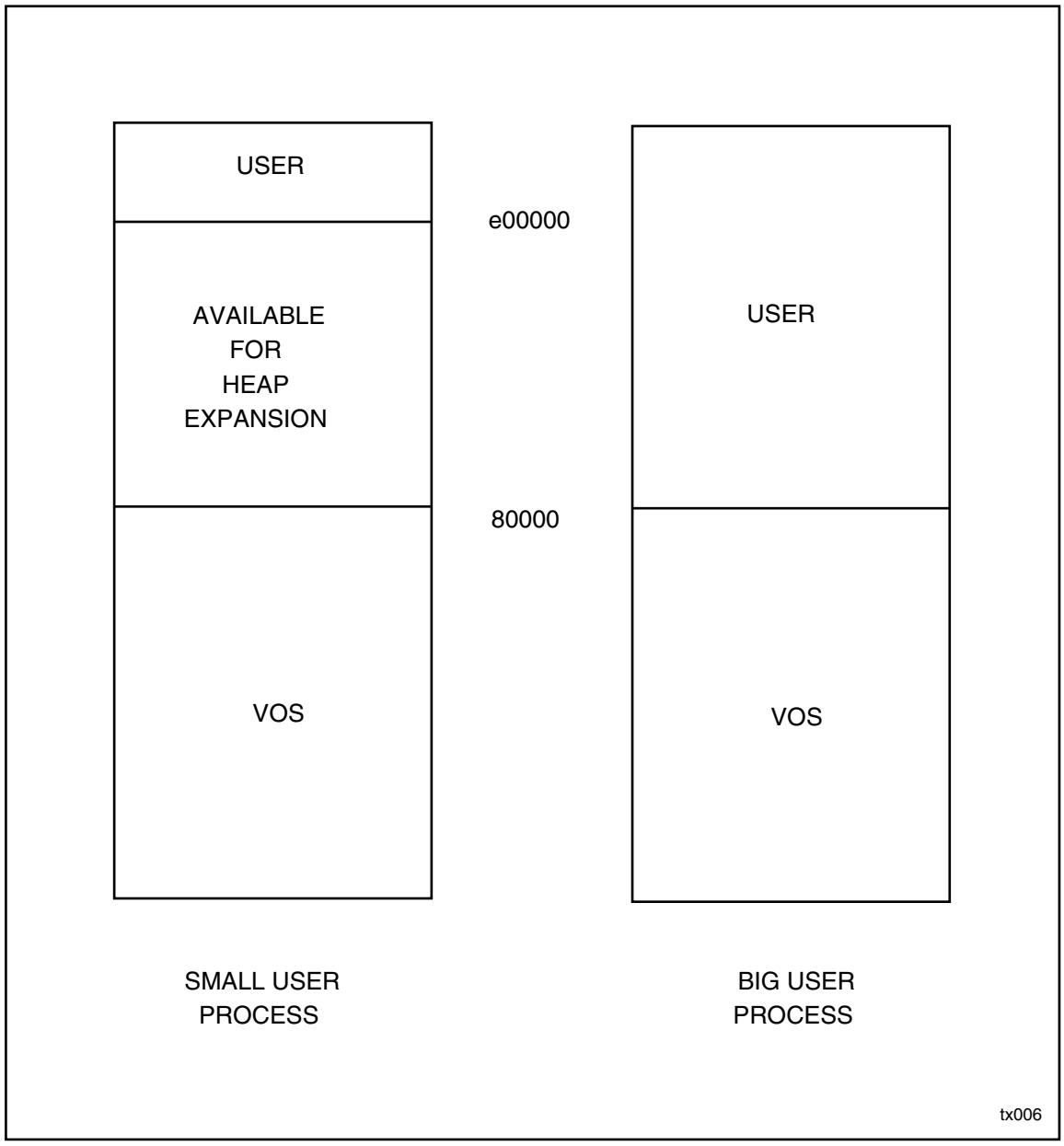

Figure 3-1. Virtual Address Space of a Process

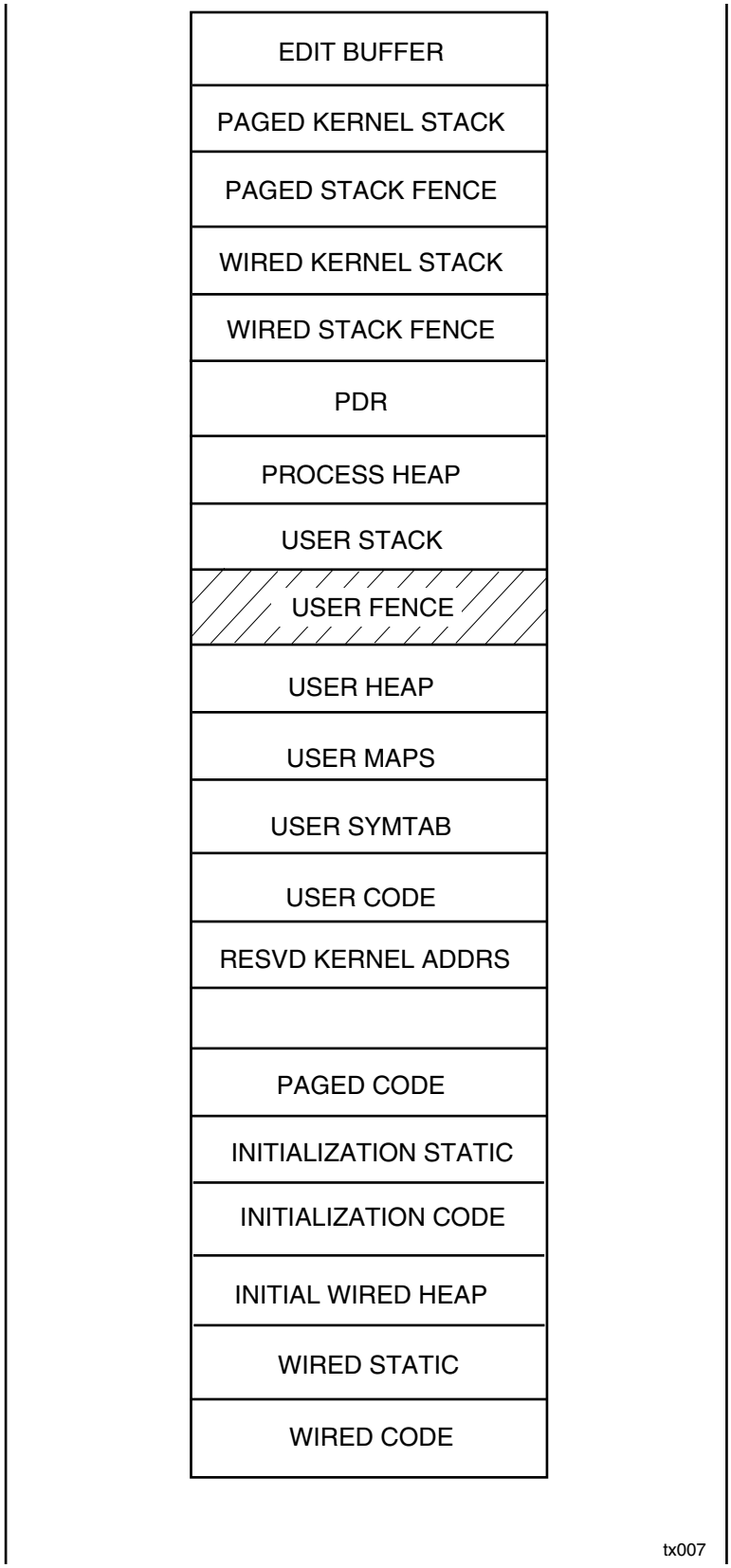

Figure 3-2. Detailed Virtual Address Space of a Process

Chapter 2 describes stacks and heaps.

The process data region (PDR) is described in the section "Per-Process Data Structures."

When a process has tasks within its space, the process stack and static area are divided between all of the tasks. See the section on tasking later in this chapter.

Per-Process Data Structures

Information about processes is kept in two main places: the process data region (PDR) and the process table entry (PTE). Information about the process that must be accessible by other processes is stored in the PTE. For example, scheduling parameters are stored in the PTE. Information about the process that is of primary interest to the process itself is stored in the PDR. The PDR contains pointers to ports and heaps, network client information, and information related to command processing.

Process Execution States and Environments

The operating system must prevent processes from writing to certain areas of memory and from executing certain powerful instructions. To do this, it uses the processor's two privilege states: user state and supervisor state.

In *user state*, memory accessibility is contingent upon access rights as specified in one of the memory management data structures. Execution of a specific subset of processor instructions is prohibited. For example, the process cannot access certain areas of memory used by the operating system and cannot execute certain critical instructions, such as the 68000 `halt` instruction.

In *supervisor state*, all instructions can be executed and all addressable memory locations are accessible. For information on which instructions can be issued only in supervisor state, see the *VOS Hardware Reference Manual (R008)*.

There is a third privilege state on the 68010 and 68020 processors called *superuser state*, in which the processor is in user state but the address translation hardware functions in supervisor state.

The only way to switch from user state to supervisor state is through the exception mechanism. When a process must switch from user state to supervisor state, the operating system executes a procedure called *kernel trap*. Kernel trap switches the process between user mode and supervisor mode.

Process Execution Environments

When a process is operating in user state, it is said to be executing in the *user environment*. If the process must execute a privileged instruction, it first executes a trap instruction and switches to the kernel environment. At this point, the processor is in supervisor state. This environment is also called the *paged kernel environment*, since the process can be paged out of memory.

If the process must not take a page fault (for example, if the process holds critical locks or if the process is itself executing the code that performs page faults), it executes in the *wired*

kernel environment. In this environment, like the paged kernel environment, the processor is in supervisor state and can execute all processor instructions. In addition, the process cannot be paged out of memory. A process moves from the paged kernel stack to the wired kernel stack by executing a wire and forward procedure.

The procedure uses a different stack in each of these environments. These stacks are called the user stack, the paged kernel stack, and the wired kernel stack (see Figure 3-3).

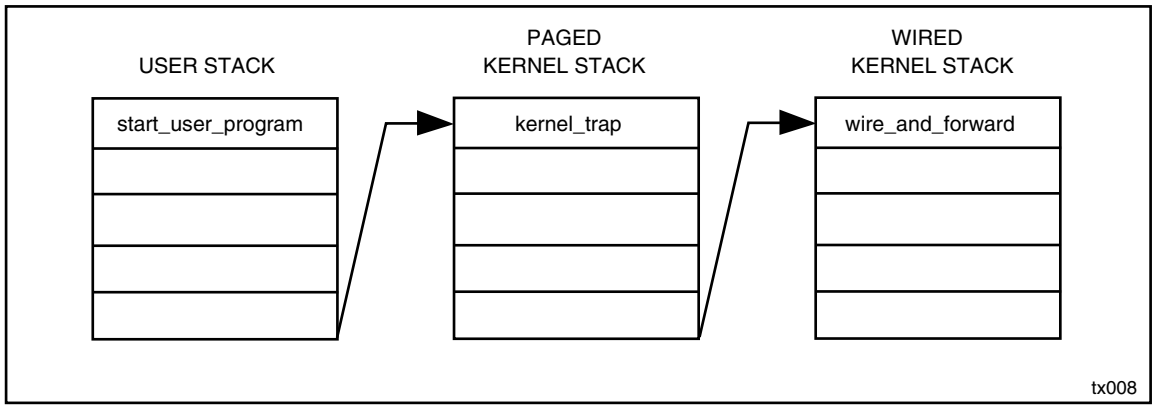

Figure 3-3. Process Operating Environments and Stacks

Faults and Interrupts

The process execution environment changes to the user fault environment when a hardware fault occurs, or to the interrupt environment when an external interrupt occurs. The hardware handles these two types of exceptions by suspending execution of the running procedure. It saves information about the environment (instruction counter, address and data registers, and so forth) in a set of machine conditions stored on the wired kernel stack, and then transfers control to an exception handler procedure.

Since a fault is caused by the process running on the CPU, the fault handler begins execution on the process's wired kernel stack. The fault handler may switch execution to the larger, per-CPU wired stack called the *interrupt stack* (6 pages long) if it needs more stack space than is available on the wired kernel stack.

External interrupts (such as disk I/O completion, communications interrupts, or level 7 interrupts) can occur at any time. They are triggered by other CPUs or by device controllers not associated with the running process. They are handled by any available CPU that is not running a process which has masked interrupts at the level of the incoming interrupt, or higher, or may be directed to a particular CPU.

Kernel interrupt handlers usually interrupt a running process to provide a service to some other process. Therefore, interrupt handlers are designed to minimize the length of interruption by performing short service actions. A new process is not switched in to service the interrupt. Instead, the interrupt is serviced by the process running on the CPU. The process is said to be running on *interrupt side*. Processes not on interrupt side are running on *call side*.

To minimize interrupt time and to avoid deadlocks among the shared data locks, external interrupt handlers may not reference paged stacks or other paged data structures. Such references could result in a page fault which delays handling of the interrupt.

When a process services an interrupt, it does not use its own kernel stacks. Instead, it uses the interrupt stack.

Process Priority

Process priority is established by levels 0-9, with 0 as the lowest and 9 as the highest priority. For each priority level in both types of processes, the operating system assigns values that define the attributes of the time slices that processes can receive. The operating system uses these values to compute the amount of processor time it will schedule for processes at each priority level. For more information, see Chapter 2, "The Operating System."

Depending upon its process type, a process follows one of two sets of scheduler parameters to determine how much time is allocated per execution. Interactive and sub processes have shorter quantums and, as a result, get to reschedule more often. The batch parameters, on the other hand, are longer and allow for more work to be performed before rescheduling. However, batch processes generally have higher quantum types (lower real priority) than interactive processes of the same priority level.

Process States

Processes can be in one of three states:

- ready
- wait
- stopped.

A process in the ready state is either executing or waiting in the ready queue to execute. A process in the wait state (also called short wait) is waiting for some action to occur; typically it is waiting for some I/O. A stopped process has terminated and is awaiting destruction by the operating system.

A number of other states applied to processes on the now-obsolete fault-tolerant series.

Process Resources and Features

Processes created by the prelogin process have the following resources and associated items after they are created:

- a home directory
- a current directory
- an active set of user-definable abbreviations for the command processor saved in the process's data region
- a set of attached ports

- a set of library search rules
- locked files and records
- client information for network access
- a program instruction counter value (address).

Subprocesses and batch processes inherit many of these resources from the parent process. For example, they have the same home directory, use the same start up macro (`start_up.cm`), and generally use the same abbreviations and library paths.

Home Directory

Every user process has a home directory. For an interactive login process, this becomes the current directory after login, until the user changes directories. For user subprocesses, started process, and batch processes, the home directory is the same as for the parent process. System processes can also have a home directory. The Overseer, for example, has the home directory `(master_disk)>Overseer`.

For user processes, the home directory holds certain housekeeping files. For instance, the start up macro and abbreviations file are in the home directory. Each time a user logs in or starts a process, VOS looks in the home directory for the start up macro. Generally, the name of the object name of the home directory is the same as the user name. (The full path name is generally `%`*`system_name>group_name>user_name`*.)

Current Directory

Working directoryCurrent directory A process's current directory acts as a default directory during interaction between it and the operating system. Many commands assume that, when you omit a directory path name argument from a command, you are referring to your current directory. Commands that create files, such as compilers, place the files in the current directory. Also, as explained in Chapter 5, the operating system uses the current directory to interpret relative path names.

You cannot change your home directory, but you can change your current directory with the command `change_current_dir`.

Abbreviations

Any process can have abbreviations assigned to it. Each command is processed to determine if it contains any abbreviations for expansion. Abbreviations are stored in the PDR heap (see Chapter 2).

For more information on abbreviations, see the *VOS Commands User's Guide (R089)*.

Port Attachments

Any login process by default has six default port attachments:

- `default_input` (1)
- `terminal_output` (2)
- `command_input` (3)
- `default_output` (4)
- `terminal` (5)
- *`unique_name`* (6).

Ports 1 through 5 are, by default, attached to the terminal, though some can be redirected. Port 6 is attached to the error codes text file for the process.

Any process can attach more ports, up to a maximum of 255.

For more information on ports, see Chapter 2.

Library Search Rules

Library search rules specify where a process is to search for command, include, object, and message files. Every process on a module uses the module's default library paths. A process can also add additional library paths. For more information on library paths, see Chapter 4.

Interprocess Communications

Two or more processes can communicate with one another in various ways:

- events
- queues
- pipe files
- virtual circuits
- shared virtual memory.

Each of these methods is outlined in the sections that follow.

Events

An *event* is a data structure, associated with a file or device, that is used by processes to communicate with one another. When one process notices that some action has occurred that is of potential interest to one or more other processes, the first process *notifies* the event. This notification causes the operating system to inform all processes that have been waiting for the action. (These processes are said to be *waiting on the event*.)

There are two types of events: system-defined events and user-defined events. A *system-defined event* (also called a *system event*) is used by the operating system to coordinate its interactions and by processes to coordinate use of system resources. Processes often use system events, for example, in using communications devices in no-wait mode. If an I/O operation cannot be completed, the process can do other work, then wait on the system event which is notified when the I/O can be completed. A *user-defined event* (*user event*) synchronizes activity between processes and can be used for primitive interprocess communication.

A maximum of 8160 events can be defined per module. By default, the maximum number of events per process is 16. This can be changed for the module (using `set_tuning_parameters`) to a maximum of 1500.

Events have associated with them an *event ID*, a four-byte integer that uniquely identifies that event, and an *event count*, a four-byte integer that indicates the number of times the event has been notified since it was created. User events also have an associated *event status*, another four-byte integer. The process notifying the event can use the event status to pass four bytes of data to the process waiting on the event.

Using User Events

A user event is associated with a file. This file can be empty. VOS uses the file to determine processes's access rights to the event. However, the event itself is otherwise not related to the file.

To wait on a user event, a process must follow these steps.

1. Attach the event to the process, using `s$attach_event`.
2. Read the event to determine the event counts using `s$read_event`.
3. Use `s$wait_event` to wait for the event to be notified (in which case the event count is incremented) or for a timeout to occur.

A process can also check the status of an event using `s$read_event`. When a process is done with the event, it detaches from the event by calling `s$detach_event`.

The process that notifies the user event must follow these steps.

1. Attach the event to the process using `s$attach_event`.
2. Notify the event using `s$notify_event`.
3. Detach from the event using `s$detach_event`.

The subroutine `s$notify_path` performs all three steps.

Using System Events

System events are associated with VOS actions. Typically, they are used in performing no-wait device I/O.

There is a standard method of using no-wait mode that you should use.

1. Call `s$set_no_wait_mode`. The application must supply the port ID (returned by `s$attach_port`. See Chapter 2 under the section "The I/O System" for more information on ports.) This subroutine places the port in no-wait mode and returns the event ID.
2. Call `s$read_event` or `s$read_device_event` to obtain the current event count.
3. Read or write to the port until the error code `e$caller_must_wait` (`1277`) is returned.

4. Continue processing or call `s$wait_event` using the event ID and the event count. This subroutine returns to the application when VOS notifies the event. It returns the updated event count.

Queues

A *queue* is a special kind of file that one process can use to communicate with another process. One process or task (see "Multitasking Processes," later in this chapter) puts a message into a queue and another takes it out. Since queues are implemented using the file system, any process using a queue to communicate with another process must attach a port to the queue and refer to the queue by the port identifier.

A process (or task) that sends a message to the queue is called a *requester.* A process (or task) that reads a message from the queue is called a *server.*

A server can access a message in the queue by *receiving* the message or by *reading* the message. When a server receives the message, it changes the message's status to busy, preventing other servers from receiving it. When a server reads a message, it does not change the status of the message.

Access to the queue is like access to a file. If you have write access to the queue, you can issue any permitted call on any message in the queue. If you have read access, you can read any message in the queue but receive only those messages with your person name (since receiving a message removes it from the queue). If you have execute access, you can read and receive only messages with your person name.

One-Way and Two-Way Queues

A *one-way queue* handles messages, but it does not handle replies to messages. The program sending the messages does not expect or wait for replies. Requester programs send messages to the queue. Server programs receive messages from the queue.

A *two-way queue* handles messages and replies to those messages. Requester programs send messages to the queue and expect replies. Server programs receive a message from the queue and then send a reply to a message that it has received. The reply replaces the message in the queue. The requester program that sent the original message is the only requester that can receive the reply.

Queue Structure

A queue has an established file structure. Each message unit in a queue consists of a header and a message.

The *message* is the character string sent by the requester program. The sending program defines the message length and message format through the variables required by the `s$msg_send` subroutine. On two-way queues, the reply replaces the original message in the queue.

The *header* contains information such as the message priority, the message ID, the process IDs of both the requester and the server programs, time and date data, and various switches. The header format is predefined. Most information in the header is provided by the operating system. Programs can obtain header information by using the `s$msg_read` and `s$msg_receive` subroutines.

The header is created when the message is sent. On a one-way queue, the header is deleted when the message is deleted. On two-way queues, some fields in the header are updated when the reply is sent (when the reply replaces the message). When the reply is received and deleted from the queue, the header is also deleted.

Queue Types

The operating system supports three queue types.

- Message queues. These queues are the least restrictive in terms of the actions that can be performed on a queue. Message queues are always one-way queues. Both the header and the message reside on disk. Message queues are the slowest type of queue.

- One-way and two-way server queues. These queues are more restrictive than message queues in terms of the actions that can be performed on a queue. The message header resides in the paged heap portion of memory, and the message text resides in disk cache. Server queues are faster than message queues but slower than direct queues.

- One-way and two-way direct queues. These queues are the fastest type of queue, but also the most restrictive in terms of the actions that can be performed on a queue. Both the header and the message reside in wired memory. Each requester port attached to the queue can handle only one message at a time.

Table 3-1 summarizes the major characteristics of the various queue types supported by the operating system. Each type is summarized later in this chapter and discussed in detail in *VOS Transaction Processing Facility Guide (R215)*.

Table 3-1. Major Features of Queues *(Page 1 of 2)*

Queue Type	Features
Message	Transaction protection is allowed on the queue. Messages remain in the queue until explicitly deleted. Replies are not handled. This is the most flexible queue type in terms of operations permitted by requester and server programs. The maximum message length for a queue on the same module as the calling process is 2^{23} bytes; for a queue on a different module, 2^{20} bytes.
Two-Way Server	Transaction protection is transmitted by the queue. A message is replaced by a reply; the reply and message header are deleted when the reply is received. If a requester closes a queue, all messages entered by that requester and all replies to that requester are deleted from the queue, regardless of the message status. Messages sent by other requesters remain unaffected. A reply is required. This queue type is faster than a message queue. The sender can wait or cancel if no receiver is available. The maximum message length for a queue on the same module as the calling process is 2^{23} bytes; for a queue on a different module, 2^{20} bytes. The maximum number of messages per module depends on the module's memory capacity and the maximum queue depth established for the queue.

Table 3-1. Major Features of Queues *(Page 2 of 2)*

Queue Type	Features
One-Way Server	No transaction protection is offered. Messages are deleted when they are received. Messages entered by a requester are deleted if all requesters and servers with ports attached to the queue close the queue. Replies are not handled. This queue type is faster than a message queue or a two-way server queue. The maximum message length for a queue on the same module as the calling process is 2^{23} bytes; for a queue on a different module, 2^{20} bytes. The maximum number of messages depends on the module's memory capacity and the maximum queue depth established for the queue.
Two-Way Direct	No transaction protection is offered. A reply is required. This queue type is faster than a two-way server queue. The queue and all of its server programs must reside on the same module. A message is replaced by a reply; the reply and the header are deleted when the reply is received. If no server program is active, a message cannot be sent. The maximum message length is 3,072 bytes (or less, if so specified when the queue is opened). The maximum number of messages is one message for each port attached to the queue. A requester cannot send a second message before receiving a reply to the previous message.
One-Way Direct	No transaction protection is offered. Replies are not handled. This queue type is the fastest queue type. The queue and all of its server programs must reside on the same module. A message is deleted when it is received. If no server program is active, a message cannot be sent. The maximum message length is 3,072 bytes (or less, if so specified when the queue is opened). The maximum number of messages is one message for each port attached to the queue. The sender cannot send a second message until a server has received the previous message.

Creating Queues

To create a queue, use the command `create_file` or the subroutine `s$create_file` (described, respectively, in the *VOS Commands Reference Manual (R098)* and the VOS Subroutines Manuals). The second argument of both the command and the subroutine refers to the organization of the file. If you are creating a direct queue (one-way or two-way), create a **sequential** file. Other options include `one_way_server_queue`, `server_queue`, and `message_queue`, which are used to create the other types of queues.

Queue Related Subroutines

To use a queue, a process must first attach a port to the queue (using `s$attach_port`). It must then open the queue, using either the subroutine `s$msg_open` (for message and server queues) or `s$msg_open_direct` (for direct queues). One of the arguments for both of those subroutines is `io_type`, which specifies whether the opener is a requester or server and whether it is one-way or two-way.

Processes send messages to the queue and receive messages from the queue using system subroutines. Most subroutines that are used with queues have names beginning with `s$msg_`. Which subroutines are used depends upon whether the process (task) is a requester or a server and upon what kind of queue is being used. Table 3-2 lists which subroutines are used to access which kinds of queues. For more information on any of these subroutines, see *VOS Transaction Processing Facility Guide (R215)*.

Table 3-2. Subroutines Valid for Each Queue Type

Queue Type	Valid Requester Subroutines	Valid Server Subroutines

Message Queue	`s$msg_open`
	`s$msg_read`
	`s$msg_send`
	`s$msg_receive`
	`s$msg_rewrite`
	`s$msg_cancel_receive`
	`s$msg_delete`
		`s$msg_open`
		`s$msg_read`
		`s$msg_receive`
		`s$msg_rewrite`
		`s$msg_cancel_receive`
		`s$msg_delete`
		`s$truncate_queue`

Two-Way Server	`s$msg_open`
	`s$msg_read`
	`s$msg_send`
	`s$msg_receive_reply`
	`s$call_server`
		`s$msg_open`
		`s$msg_read`
		`s$msg_receive`
		`s$msg_send_reply`
		`s$msg_cancel_receive`

One-Way Server	`s$msg_open`
	`s$msg_read`
	`s$msg_send`
		`s$msg_open`
		`s$msg_read`
		`s$msg_receive`

Two-Way Direct	`s$call_server`
	`s$msg_open_direct`
	`s$msg_receive_reply`
	`s$msg_send`
		`s$msg_open_direct`
		`s$msg_receive`
		`s$msg_send_reply`

One-Way Direct	`s$msg_open_direct` `s$msg_send`	`s$msg_open_direct` `s$msg_receive`

Message Priorities

When you enter a request or message into a queue, you must assign it a priority. The priority must be an integer between 0 and 19, with 19 being the highest priority.

A server, when it is ready to service a message in a queue, takes the message with the highest priority. Among the messages with equal priority, the messages are serviced on a first-come first-served basis.

For server queues, priorities 0 through 9 mean the requester (sender) will wait (up to some maximum length of time) until the message is received by a server. Priorities 10 through 19 mean the requester will give up sending the message if a server is not immediately available to receive it. A higher number gives the message a higher priority; a message with priority 16 will have precedence over a message with priority 7.

For message queues and direct queues, the priorities simply go from 0 to 19. A higher number means a higher priority.

Message Queues

Message queues are always one-way queues. Requester programs can send messages to and receive messages from this type of queue. Requester and server programs can receive and rewrite (essentially resend) messages. Therefore, message queues can pass messages among requester programs, among server programs, or among requesters and servers.

For a message queue, the maximum message length is 2^{23} bytes when the queue and the calling process are located on the same module, and 2^{20} bytes when the queue and the calling process are located on different modules. There are no limitations on the number of messages in the queue except for the amount of space available on the disk where the queue resides.

In a message queue, both the header and the message reside on disk. Subsequent access to a message requires disk access, making this the slowest queue type. The disk storage method also makes message queues act more like normal data files than the other queue types. Messages remain in the queue until they are explicitly deleted with the `s$msg_delete` subroutine. All messages remain in a message queue even if one or all of the programs using the queue stop, or if one or all of the ports attached to the queue are closed.

Message queues allow you to use transaction protection on the contents of the queue. If you call `s$start_transaction` and then enter a message into a message queue, the message will not become available to another process until the transaction is committed. If the transaction is aborted, the message will be deleted from the queue. Similarly, if you call `s$msg_receive` after `s$start_transaction`, the message in the queue will not be disturbed (except to be unavailable to other users) until the transaction is committed. If the transaction is aborted, it will be as if the message was never received.

Server Queues

Server queue message headers are stored in the paged heap portion of memory. (Paged heap refers to an area of shared virtual memory on a module.) The message text is stored in disk cache. Depending on resource availability, the text in the cache might be written to disk. This storage arrangement makes server queue I/O faster than message queue I/O, but slower than direct queue I/O.

The maximum message length for a server queue is 2^{23} bytes when the queue and the calling process are located on the same module, and 2^{20} bytes when the queue and the calling process are located on different modules.

When **all** ports attached to a one-way server queue are closed (that is, all requester and all server ports), all messages are deleted from the queue, regardless of the message status. Otherwise, if some ports are closed but at least one port remains open, messages in the queue are not affected.

When a requester program closes a port attached to a two-way server queue, all messages that were sent on the closed port are deleted from the queue. Also, all replies to messages that were sent on the closed port are deleted from the queue.

When a server program closes a port attached to a two-way server queue, there is no effect on messages that the server has not touched. There is also no effect on messages that the server has received and replied to. However, messages that were received but not replied to before the server port was closed are deleted.

Transaction protection is supported for two-way server queues. The requester calls `s$start_transaction` before calling `s$call_server` to send the message. It can then call `s$commit_transaction`. The server receives the transaction ID from the requester and returns it with `s$msg_send_reply`. The server must always check for the error code `e$tp_aborted`. If it is received, it sends a message to the requester to abort the transaction. The requester must use `s$call_server` to avoid more than one process using the same transaction ID.

One-way server queues do not support transaction protection.

Direct Queues

Direct queues can be one-way or two-way queues. Both the header and the message portions of one-way and two-way direct queues reside in wired memory --- nothing ever goes to disk. This storage method makes direct queues the fastest type of queue, but not the most efficient queue in terms of memory usage. However, memory for direct queues is allocated using a shared buffer scheme, which reduces memory usage to a manageable level for most applications requiring the speed advantages of direct queues.

When a direct queue is used in multimodule applications, the queue must reside on the **same** module on which all of the server processes are executing. The requester processes can be remote.

One message exists for any port attached to a direct queue. When this message is completely serviced, it is deleted from the queue, and another message can be sent on the port. For a one-way direct queue, a message sent on a requester port must be received by a server before the requester can send another message on the same port. For a two-way direct queue, the message must be received by a server, and the reply must be sent by the server and received by the requester before the requester can send another message through the same port.

With direct queues, you have the option of omitting the header portion of messages. For a specific queue, either all of the messages must have headers, or no messages must have headers. The option is specified when the queue's server processes open ports to the queue.

The maximum length of the queue's messages is determined when the queue is opened. The value must be between 0 and 3,072, inclusive. All ports opening the same queue must specify the same maximum message size.

Transaction protection is **not** available on direct queues.

Pipe Files

A *pipe file* is a file which is used to transfer data from one process to another. One process, called a *producer*, sends data to the pipe file while another process, called a *consumer*, reads data from the pipe file. These reads and writes are done the same as ordinary sequential file I/O (using `s$seq_read` and `s$seq_write` for example). Data written to a pipe remains in the pipe only until it is read. Once read, it is discarded. There can be more than one producer process and more than one consumer process.

The pipe file attribute can be associated with any file type. A pipe file is designated by using the `set_pipe_file` command or the `s$set_pipe_file` subroutine. The pipe file attribute can be turned on but there is no way to turn it off. Once a file is a pipe file, it must remain a pipe file. The pipe file attribute may be put on any empty file with the type fixed, relative, sequential or stream, that does not have any indexes.

The only valid access mode for a pipe file is `sequential`. The valid I/O types are `input`, `output`, `update`, and `append`. The locking mode can be `set_lock_dont_wait`, `implicit`, `dont_set_lock`, and `wait_for_lock`.

All sequential I/O system subroutines, except for `s$seq_position`, that normally work on a file of the pipe's base type (fixed, relative, sequential, and stream) will work on the pipe file.

A pipe file is full when it contains 64K or more bytes. If it becomes full, subsequent write operations wait until read operations reduce the size of the file to less than about 32K bytes. An application can read data from a pipe until it is empty (all data consumed). It then waits until more data is written. A partial record cannot be read, except by the process that wrote it, until it has been ended by a non-partial write.

A pipe can be used in wait mode (the call will not return until the data is read or written) or in no-wait mode (the call will return `e$caller_must_wait` if the data cannot be read or written yet). A pipe, therefore, behaves like a device with regard to wait and no-wait modes.

Data exists in a pipe only while some process has the pipe open. All the data disappears when the last process closes the pipe. If a pipe is not opened for writing by at least one of its openers, then readers get `e$end_of_file` returned after all the current data in the pipe is consumed. The maximum size of a pipe file is 17 blocks.

Typical use of the pipe file might be a batch environment where the output of one process is needed as input to another. Instead of waiting until the first process has completed to start the second, as each record is produced by the first process, pipes can make it immediately available to the second.

Virtual Circuits

A *virtual circuit* is a bidirectional communications path that provides for full-duplex data transmission independent of a fixed physical circuit. The VOS Virtual Circuit Facility provides the user interface to virtual circuits and to X.25. It includes the virtual circuit subroutines that allow users (application programs) to establish X.25 and/or VOS virtual circuits on a per-call basis.

For virtual circuit operations on a single system, neither X.25 packets nor X.25 channels are used. VOS uses logical packets, which are the I/O buffers used by the process. It simply moves the buffer contents from one process to the other when communications are on the same module. Therefore, virtual circuit operations on a single module can be very fast.

The Virtual Circuit Facility supports process-to-process (program-to-program) addressing by use of extensions. VOS uses extensions to identify an individual process (or a particular service) that is the target of a connection request.

The operating system provides this extension addressing feature for Stratus-to-Stratus connections. Each Stratus module contains a set of numbered *extensions*, ranging from 0 through 255. A process that needs to receive a connection request informs the Virtual Circuit Facility that it wishes to listen to an extension number on its module. A process that needs to issue a connection request indicates the module and extension number that it needs to call.

There are three phases in using virtual circuits.

- Call Request Phase. Both the calling and called processes must initiate the virtual connect.
- Data Transfer Phase. The calling process transmits data to the called process.
- Call Clearing Phase. The calling process signals that transmission is over and the virtual circuit is cleared by both processes.

The virtual circuit subroutines used for interprocess communication are listed in Table 3-3 and documented in *VOS Communications Software: X.25 and X.29 Programming (R028)*.

Table 3-3. Virtual Circuit Subroutines

Group	Subroutine	Purpose
Call Setup Subroutines	`s$vc_call_sts`	Issue a Stratus call to an extension.
	`s$vc_call_full_sts`	Issue a Stratus call with facilities and data to an extension.
	`s$vc_find_call_sts`	Find an incoming Stratus call to an extension.
	`s$vc_find_call_full_sts`	Find an incoming Stratus call to an extension and return call data/facilities.
	`s$vc_accept`	Accept an incoming call.
	`s$vc_accept_full`	Accept an incoming call, specifying call-accepted facilities and data.
Data Transfer Subroutines	`s$vc_send_packet`	Send a packet on a virtual circuit.
	`s$vc_recv_packet`	Receive a packet from a virtual circuit.
	`s$vc_send_interrupt`	Send an interrupt packet on a virtual circuit.
	`s$vc_recv_interrupt`	Receive an interrupt packet from a virtual circuit.
	`s$vc_confirm_interrupt`	Confirm a previously received interrupt packet.
	`s$vc_reset`	Reset or confirm a reset of a virtual circuit.
	`s$vc_status`	Check the status of a virtual circuit.
Call Clearing Subroutines	`s$vc_clear`	Reject an incoming call or terminate a connection.
	`s$vc_clear_full`	Reject an incoming call or terminate a connection, specifying call-clearing data.
Utility Subroutines	`s$vc_set_wait_mode`	Enable wait mode for subsequent operations.
	`s$vc_set_no_wait_mode`	Enable no-wait mode for subsequent operations.
	`s$vc_set_interrupt_mode`	Set the interrupt notification/confirmation mode.

Shared Virtual Memory

Two or more processes can share regions of their virtual address space by connecting the regions through a file using the `s$connect_vm_region` subroutine. When two or more processes share regions of their virtual address space by connecting to files, the operating system maps the virtual addresses of each process to the same physical memory. (The physical memory may be in main memory or on disk, as is any other pageable data.) When

one process makes a change to the shared region, the change is immediately visible to all other processes sharing the region.

The file must be a fixed file of record length 4096. You must open it for virtual memory access.

All processes sharing virtual memory must be running on the same module. The file must be on a disk that is local to that module.

When you call `s$connect_vm_region`, you define a region of a file (beginning at `starting_record_number` and `number_pages` long) and a region of your virtual address space (beginning at the location of `virtual_memory` and `number_pages` long).

The region of the file is valid if both of the following conditions exist:

- No page in the region is defined in any other region.
- All connections to each page in the region use the same starting record number and are the same number of pages in length.

The region of virtual memory is valid if both of the following conditions exist:

- It does not overlap any region of virtual memory defined by another call to `s$connect_vm_region`.

- If more than one process is connected to the region, then in addition to using the same starting record number and number of pages, all connections to the page use the same virtual memory starting address.

These rules mean that the first process to connect to a region of a file defines the size and location of the region in the file and in virtual memory. All subsequent connections, whether by another or by the same process, must agree with the first in their use of the region.

As long as these rules are followed, processes can select any number of regions to share and can specify any region as shared.

Defining a Region of Shared Virtual Memory

All processes sharing a region of virtual memory must use the same virtual address to reference that region. The operating system provides two methods of defining a virtual memory region for this purpose.

- If your application consists of a single program being executed by several processes that share one or more regions of virtual memory, all the processes make the same call(s) to `s$connect_vm_region`. You can simply define page-aligned external variables for each shared region. Since a single program is involved, each process has the same virtual address for each region.

- If several processes executing different programs need to share virtual memory, external variables sharable by more than one program cannot be created at bind time. The binder can assign different virtual addresses to the same regions when the programs are bound separately. Therefore, you must first create data region object modules to which the binder assigns the same virtual addresses when the programs are

bound separately. Create these data region object modules using the command `create_data_object`. See the *VOS Commands Reference Manual (R098)* for a description of the `create_data_object` command.

Each program that will use the shared region must be bound the same size (that is, as a small program, a large program, etc., as described in the section "Process Virtual Address Space"). In the binder control file, all data regions shared by any process in the group **must be listed first and in the same order** in the list of object modules that you want bound together. This procedure guarantees that each shared region occupies the same virtual address for each process using it and ensures that a shared data region resides at the same virtual address for all processes sharing the region, regardless of the program they are executing.

Shared Virtual Memory Access Modes

The access mode permitted depends on the type of I/O and locking mode specified when the file was opened. Table 3-4 describes access modes.

Table 3-4. The Access Modes for Connecting a Process's Virtual Memory to a File

Code	Access Mode	I/O Type	Locking Mode
`1`	Read and execute access. The calling process may read or execute any page in the region. The `copy_on_write` switch must be false.	input	set-lock-don't-wait
`2`	Read and write access. The calling process may read or write any page in the region.	update	implicit
`3`	Interlocked-read and write access. The calling process may read or write any page in the region. All reads are done with a hardware interlock mechanism. (See the explanation immediately following this table.) The `copy_on_write` switch must be false.	update	implicit
`4`	Exclusive-read and write access. Only the calling process may read or write any page in the region. On an XA600 or an XA2000-series CPU (models 110 to 160), data in these pages can be cached for faster access by the CPU.	update	set-lock-don't-wait
`5`	No access. Any attempted access causes a signal of the error condition. This access mode is useful for establishing fences between memory regions.	any	any
`6`	Read access. The calling process may read any page in the region. The `copy_on_write` switch must be false.	update	implicit

When the calling process connects to a region of virtual memory in interlocked-read mode, the region of virtual memory connected consists entirely of lock words. The calling process can then implement an application-defined locking convention. This convention associates

lock words in a region of shared memory containing lock words with areas in shared regions containing other data.

The access mode interlocked-read uses an indivisible interlock mechanism with which you can control the access of processes to data in connected regions of virtual memory. If a page is connected in the access mode interlocked-read, read operations on words in that page are performed using a hardware interlock cycle. Hardware interlock cycles work by providing an indivisible means of testing and clearing a bit without another CPU or I/O controller accessing the same word during the cycle. When an interlock read operation is performed, memory returns the current value of the word to the CPU, which rewrites it back into memory, clearing the high-order bit.

An interlock word is in an *unlocked* state when the high-order bit is set to 1, that is, when a negative value is stored in the lock word. If a lock word is read and the high-order bit is set to 1, the act of reading resets the high-order bit to 0. Any other process reading the lock word at the same time sees the word with the high-order bit set to 0. To unlock a lock, write a word with the high-order bit set to 1.

A common application of this interlock has several processes reading the high-order bit of a lock word until one process reads the bit set to 1. The value of the word is returned to that process, and the high-order bit is automatically reset to 0. Other processes that read the lock word will read the high-order bit set to 0 until the first process resets the high-order bit by writing a value to it that sets the high-order bit to 1.

Note: The application is responsible for initializing any lock words it uses.

To write CPU-independent programs that perform interlock mode locking, you can use the subroutine `s$get_vm_lock_word` to read the lock word. Sometimes, however, a process uses interlock read access to connect a shared virtual memory region, and then continues to try unsuccessfully to lock a word. These prolonged, unsuccessful attempts can interfere with disk I/O. To avoid these problems, use the operating system library routines `lock_spin_lock`, `try_spin_lock`, and `unlock_spin_lock` to ensure that repeated unsuccessful attempts to lock a word are handled in a safe and compatible manner.

Multitasking Processes

The operating system multitasking facility allows a single process to manage multiple executing tasks, each of which proceeds independently through the same program code. All tasks share the same program module, although they may or may not be executing the same section of code in the program module. In most applications, each task is associated with a separate terminal; however, tasks can exist without terminals, and tasks can share terminals.

Memory Resources in a Tasking Environment

When a task executes, it uses some shared resources and some resources established only for itself. Figure 3-4 shows how the memory resources are allocated for a process executing a multitasking program. Compare this figure to Figure 3-2 earlier in this chapter, which shows memory for a nontasking process. Note that most of the user memory areas are shared by all tasks in the process. Also, all the process memory areas, the wired areas, the kernel, and the forms buffer are shared by all tasks in the process.

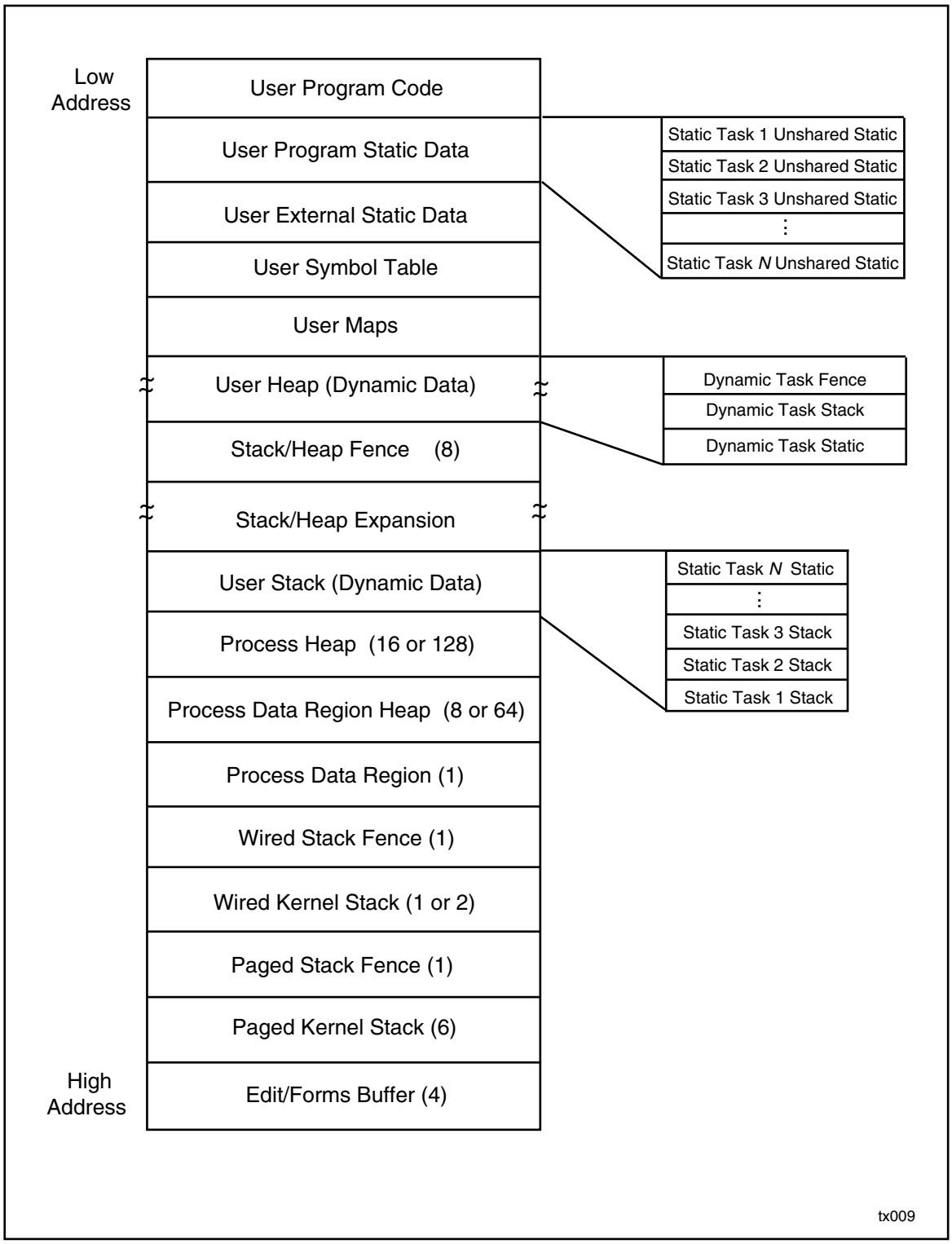

Figure 3-4. Memory Map for a Tasking Process

The following memory resources are not shared among tasks:

- Task stack. This stack contains the list of return entry point addresses generated by subroutine calls in the code; that is, it maintains proper program flow during execution. It also contains *automatic data*, which consists of variables that exist only while a called subroutine is active.

- Task static data area. This area contains all static variables for the task, except for the ones that are labeled as shared among tasks. *Static variables* are those whose value is either initialized or retained across subroutine calls.

The Task Data Region (TDR)

TDRTask data region The *task data region* (TDR) is a data structure maintained only for tasking programs. It is located in the user heap, and contains information such as task state, task priority, task terminal's port ID, CPU time spent on the task, and task stack length. The information contained in the TDR is available to the application program through the use of the `s$get_task_info` subroutine. Likewise, this information is available to the monitor task with the `display_task_info` request.

Chapter 4:
The VOS Command Language

This chapter provides a brief overview of the VOS command language. It describes how a command is processed, system command level, command forms, search rules, abbreviations, and the help system. It also lists operating system commands. For more information, see the *VOS Commands User's Guide (R089)*, *VOS Commands Reference Manual (R098)*, and *VOS System Administrator's Guide (R012)*.

Gaining Access

To gain access to the system, log in from a terminal connected to the system using the `login` command. With the `login` command, you must also specify your person name, which is listed in the system registration file, or an *alias*, which is a second, usually briefer form of your name; to be valid, an alias must also be listed in the system registration file.

After you issue the `login` command, the operating system prompts you for a password if your registration entry specifies that you must supply one.

To end an interactive process, use the `logout` command. The `logout` command also cleans up your process and address space.

A central registration database file maintains the following information for each user who has access to the system:

- person name (account name) and an optional alias
- the default group name and any other valid group names
- password, if required
- default priority and maximum priority
- whether or not the user can log in as privileged
- maximum number of concurrent processes that the user can create on a module.

A system administrator maintains the system registration file using the `registration_admin` command (see the *VOS System Administrator's Guide (R012)*). The system administrator also controls other aspects of logging into and out of the system using the `login_admin` and `logout_admin` commands.

Types of Commands

Commands are divided into two classes.

- *Internal commands* are those operating system commands that are always in a process's address space. They are part of the operating system and do not have corresponding `.pm` files.
- *External commands* must be loaded when you execute them. They are in the form of `.pm` files.

Some external commands are part of the operating system. Their `.pm` files are located in system libraries (such as `>system>command_library`). Other external commands are applications that you may have written.

Command macros are issued like commands, but the operating system processes them differently. Command macros are files (with the suffix `.cm`) that may contain internal and external commands, as well as command macro statements that control the execution of the command macro and allow you to pass parameters to it. For information on command macros and how they are processed, see the section "Command Macros" later in this chapter and refer to *VOS Commands User's Guide (R089)*.

The Command Processing Loop

As described in Chapter 2, the command processor accepts and executes commands. In an interactive process, it cycles through a series of steps called the command processing loop. The command processor performs the following steps:

1. Displays a prompting message and waits for a command. This waiting state is called *command level*.
2. Reads a command entered from the terminal.
3. Expands any abbreviations and processes command functions.
4. Executes the command line.
5. Returns to command level.

A single command line can contain one or more command strings, separated by semicolons. If so, the command processor expands abbreviations, processes command functions, and executes each in turn.

When you enter a command, you are specifying an object for the command processor to locate and then execute. The operating system must determine exactly which object you mean. You may be referring to a user-written program or to an operating system program. The operating system has no reserved command names, so you could specify a program with the same name as an operating system command. However, to resolve such an ambiguity, each process has a set of search rules that determine the order in which objects are located and subsequently executed.

When you enter a command, the command processor reads from your terminal one or more words, separated by spaces. It assumes that the first word is the name of a command. The process of separating a command line into individual tokens (the command itself and its arguments) is called *parsing* and is performed by the subroutine `s$parse_command`. This process is described in more detail in the section "Command Arguments."

To determine the object to which you refer when you issue a command, the operating system makes the following decisions:

1. It decides whether the first word is a path name by looking for the symbols `%`, `#`, `>`, and `<`.

2. It determines whether the first word has the suffix `.cm` or the suffix `.pm`.

Once these decisions are made, the operating system can begin the search for the object. If the command is not an internal command, the operating system searches your command library paths for the command. See the sections on "Library Paths" and "Command Search Rules" for more information.

The Display and Command Line Forms

Every operating system command except `debug` calls the `s$parse_command` subroutine. This subroutine, which is generally called at or near the start of command execution, parses the command line. It produces two forms of every command---the display form and the command line form---and also enables you to specify the `-usage` argument to get information on command usage.

The following sections describe the display and command line forms of a command. The `s$parse_command` subroutine is described in more detail in the VOS Subroutines Manuals.

The Display Form

To issue a command in the display form, type the name of the command and press the DISPLAY_FORM key or specify the `-form` argument and press the RETURN key. The resulting form displays a label and a field for each argument. The values initially displayed in argument fields of the display form of a command are default values (see the section "Command Arguments"). Fields that require arguments are displayed in inverse video. If the command has no arguments, the operating system displays the message:

```
No arguments required. Press ENTER to continue.
```

You can type in any number of a command's arguments after the command name, before you press DISPLAY_FORM. The values you give for the arguments are then shown in the initial display of the display form.

To move from one field to another, use the RETURN, TAB, UPARROW, and DOWNARROW keys. When all of the fields display the values that you want, you must press the ENTER key to issue the command.

In some cases, arguments can be longer than the space allowed for them in the display form. For example, many full path names do not fit in the space allotted. If you try to type past the

end of the field, the operating system returns the error message `Field overflow`. In this case, issue the command in command line form. For example, using the command line form of the `batch` command lets you enter a longer command line than using the display form.

The Command Line Form

To issue a command in the command line form, type the name of the command followed by any arguments you select, then press either the RETURN key or the ENTER key. This is how commands are issued under most operating systems.

Command Arguments

Most commands take arguments. An *argument* is a value given with a command that tells the command processor how to execute the command. For example, the *`file_names`* argument of the `print` command tells the operating system which files to print.

Some arguments have default values. A *default value* is a value predefined by the operating system that is in effect when you do not specify a value. For example, in the `print` command, `1` is the default value for the `-copies` argument. In some cases, the default depends upon the current value of some system parameter. For example, for the `update_channel_info` command, the values for `-baud`, `-parity`, and so forth are the current values for these parameters for the channel specified.

Operating system commands use the `s$parse_command` subroutine to read command arguments, to specify possible values, to provide defaults, and to provide limits on data types for an argument.

There are three types of arguments as defined by `s$parse_command`:

- positional arguments
- option arguments (also called keyword arguments)
- switch arguments.

Positional Arguments

Positional arguments are arguments that you specify in the command line form of a command (see "The Display and Lineal Forms," later in this chapter) in the appropriate location in the command line without preceding them by a keyword. For example, you specify the file name to `delete_file` as a positional argument.

```
delete_file myfile
```

Similarly, the `rename` command takes two positional arguments.

```
rename myfile yourfile
```

Option Arguments

Option arguments (also called *keyword arguments*) are preceded by a keyword (the name of the argument preceded by a hyphen) in the command line form. For example, if you specify a queue to the `print` command, you use the `-queue` positional argument:

```
print myfile -queue letterquality
```

Some option arguments have two or more predefined values; you can specify only one of these predefined values and any other value is invalid. In most cases, one of the predefined values is the default. For example, the `-format` argument of the `set_ready` command can have the values `medium`, `long`, `brief`, or `off`; the default is `medium`. When an argument has two or more predefined values, they are referred to as *cycle values* and the fields that contain them are called *cycle fields*; you step through the possible values in the display form using the CYCLE and CYCLEBACK keys. You can also use the LEFTARROW and RIGHTARROW keys to step through the list of values.

Switch Arguments

A *switch* argument is an option argument that has two predefined values: "yes" and "no." For example, see the `-line_numbers` argument of the `print` command. It differs from other option arguments in the way it is issued in command line form. To specify the negative (no) option, you do **not** specify `-line_numbers no`. Instead, you specify `-no_line_numbers`.

Abbreviations

After reading a command entered from your terminal, the operating system expands abbreviations and processes command functions. This section explains the steps followed in expanding abbreviations.

For a process to use abbreviations, it must have issued the `use_abbreviations` command. This command processes an abbreviations file, which contains one or more replacement directives. A *replacement directive* is a line of text that contains a token that the operating system is to replace with one or more specified. There are four types of directives.

- A *first directive* tells the operating system to replace the specified string when it is the first token in a command string.
- A *subsequent directive* tells the operating system to replace the specified token when it is after the first token in a command string.
- An *all directive* tells the operating system to replace the specified token regardless of where it is in a command string.
- A *symbol directive* tells the operating system to replace one character with another.

The operating system expands abbreviations following these steps.

1. The operating system replaces symbols according to symbol directives first. Since symbol directives output characters that delimit words and identify path names in a command line, this step allows the operating system to separate the string into its individual components.

2. If a token appears in your abbreviations file as input to an `all` or a `subsequent` directive, and if its position in the command is the same as the position specified in the directive, the operating system replaces it with the directive's output string.

3. The operating system evaluates all command functions and substitutes the values they return into the command line. (See the section "Command Functions" for an explanation of command functions.)

4. The operating system now expands first directives. The first word in the command begins with the first non-blank character in the command string and ends at the first unquoted space or semicolon (`;`). Any character preceding the first space or semicolon is included in the first word. If the first word matches the input token of a `first` directive, and if it is not part of the output of an `all` directive or a command function, then the command processor replaces the word with the first directive's output string.

5. Next, the operating system evaluates abbreviation parameters. If a parameter is of the form `&`*`n`*`&`, it replaces the parameter with the token in position *`n`* after the first token in the command. If the parameter is of the form `&f`*`n`*`&`, the operating system replaces the parameter with the token in position *`n`* and with all remaining tokens in the command string.

   If there are more abbreviation parameters than tokens in the command string, the operating system substitutes an empty string for each of the extra parameters. This has no effect on how the command executes.

6. Finally, the operating system evaluates command functions again. During the expansion of abbreviations, one or more new command functions may be introduced in the command; these functions are now evaluated.

   For example, suppose you want to list all files in your home directory that contain the current date as a suffix. Suppose you also have the following lines in your abbreviations file:

   ```
   all   HD       by (home_dir)
   first ls       by list -full
   subsequent -s  by -sort
   ```

   You could issue the following command:

   ```
   ls HD>*.(date) -s size
   ```

The operating system then follows these steps in expanding the abbreviations.

7. There are no symbol directives, so nothing happens at this step.

8. The `all` and `subsequent` directives are expanded, so the command becomes:

   ```
   ls (home_dir)>*.(date) -sort size
   ```

9. Command functions are expanded so the command becomes:

```
ls %s#d01>Sales>Smith>*.89-12-25 -sort size
```

10. First directives are replaced so the command becomes:

```
list -full  %s#d01>Sales>Smith>*.89-12-25 -sort list
```

11. None of the abbreviations use parameters, so nothing happens at this step.

12. No further command functions were introduced in the expansions that occurred since step **3**. Abbreviation expansion is therefore complete.

The `!` character can be used to suppress expansion of abbreviations. If the `!` character appears in a command line, no abbreviations to the right of this character are expanded.

Retrieving a Command Line

The operating system has two storage areas where it saves the last command line you entered. You can retrieve command lines from these storage areas, and thus avoid retyping them. Once a command line has been retrieved, you can edit it before entering it again.

The storage areas are called the insert saved buffer and the insert default buffer.

Pressing RETURN or DISPLAY_FORM places the text on the command line in both buffers; the old text cannot then be retrieved. Pressing ENTER replaces text in the insert saved buffer only.

To retrieve text stored in the insert saved buffer, press the INSERT_SAVED key. To retrieve text stored in the insert default buffer, press the INSERT_DEFAULT key.

Note that when you enter a break level command (such as typing `s` for `stop`) or when you answer a prompt during the execution of a program (such as answering `y` to the prompt `Verify deletions of` *`star_name`*`.` *`file_name`*`?`), pressing RETURN stores the text in the insert saved buffer only. In either case, when your process returns to command level, you can use the INSERT_DEFAULT key to retrieve the command line you entered the last time your process was at command level.

Break Level

When you issue a break request by pressing the CTRL BREAK keys simultaneously on your terminal, the operating system suspends an executing program. After you issue the break request, your process is at break level. Some program errors can put a process at break level. The operating system indicates that you are at break level by displaying the following message on your screen:

```
BREAK
Request?  (stop, continue, debug, keep, login, re-enter)
```

At break level, you can issue only these six commands.

- The `stop` command tells the operating system to terminate the program abnormally, discard all current command macros, and return your process to command level.
- The `continue` command tells the operating system to resume processing as if the interruption had not occurred.
- The `debug` command invokes the symbolic debugger, allowing you to examine your program and set breakpoints. You can restart the program from the debugger or terminate it.
- The `keep` command saves an interrupted executable image of the program in a file that can be debugged later. This file is called a keep module and has the suffix `.kp`. After issuing the `keep` command, you can issue any of the other break level commands.
- The `login` command creates a new log-in process for you, as a subprocess of the current process.
- The `re-enter` command returns control to the interrupted program at a predetermined execution point. This is generally only possible with commands that place you into a subsystem from which you can enter requests. You can then issue a break request to suspend execution of the request, and use `re-enter` to return to request level.

When the execution of a program terminates, whether normally or abnormally, all of the files it opened are closed, all of the locks that it held are unlocked, all of the events it attached are detached, and all of the ports it attached are detached.

In some circumstances, issuing the break request does not put your process at break level:

- when your process is in the display form of a command.
- when your process is in a program that has special break processing, such as Emacs or the Word Processing Editor.
- when your process is at command level or at break level. You receive a message telling you that you are at command or break level and that the break is being ignored.
- when the `set_terminal_parameters` command has been used to set your terminal so that it ignores the break request. Note that when you log in, breaks are disabled until your start-up command macro completes execution, or until it specifically enables breaks with the `set_terminal_parameters` command.

Note that for most terminal types you can also issue a break request by pressing CTRL-C.

Operating System Commands

This section lists operating system commands, grouped by function. For more information on these commands, see the *VOS Commands Reference Manual (R098)* and the *VOS System Administrator's Guide (R012)*.

Languages and Programming

add_entry_names
add_profile
assemble
basic
bind
bind_kernel
c
c_preprocess
cobol
create_data_object
debug
fortran
mp_debug
pascal
pl1
profile
use_message_file

Batch Processing

batch
batch_admin
cancel_batch_requests
cancel_device_reservation
create_batch_queue
display_batch_status
list_batch_requests
load_control_admin
move_device_reservation
reserve_device
set_cpu_time_limit
set_priority
sleep
start_process
stop_process
update_batch_requests

Library Paths

add_default_library_path
add_library_path
delete_default_library_path
delete_library_path
list_library_path
set_default_library_path
set_library_path

Ports

```
attach_default_output
attach_port
detach_default_output
detach_port
list_port_attachments
start_logging
stop_logging
```

Break/Interrupt

```
break_process
continue
debug
keep
re-enter
stop
```

Process Management

```
call_thru
display_date_time
display_error
display_line
display_notices
help
login
logout
set_ready
send_message
set_time_zone
use_abbreviations
verify_system_access
where_command
```

Printing

```
cancel_print_requests
create_print_queue
display_print_default
display_print_status
list_print_requests
spooler_admin
print
```

File Management

```
compare_files
copy_file
create_deleted_record_index
create_file
create_index
create_record_index
delete_file
```

```
delete_index
display
display_file
display_file_status
dump_file
dump_record
enforce_region_locks
locate_files
move_file
rename
set_expiration_date
set_file_allocation
set_implicit_locking
set_index_flags
set_pipe_file
set_safety_switch
sort
text_data_merge
truncate_file
where_path
who_locked
```

Directory Management

```
change_current_dir
compare_dirs
copy_dir
create_dir
delete_dir
display_current_dir
display_dir_status
link
link_dirs
list
move_dir
rename
unlink
walk_dir
where_path
```

Access Control

```
display_access
display_access_list
display_default_access_list
give_access
give_default_access
propagate_access
remove_access
remove_default_access
set_owner_access
```

Tape Processing

```
dismount_tape
display_tape_defaults
dump_disk
dump_tape
list_save_tape
list_tape
mount_tape
position_tape
read_tape
restore
restore_object
save
```

System Information

```
analyze_system
display_current_module
display_system_usage
list_gateways
list_modules
list_systems
list_users
who_locked
```

Devices

```
add_device
configure_devices
display_device_info
list_devices
```

Disks

```
add_disk
cancel_disk_retry
cancel_fast_disk_recovery
display_disk_info
display_disk_usage
configure_disks
delete_disk
dismount_disk
display_bad_blocks
display_disk_label
dump_disk
format_disk
initialize_boot_disk
initialize_duplex_disk
iop_disk_tape_admin
mount_disk
recover_disk
reload_disk
remove_disk_pack
set_partition_size
```

```
setup_disk_pack
start_disk_recovery
uninitialize_disk
update_disk_label
```

Tape Processing

```
save_object
set_second_tape
set_tape_defaults
system_operator
write_tape
```

Editing

```
edit
edit_form
emacs
```

Terminal Management

```
display_terminal_parameters
list_terminal_types
set_ready
line_edit
wp
```

System Administration Commands Not Listed Elsewhere

```
accounting_admin
broadcast
broadcast_file
cancel_broadcast_requests
check_jiffy_times
comment_on_manual
configure_boards
configure_comm_protocol
configure_languages
copy_dump
copy_kernel
create_os_symtab
create_table
create_user_sysdbs
display_calendar_clock
display_lock_wait_time
display_scheduler_info
display_software_purchased
display_tuning_parameters
dump_accounting_file
link_boot_server
list_broadcast_requests
list_comm_protocols
load_control_histogram
load_kernel_program
log_registered_users
```

```
log_syserr_message
login_admin
logout_admin
maint_request
make_message_file
memory_control
network_watchdog
notify_hardware_error
notify_security_violation
process_broadcast_queue
reconfigure_memory
recreate_tin
registration_admin
reset_configuration
rsn_mail
rsn_mail_janitor
set_bootload_time
set_date_time
set_default_time_zone
set_jiffy_times
set_lock_wait_time
set_registration_info
set_scheduler_info
set_system_log_mode
set_tuning_parameters
shutdown
site_call_system
start_mailhandler
start_rsn
update_user_info
```

Command Macros

A *command macro* is a text file that contains one or more commands and/or command macro statements and that has a name ending with the suffix `.cm`. You invoke a macro by entering its name. The operating system then sequentially executes the commands and macro statements in the file.

A command macro is useful to replace one of the following:

- a long command string that you issue frequently
- a group of internal commands, command macros, and/or program modules that together perform an operation you regularly require
- a sequence of internal commands, command macros, and/or program modules that you execute in a certain way depending on external factors.

The *macro processor* is that part of the operating system software that reads macro files and processes macro statements (defined later in this section). The macro processor does not process command lines and macro input lines that it encounters in macros; instead, it returns

these lines to the command processor. The operating system starts up the macro processor when a command macro is issued.

A line of a command macro can be any of the following:

- a command line
- a macro line
- an input line.

Command Lines

A *command line* is a set of one or more command strings, separated by semicolons. A *command string* is a command name and any of the command's arguments you select. If you want to issue a command with the same name as the command macro, include the `.pm` suffix in the command name. Otherwise, the command macro will be called recursively.

Macro Lines

A *macro line* can be one of the following:

- a command macro statement
- a comment line
- a parameter declaration.

A *command macro statement* (or, simply, a *macro statement*) is a statement in the form `&keyword` that can be included in a command macro to perform a predefined function.

Table 4-1 lists the valid macro statements. See the *VOS Commands User's Guide (R089)* for more information.

Table 4-1. Command Macro Statements *(Page 1 of 2)*

`&attach_input`	Detaches the default input port from its current attachment and attaches it to the current command macro.
`&begin_parameters`	Begins a macro parameter declaration section.
`&control`	Turns the macro processor on or off.
`&detach_input`	Reattaches the default input port to the attachment it had prior to the most recent `&attach_input` statement.
`&display_line`	Writes the text given in the statement to the default output port.
`&display_line_partial`	Writes the text given in the statement to the default output port without a trailing carriage return.
`&echo`	Controls whether command lines, input lines, and macro statements are written to the default output port.
`&end_parameters`	Terminates a parameter declaration section.
`&eof`	Passes an end-of-file status code to the program reading the file.

Table 4-1. Command Macro Statements *(Page 2 of 2)*

`&eval`	Forces evaluation and execution of a specified string.
`&goto`	Indicates which line of the macro to execute next.
`&if`	Evaluates a logical expression to control the execution of a command macro. The statement `&then` must be used with `&if`. The statement `&else` can optionally be used.
`&label`	Sets a target for a `&goto` statement.
`&mode`	Controls the behavior of a macro in the event of an error.
`&return`	Terminates the current command macro.
`&set`	Evaluates an expression and assigns the value to a variable.
`&set_string`	Assigns a character-string value to a variable.

You can put only one macro statement on a line. You cannot use abbreviations in macro statements.

A *comment line* is either a line that begins with an ampersand (`&`) followed by a space or a line that consists only of an ampersand. The characters that follow the ampersand on the line, if any, are taken to be the comment. The command processor ignores all comment lines.

Comment lines can be used to explain the purpose of the macro or to describe how it works. Comment lines containing only an ampersand can be used to improve the readability of a macro by clarifying its organization.

For example, the first two lines of the following macro are comment lines.

```
& This command macro is named set_cursor.cm.
&
!set_terminal_parameters -cursor_format blinking_underline
```

Parameters

The macro can take parameters. A *parameter* in a command macro is a named variable that is declared within the parameter declaration section of a command macro. A parameter receives its initial value from an argument specified in the command string that calls the macro or from a default value specified within the macro.

The macro processor stores parameters as character strings. When the value of a parameter is converted, it is treated as numeric if it can be converted to a number; otherwise the value is treated as a string.

Parameter Types

There are three types of parameters: positional, optional, and switch parameters. These correspond to the three types of command arguments described earlier in this section.

A *positional parameter* is a parameter that the macro processor identifies in a command line by the position of its value (rather than by a keyword).

An *optional parameter* is a parameter that the macro processor identifies in a command line by a keyword that begins with a hyphen and is followed by a value. You can specify allowed values (which the user can then cycle through in the display form). You can also specify a default value.

A *switch parameter* is a parameter that has only two possible values: `yes` or `no`. It is used like a command switch argument.

Parameter Data Types

Possible data types for parameters in a command macro are shown in Table 4-2. These data types can be modified by various qualifiers. See the *VOS Commands User's Guide (R089)* for more information.

Table 4-2. Command Macro Parameter Data Types

Data Type	Description
`date_time`	A date/time string.
`device_name`	The path name of a device.
`module_name`	The path name of a module.
`name`	An ASCII character string of up to 32 characters.
`number`	A string of integers.
`pathname`	A path name or a star name.
`starname`	A star name or a path name.
`string`	Any string of characters up to 300 characters long.
`system_name`	The path name of a system.
`unclaimed`	Any parameters not specifically declared with a parameter descriptor.
`user_name`	The name of a user.

Parameter Declarations

A *parameter declaration* is a line in a command macro that appears between the macro statements `&begin_parameters` and `&end_parameters`. It must contain the name of a parameter and it can contain a parameter descriptor (described later in this section). The form of a parameter declaration follows.

```
parameter_name [parameter_descriptor]
```

That portion of a macro from the `&begin_parameters` statement to the `&end_parameters` statement is called the *parameter declaration section*. The parameter declaration section must begin on the first line of the macro that is not a comment line.

If a command macro has one or more parameters defined in a parameter declaration section, the macro accepts arguments from the command line. When you issue a command macro with

one or more argument values, each value is assigned to the parameter whose description it matches; the input value becomes the initial value of the corresponding parameter.

Parameter Descriptors

A *parameter descriptor* is that part of a parameter declaration that specifies certain characteristics of the parameter. For example, some of the characteristics you can specify are:

- whether the user is required to supply a value for the parameter
- the type of the parameter (positional, optional, or switch)
- the data type of the value
- the default value, if any.

If a parameter declaration contains no descriptor, the parameter has the following characteristics by default:

- It is not required.
- It is a positional parameter.
- Its data type is a character string of up to 256 characters.

If you give a parameter descriptor, it must contain at least a specifier (see "Parameter Specifiers").

Parameter Labels

A *label* is an optional element of a parameter descriptor. Its purpose is to provide a descriptive term that will be useful to the user of a macro. It has the following form:

```
label:
```

The label, if specified, must be the first component of the parameter descriptor. The colon following `label` must be the only colon in the descriptor.

Depending on the type of the parameter, the label appears in the display form, the command line form, both of these, or neither of these. The information provided by `label` is used differently depending on the type of the parameter.

- For a positional parameter, it labels the parameter's field in the display form. It also provides a label in the command line form when you give the command macro with the `-usage` option.

- For an optional parameter, it labels the parameter's field in the command line form only.

- For a switch parameter, there is no purpose for specifying a label, since it is used in neither the display form nor the command line form.

Parameter Specifiers

A *specifier* is an element of a parameter descriptor that defines the type of the parameter. It is the principal element of the descriptor; a parameter descriptor is invalid if it does not contain a specifier.

There are three types of parameters, and each type uses a different form for *specifier*:

A positional parameter has the following form.

parameter_data_type[suffix]

An optional parameter has the following form.

option(*option_keyword*),*parameter_data_type*[suffix]

A switch parameter has the following form.

switch(*switch_keyword*)

Input Lines

You can use a command macro to call a program such as `line_edit` or `analyze_system` that places the calling process into a request loop. You then use subsequent lines of the macro to provide input to the program and to exit from the program.

To supply input to a program in a macro, you must include the following elements in the macro in the order shown:

- the `&attach_input` macro statement, which allows the program to accept input directly from the macro file
- the command to invoke the program
- input to the program (probably including a request such as `quit` to exit from the program)
- the `&detach_input` macro statement, which prevents any further program input from coming from the macro.

Command Functions

A *command function* is a self-contained function that you can use as an argument in a command line. Before executing the rest of the command line in which a command function appears, the command processor evaluates the command function and replaces it with a value. The value returned becomes part of the command line. For example, to display the current time, you could enter the following command line with the command function `(time)`:

```
display_line (time)
```

The output for the preceding example could be `13:10:59`, which specifies the current hour (`13`, in 24-hour format), minutes (`10`), and seconds (`59`).

The most common uses for command functions are in abbreviations and in command macros. However, some functions are also frequently used with commands. For a list of those command functions and information about how they can be used with commands, see the *VOS Commands Reference Manual (R098)*.

When you use command functions, note the following:

- Parentheses are part of every command function; you must include them whenever you type a command function.
- The command processor expands abbreviations before it evaluates command functions. You can, therefore, create abbreviations for command functions. (Chapter 5 describes abbreviations.)

To suppress immediate evaluation, you can enclose the command function in quotes.

You can use command functions to return values of the following data types:

- path names
- user names
- module, system, or device names
- numeric values
- date-time values
- character-string values.

See the *VOS Commands User's Guide (R089)* for more information on command functions.

Library Paths

The operating system maintains a set of library paths for each process. *Library paths* specify where the operating system searches for external commands, include files, object files, or message files. The libraries are referred to as the command, include, object, and message libraries.

Library paths fall into two categories. *Default library paths* are set for an entire module. All processes automatically include all default library paths in their library search paths. These library paths are generally set using the `add_default_library_path` command in `module_start_up.cm`. Typically these are library paths that all or almost all processes on the system will need (such as `>system>command_library`).

A process can establish other library paths for itself using the `add_library_path` command. Library paths set with this command are for the duration of that process only. Library paths established for a process are not passed to subprocesses or batch processes it starts; these processes must add their own library paths. Typically, non-default library paths are added in a user's start-up command macro.

Command Search Rules

Figure 4-1t shows the search rules the command processor follows when it tries to locate a command.

The figure depicts, from top to bottom, the order in which the operating system conducts the search. The search ends when the object is located. If it fails to locate the object (because it doesn't exist according to the current search rules), it displays the message `Object not found`. The top of the figure shows the order of the first two determinations the operating

system makes, namely, whether the object is a path name and if it has a suffix of either `.cm` (for a command macro) or `.pm` (for a program module).

Figure 4-1 shows the search paths in the case where there is only one library path (`system>command_library`) in the command library. Figure 4-2 shows a more complex example with more libraries in the search path.

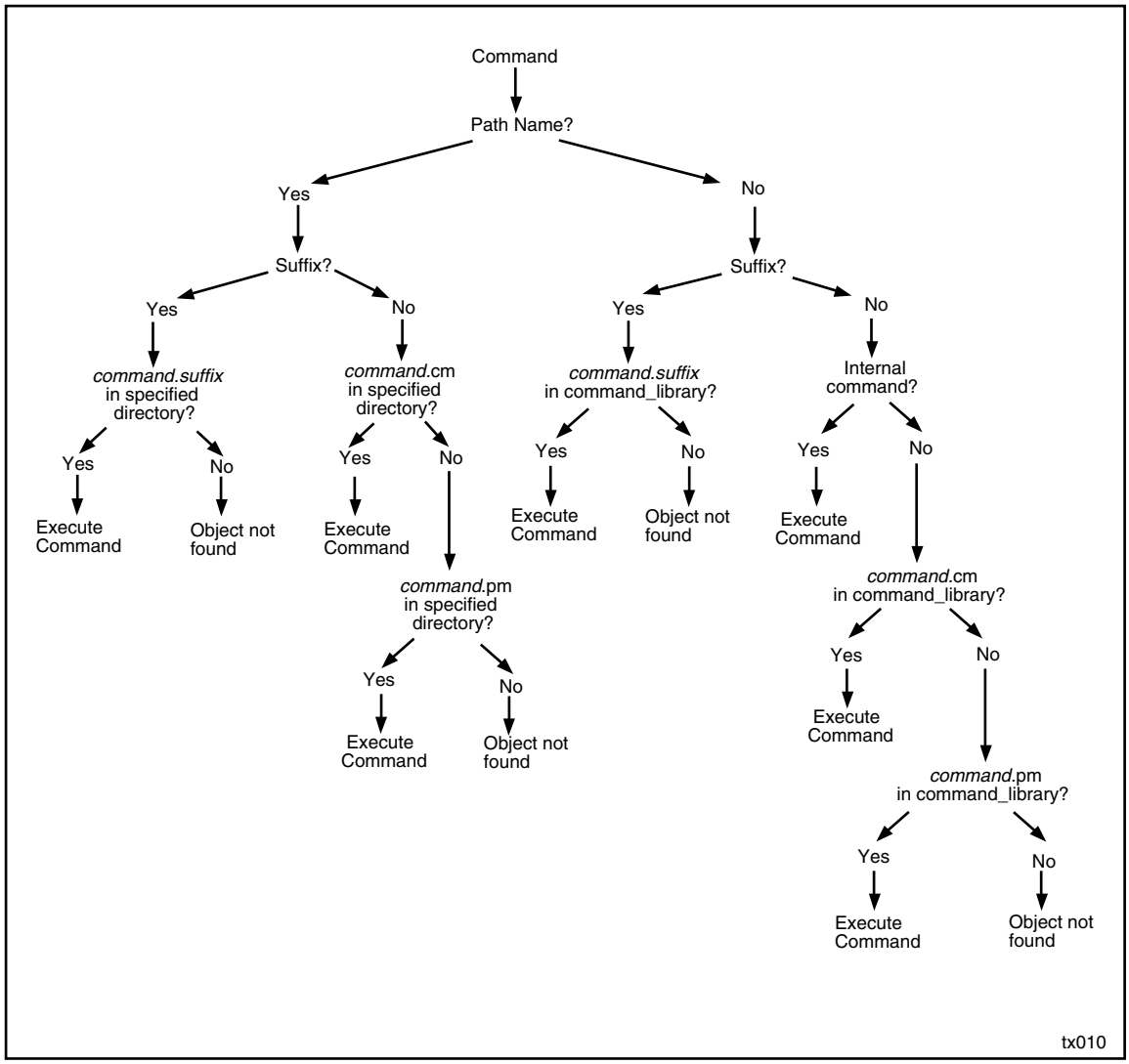

Figure 4-1. The Operating System's Command Search Rules

Note the following conditions.

- When you omit a suffix, the order of precedence is first `.cm`, then `.pm`.

- When provided with the least amount of information (an object name with no path name or suffix), the operating system relies entirely on the search rules as shown in Figure 4-1, under the right hand column.

The order of precedence in searching for a command when you omit a path name and a suffix is:

1. an internal command
2. a command macro in `command_library`
3. a program module in `command_library`.

Changing the Search Rules for a Module

Frequently, users on the same system require shared access to commands and programs contained in directories that are not on the operating system's default search list. Rather than allocating many copies of the same object to multiple users, it is more efficient to add library paths to the command library that point to directories that contain site-specific tools and applications that all users can access.

System administrators can use the `set_default_library_paths` command to change the list of directories that the operating system searches through for a specified library. For example, suppose the directory `%s1#d01>Accounting>tools_library` contains programs and macros used throughout the Accounting Department. The system administrator could add this directory to the module's list of command library directories. The changes are effective for all new processes in a module. It has no effect on processes that were already in existence when the command was issued.

One library path that is typically defined with the `set_default_library_paths` command as the first path name in a given library for a module is the command function `(current_dir)`. When the operating system encounters `(current_dir)` as a library path, it expands the command function relative to the current process's current directory each time it searches for an object.

You can display the list of library paths defined for your module with the command `list_default_library_paths`.

For information about the commands `set_default_library_paths` and `list_default_library_paths`, see the *VOS System Administrator's Guide (R012)*.

The `(current_dir)` command function is generally at the top of the list for each library. This means that the current directory is the first place the operating system looks for a command (after it determines that the command is not an internal command), an include file, or an object module.

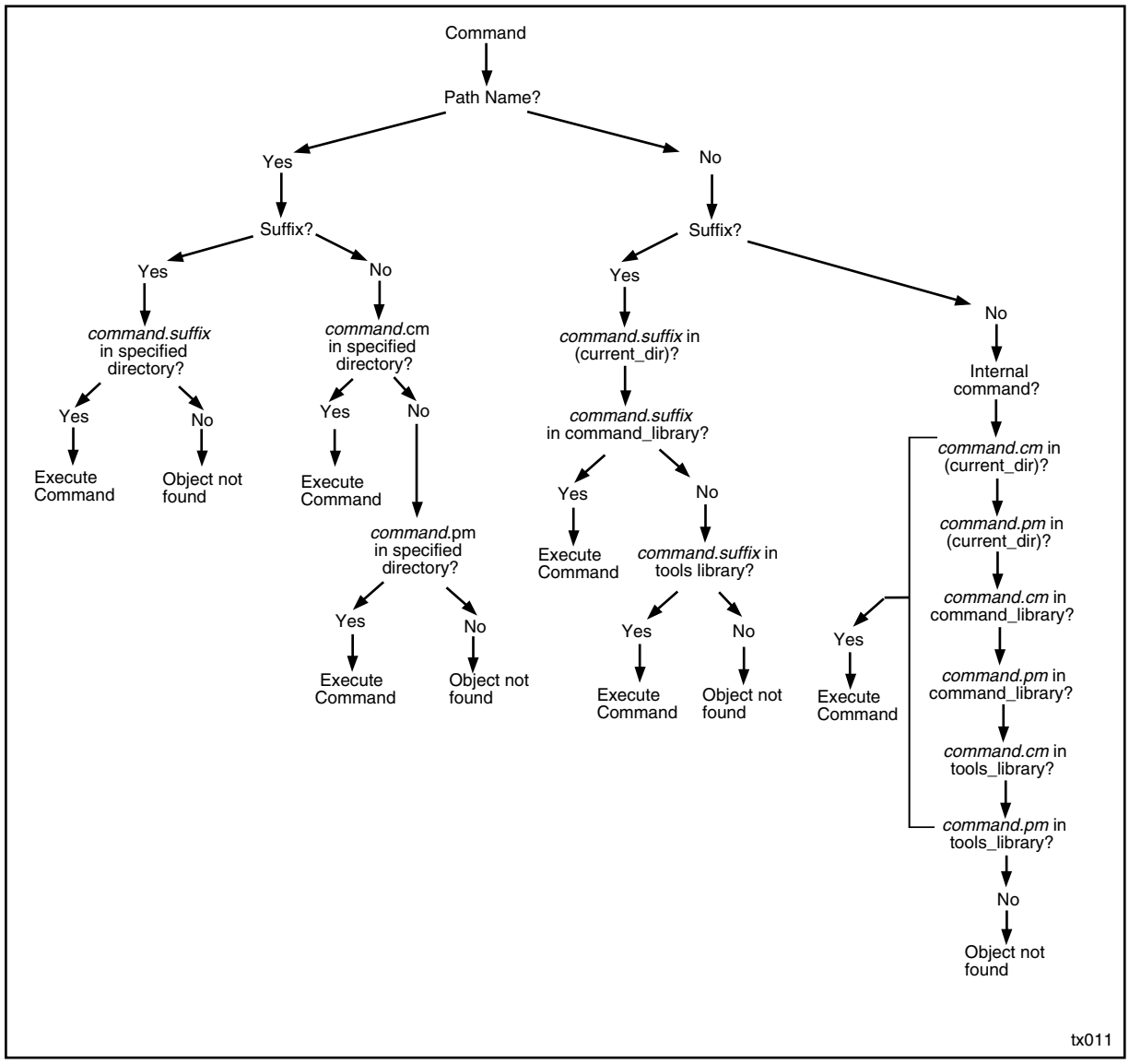

Figure 4-2. Sample Search Path for a Command

Changing Search Rules

Several commands allow you to create and maintain a list of library paths for your own processes. The operating system uses the order you specify to search for a command before it uses the order specified in the library path list defined for the module.

The commands that change the list of library paths for your current processes are:

- `add_library_path`. This command adds a directory to your list of library paths for the current log-in process.

- `delete_library_path`. This command deletes a directory from your list of library paths for the current log-in process.

- `set_library_paths`. This command defines a new list of library paths for the specified library for the current log-in process.

To list the library paths for your process, use the `list_library_paths` command. To list the default library paths for the module, use `list_default_library_paths`.

The system administrator can change the default library paths for the module using `add_default_library_path`, `delete_default_library_path`, and `set_default_library_paths`.

The Help System

The operating system help facility can be used to obtain more information about commands, how to use them, and what arguments they take. The help system can be used in four ways:

- You can issue the `help` command with no arguments to enter the help subsystem. From there, you can use the menus to get help on the topics you desire.

- You can issue the `help` command followed by the name of the command you want help on. The `help` command returns information on that command.

- You can issue the help command and specify either the `-type` or the `-match` arguments to get a list of commands matching the argument specifications. The `-type` argument enables you to list commands by type (internal, external, or command function). The `-match` argument causes help to list all commands in your command library paths that contain a string you specify.

- You can press the HELP key to get context-dependent help. If you press the HELP key at command level, you enter the help subsystem.

For more information on how to use the help system, see the *VOS Commands User's Guide (R089)*.

Chapter 5:
The File System

This chapter discusses the VOS file system. It describes file and index types, file I/O, file locking, access control, directories, and transaction protection.

A *file* is a named, logical unit of storage containing a sequence of records. The maximum file size is approximately 2.15 gigabytes. (Larger files are possible when extents are used. See the section "File Extents.") Files can span multiple physical disks. Logical disks cannot be spanned. The maximum record size is 32,767 bytes. Records in sequential, relative, and fixed files always start on word boundaries (that is, the offset of a record in the file is a multiple of two). Odd-length records are padded to the even byte.

You can create files using the `create_file` command or using any of the commands that create files as part of their operation (such as the editors, the compilers, and the binder). An application program can create a file by explicitly calling `s$create_file` or by opening a non-existent file for a write or append operation.

Files are allocated in units of *blocks*. A block is 4096 bytes in size. Commands that show file sizes (for example, `list` and `display_file_status`) do so in blocks. Records in files can span blocks.

Programs can read from or write to files using programming language statements or functions (for example, `read` and `write` in VOS PL/I and `fgetc()` and `fputc()` in VOS C). They can also use VOS subroutines such as `s$seq_read` and `s$seq_write`. Subroutines that can be used for specific operations are described below in the section "File I/O Operations." For more information on these subroutines, refer to the VOS Subroutines Manuals.

As discussed in Chapter 2, "The Operating System," all I/O is handled through the VOS port mechanism. An application program must first attach a port to a file using `s$attach_port`, then use the returned port ID in subsequent I/O operations. The I/O statements in each programming language also use the port mechanism, though this is generally hidden from the user.

Access control is also managed as part of the file system. Directories and the files within them have default access lists and access lists which determine who can access the files and at what level (that is, execute only, read only, or read and modify).

VOS File Formats

The four basic file types used for storing information are:

- sequential files
- relative files
- fixed files
- stream files.

Queues and pipe files are also treated as files by the operating system. They are used for interprocess communication rather than data storage, and are described in Chapter 3, "Processes and Interprocess Communication." They are not discussed further in this chapter. However, everything described in this chapter related to access control also applies to these files, and many of the VOS file commands such as `display_file_status` and `delete_file`, also work with queues and pipe files.

Sequential File Organization

In a sequential file, all records are of varying length. Records can vary in length from 0 to 32,767 bytes. Records start and end with two bytes (one word) that contain the length of the record. The length words always begin on an even boundary; odd-length records thus have pad character after the data and before the terminating length word. The format of records in a sequential file is shown in Figure 5-1.

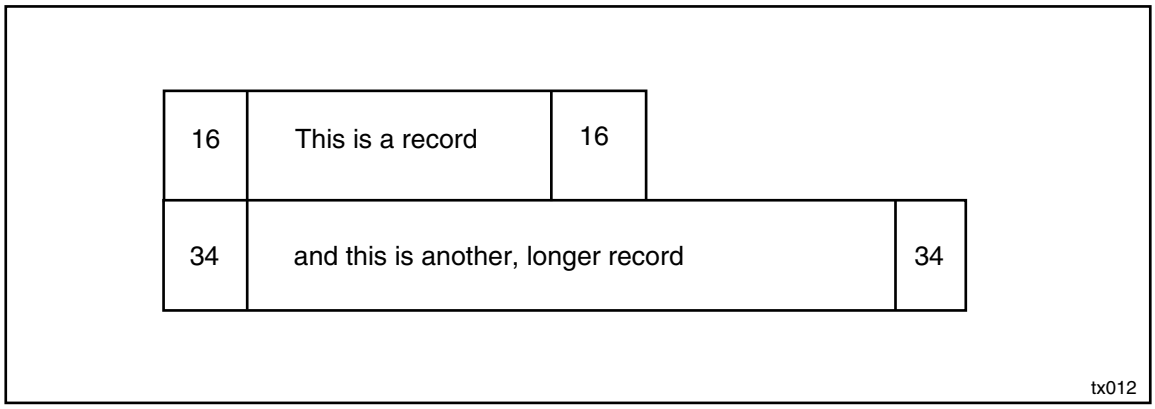

Figure 5-1. Sequential File Format

When you delete a record in a sequential file, the data is not changed but the value of the length fields is negated according to the following formula:

```
length field value = -(record size padded to even number  + 4) /2
```

For example, if a record of length 65 (0041 hexadecimal) is deleted, the length field is changed to -35 (FFDD hexadecimal). Thus, a value of -2 to -16,386 in the length words indicates that the record has been deleted.

The space from logically deleted records can be regained by using the `copy_file` command and specifying the `-pack` or `-truncate` argument.

This is the default organization for most commands which create files (including the `create_file` command and the VOS editors).

Relative File Organization

In a relative file, you define the maximum record size for the file. All records in the file can be no longer than this maximum record size. However, the actual amount of disk space taken up by each record is the same.

Figure 5-2 shows the organization of a relative file. Each record is stored with a length word that indicates the actual size of the record. Thus, the physical record size is two bytes longer than the maximum record size.

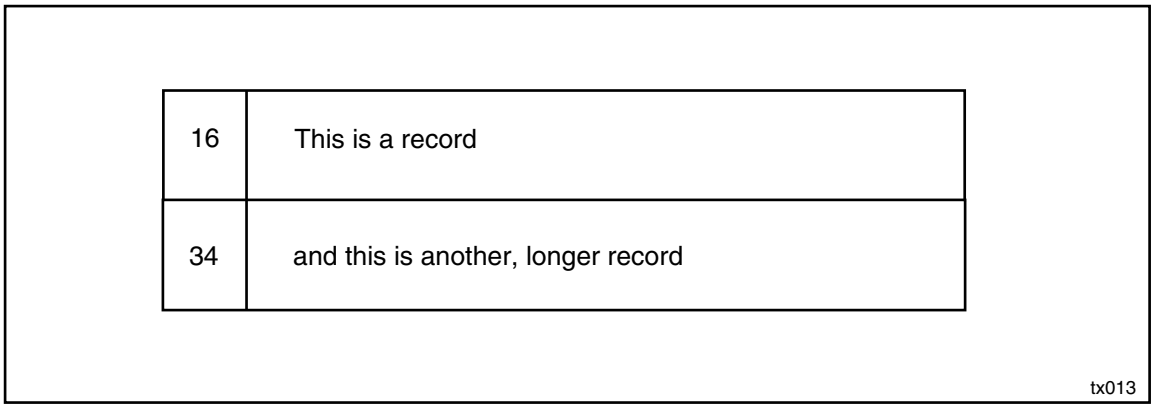

Figure 5-2. Relative File Format

The positive value in the size field specifies the number of bytes of data contained in the record. It can be less than or equal to the maximum record size. A value of -1 indicates that the record has been deleted. Any other values are illegal.

Only those file blocks containing records that are accessed by a read or write are allocated. That is, if you create a relative file with maximum record size of 510 bytes, 8 records fit in one block (4096 bytes). If you then write records 1 and 35, only two blocks (1 and 5) are allocated, not the intervening blocks (2, 3, and 4). However, the physical locations for records 2 through 8 and 33, 34, and 36 through 40 are in the allocated blocks (and have length words equal to -1). If you try to read any of these records, you will get the error `e$record_deleted`.

When you delete a record in a relative file, the data itself is not changed. The value of the size field is changed to `-1`. Deleted record space can be removed and the file packed by using the `copy_file` command and specifying the `-pack` option. However, this renumbers the records in the file, so you should **not** do this if you access the records by record number.

Fixed File Organization

A fixed file has fixed length records. The record contains only data. Unlike sequential and relative files, there are no length bytes, and so the records must all be the same length. Records cannot be physically deleted at the system level, since there is no way to indicate deletion. If you delete a record in a fixed file, no change is made to the file. An application using a fixed file must keep track of which records it has deleted. The operating system provides a way to do this using a special kind of index called a record index (see the section "Record Indexes"). If there is an embedded key index for the file, the key to the record is deleted. However, if the index is recreated, the deleted record will again have an index entry. See the section on "File Indexes" for more information on indexes.

Stream File Organization

A stream file contains variable length records. Unlike a sequential file, records are delimited by an ASCII line-feed character (`0A` hexadecimal) in the last byte of each record to mark the record end (see Figure 5-3). Bytes after the last line-feed character in a file are also treated as a record, as is a sequence of 32,767 bytes (the maximum record length) without a line-feed character.

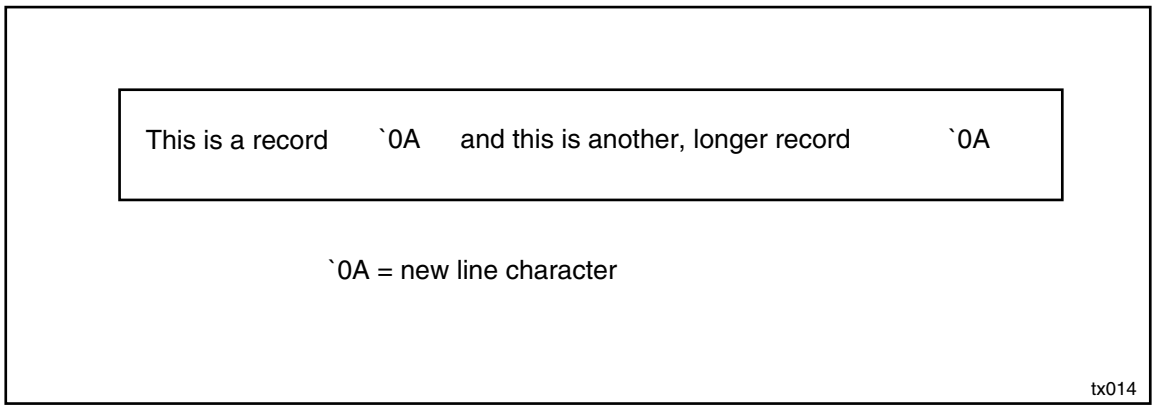

Figure 5-3. Stream File Format

Records cannot be deleted from stream files.

Stream files are data sensitive. If you insert a line-feed character within a record, the record becomes two records. To an end user or to an application program, a stream file behaves like a sequential file.

File Disk Storage

Files are allocated and stored on disk in units of blocks. When a file is created, storage is allocated on disk to hold information about the file, such as the file type, the maximum record size, and so forth. This storage is called the *file entry*. The file entry also contains the disk addresses of the first 16 data blocks in the file. If the file is longer than 16 blocks, subsequent addresses are stored in *indirect blocks*. The file entry contains the address of the first indirect block (labeled as level -1), which contains the addresses of up to 512 data blocks and up to

512 more indirect blocks (labeled as level -2). These level -2 indirect blocks can each contain the addresses of up to 1024 data blocks.

If the file is an extent file, the first address or addresses in the file entry will be those of file extents. See the section "File Extents."

File Indexes

Indexes allow fast random access to a file. The index is an ordered set of keys. Each key is associated with one or more records in the file. An application accessing records in the file can specify the key. The operating system then searches the index for that key and allows the application to access the record associated with it. This is faster than searching the file sequentially for a record, since the index is ordered, and the operating system can thus apply faster search techniques on it. (For more information on how an application accesses an indexed file, see the section on "Indexed Access" under "File Access Methods.") You can use the `-index` option of the `display` command to return the file in index-sorted order. Use `display_file_status` to see information about any indexes on a file.

Indexes can be created on any data file type. The file can be accessed through those indexes. There can be up to 64 different indexes on an individual file. The file system supports three different index types for performing indexed access:

- embedded key
- separate key
- embedded/separate key.

Embedded and embedded/separate key indexes cannot be used on stream files.

The file system also supports item indexes, which are used to quickly access small amounts of data, and deleted record indexes and record indexes, which are used to reclaim space in files.

Each index type is described in the sections that follow.

You can specify whether an index can have *duplicate keys* (that is, whether the same key value can point to more than one record). You can also specify whether *null keys* are allowed. If null keys are allowed, keys that consist entirely of a string of blanks are ignored and not added to the index. Thus, some records may not have keys in the index. If null keys are not allowed, a string of blanks can be a key.

Embedded-Key Indexes

In an embedded key index, the key is composed of components in the record. The file system automatically maintains embedded-key indexes. Every time a record is created, modified, or deleted, the index is automatically updated unless you have turned off automatic updating (see the section on "Creating Indexes").

Embedded key components are defined by their position within a record and their length. (For example, a key can be defined as starting at position 5 in the record and being 12 characters long.) The key can consist of one field of data from the record, or multiple components taken from various parts of the fields within the record. Regardless of how many data fields are used

to create the key, the key cannot be longer than 64 characters. Each record can be associated with only one key per index. If you specify that duplicate keys are allowed when you create the index, the same key can point to different records.

For an embedded key index, all information needed to determine a key is contained in the record being indexed. If an embedded key index is accidentally lost, it can be recreated using the `create_index` command.

Separate-Key Indexes

Separate-key indexes contain keys that are not necessarily part of the data record. An application defines the key independent of the contents of the record. When you create a separate key index, you cannot specify the `components` field in the `create_index` command.

For example, suppose you are indexing a file that contains information about animals. You could create a separate key index called “food” which would contain the type of food an animal eats:

```
honey-->bear
nuts-->squirrel
```

The actual record for bear may not contain any reference to what food it eats. The record may simply list species and habitat.

Separate key indexes must be maintained by the application, since the file system does not know what relations are being created.

Embedded-Separate-Key Indexes

An embedded-separate key index combines features of both an embedded and a separate key index. As with an embedded key index, there is a place in the data record where the key value is found initially, and when a record is added, updated, or deleted in the file, the system maintains the embedded keys. As with a separate key index, you can add or delete the separate keys and more than one separate key can point to the same record.

If the index is deleted and recreated, the `create_index` command only recreates the embedded keys.

Item Indexes

An item index is associated with a file, which can be empty. The index keys are not related to records in the file. Instead, each index key is associated with four bytes that are stored as part of the index itself. The file is simply used to provide a path name for `s$attach_port` and to provide access control.

When an application reads from the item index the system does not perform file I/O and return a record. Instead, it immediately returns the four byte value stored in the index.

The subroutines `s$add_item`, `s$get_item`, and `s$delete_item` are used in working with item indexes.

The system dictionary is implemented as an item index. The keys are the English words in the dictionary, and the items are bit strings indicating where the associated words can be hyphenated. The latter feature is used by the Word Processing Editor.

Deleted Record Indexes

Normally, the space occupied by deleted records is not reclaimed. That is, if you delete a record from a file and later write a record to the file, the new record is placed at the end. It does not overwrite the deleted record. A *deleted record index* provides a way for files to automatically reclaim space from deleted records. The `create_deleted_record_index` command (or the `s$create_deleted_record_index` subroutine) creates an index named `_deleted_record_index`. This index maintains a list of all deleted records in the file. When a record is added to the file, the index is searched. The record is added to the end of the file only if no deleted record space is available. This is true if you use the write routines from various operating system languages, if you use `s$rel_write` with a record number of 0 or negative, or if you use `s$seq_write` or `s$keyed_write`. Record numbers are not guaranteed to increase sequentially as you add new records. That is, suppose record number 4 is deleted and a new record is written to the file. In a fixed or relative file, record 4 is immediately reused for the new record. In a sequential file, record 4 is reused only if the new record is the same size as the old record (or one byte smaller if the new record has a size that is an odd number and the old record has an even size). If the record is reused, the new record will physically occupy record number 4 in the file and be randomly accessed as record 4.

Since you cannot delete records in a stream file, you cannot define a deleted record index on a stream file.

To delete a deleted record index, use the `delete_index` command.

Record Indexes

A *record index* acts like a deleted record index with one important difference: the record number returned is always incremented. The `create_record_index` command (or the `s$create_record_index` subroutine) creates an index named `_record_index`. The record index associates a record number with each record in the file, regardless of the file type. When the record index is created, `create_record_index` reads through the file sequentially, assigning record numbers to already existing records. Any future additions to the file will automatically be assigned a record number in the record index. These record numbers are always increasing. In addition, a record index keeps track of the location and size of deleted records, so that the deleted space can be reused. Since new records are assigned increasing record numbers and all record accesses go through the record index, records will appear to be in logical order, regardless of their physical ordering within the file. The tradeoff is that it is a more expensive access method than random or sequential access on a file without a record index, because all I/O to the file is channeled through this index.

Since you cannot delete records in a stream file, you cannot define a record index on a stream file.

You cannot delete a record index. To remove the index, copy the file to a pre-existing file, specifying the `-truncate` argument.

Creating Indexes

The `create_index` command is used to create an index on a file. If the file contains data and you create an embedded-key index, the index keys are created for each of the data records. For an embedded-separate key index, the embedded keys are created unless you specify otherwise. The application must add all separate keys. When creating an embedded or embedded-separate key index, you must specify the position and length of each key component within the record.

Indexes can also be created using the `s$create_index` subroutine.

When you create an index, you must specify on what basis the keys are to be sorted. Collation options are:

- ASCII
- alphabetic
- numeric
- ASCII varying
- alphabetic varying
- numeric varying.

In the latter three cases, you can specify only one component, and it must specify the position and length of the character varying field in the record. Note that in all cases the keys are stored in the index as characters. Numeric keys are character strings that are converted to and treated as numbers when they are used.

You must also specify whether duplicate keys or null keys are allowed.

If you do not want an embedded-key index to be updated each time you write to the file, you can specify `-no_automatic_update` when you issue the `create_index` command. You can also turn automatic update on or off with the `s$set_index_flags` subroutine. This enables you to turn off automatic updating at certain times (thereby increasing performance). You can then recreate the index at a later, less busy time. When the file is not in use, `s$set_index_flags` also allows you to turn automatic updating on or off.

The `create_index` command and the `s$create_index` subroutine are also used to create item indexes.

The `create_deleted_record_index` command or the `s$create_deleted_record_index` subroutine are used to create a deleted record index.

The `create_record_index` command or the `s$create_record_index` subroutine are used to create a record index.

Index Structure

All indexes are modified B-trees. A non-empty index contains at least two blocks: the first free index map block and an index block serving as both a root and a leaf of the tree.

The free index map block (level 0 of the tree) contains information on the size and version number of the index, as well as which allocated blocks are used or free. If the index is larger than 32,752 blocks, block 32753 will be a second free index map block.

In addition to the map block, the index can have three other types of blocks.

- The *root block* is the base of the tree. If the index has only two blocks, this block will also serve as a leaf. Otherwise, it will point to the branch block (if they exist) or to the leaf blocks.
- The *branch blocks* point to the leaf blocks.
- The *leaf blocks* of the tree consist of a block of keys and the corresponding offset or record number. They are always labeled as level 1.

As noted, an index at minimum occupies two blocks: the map block and block that serves as root and leaf. As the index grows, it expands to four blocks: the map block and a root block which points to two leaf blocks. As the index grows, blocks continue to split. When the index grows beyond a certain size, the split will result in a new level for the index. It will then have root, branch, and leaf blocks. Figure 5-4 shows an example of an index tree and how it grows.

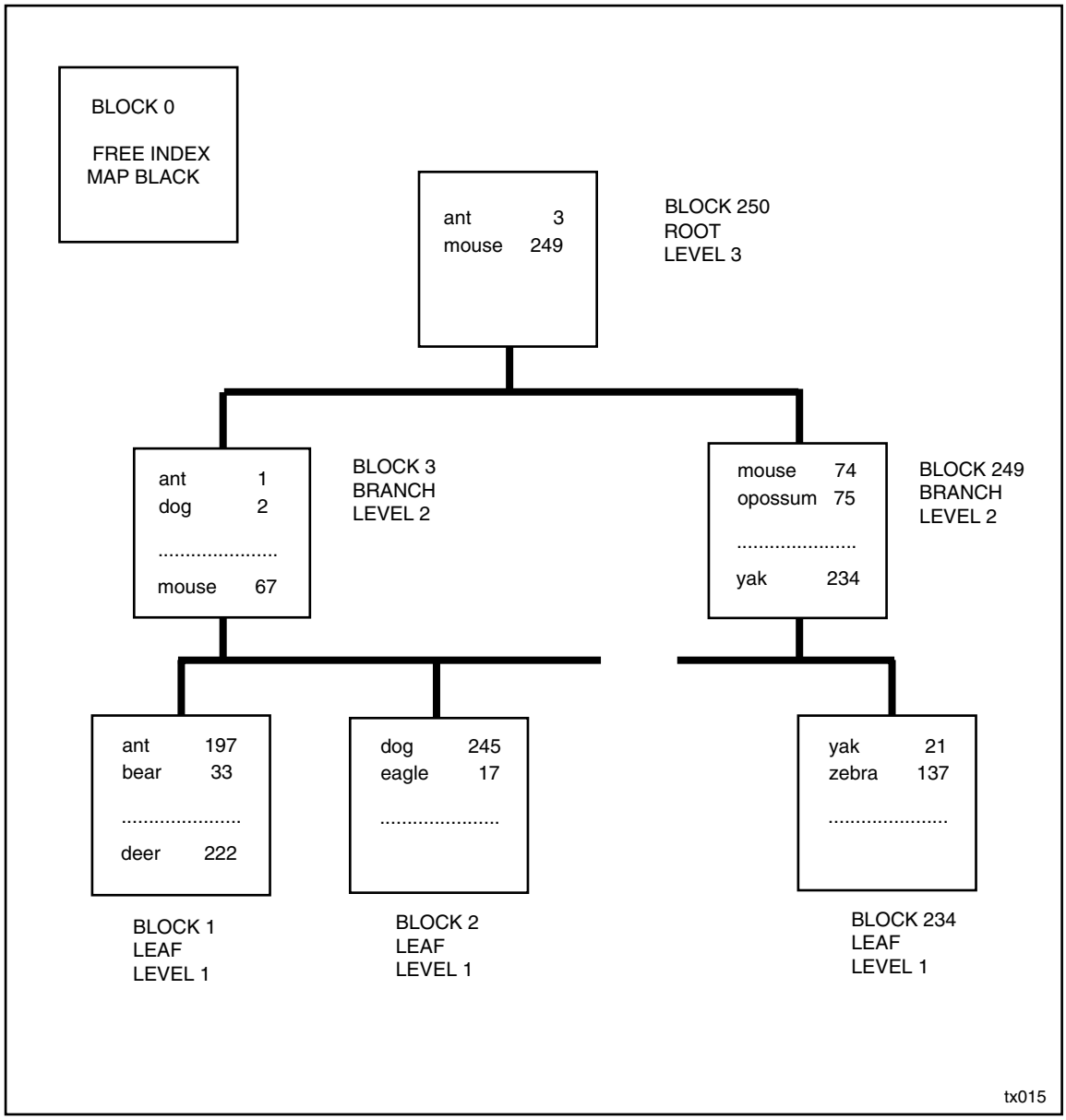

Figure 5-4. Index Tree Structure

An index block has two areas: the header area and the storage area. The storage area, in turn, is further divided into two parts: the key area, and the string area (see Figure 5-5).

The header takes up the first 20 bytes. It contains information about the block and information about the key and string areas.

The key area is allocated forward from the beginning of the storage area. All entries in the key area are the same size, either four or six bytes. Each key in the key area consists of two parts: the position of the key string in the string area of the index block and the record location. The position of the key string is two bytes long. The record location is either two

bytes or four bytes long, depending on the record offset size or the record number size. If all sizes can fit into two bytes, all record locations in the block are two bytes long. Otherwise, all record locations in the block are four bytes long. The header indicates which size is used in the block.

When an entry is added to the index, the string is added to the string area at the next available location. The key is added to the key area in order, thus allowing binary searches of the index. Duplicate keys in an index block point to the same string.

If the block is a leaf block, the record location points to the actual location of the record in the file. For a sequential or stream file, the record location is the byte offset of the record. For a relative or fixed file, the record location is a record number. If the block is not a leaf block, the record location points to the next block in the tree.

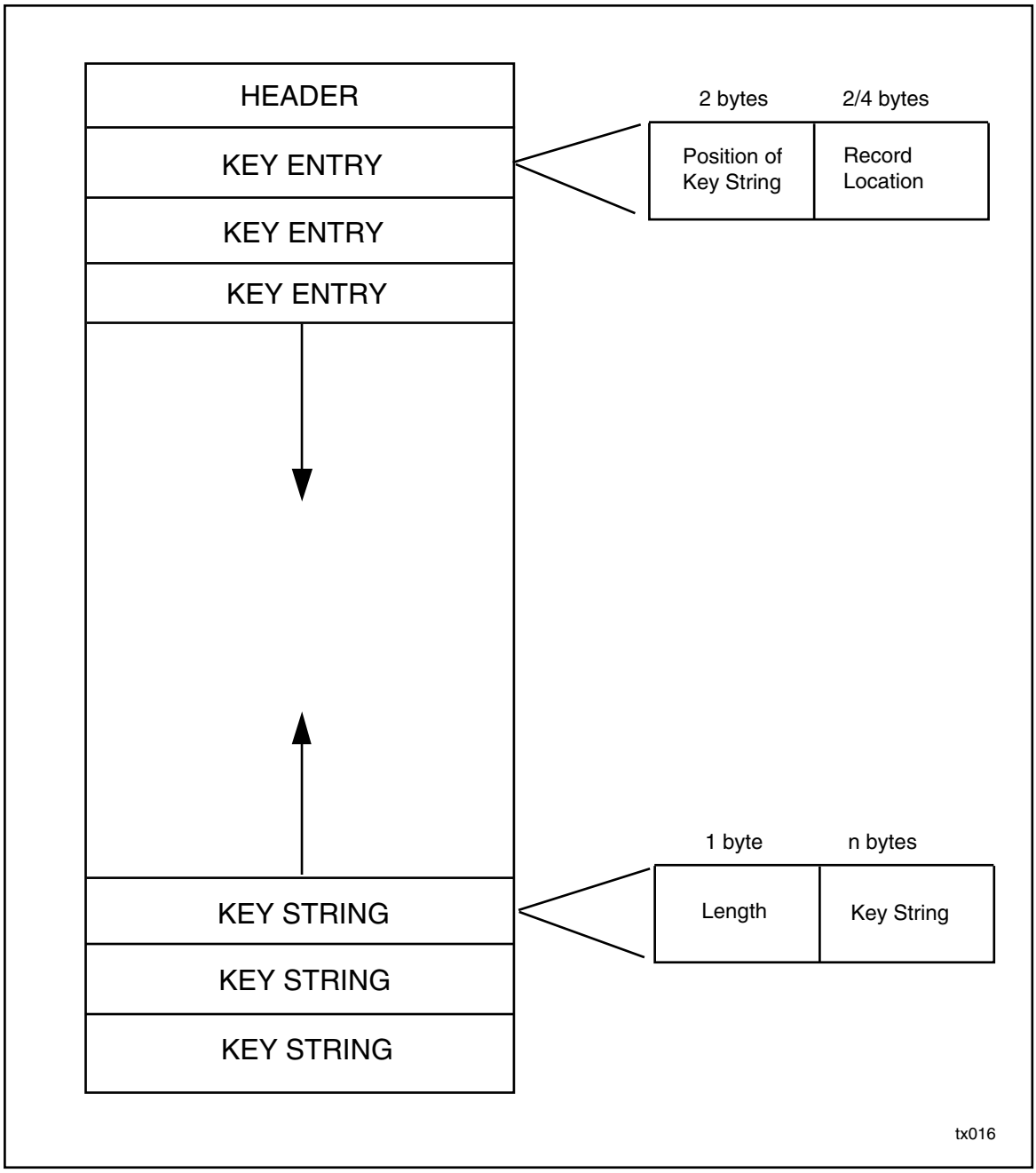

Figure 5-5. Index Block

The string area is allocated backward from the end of the storage area in the index block to the key area. The string is stored with a leading byte which contains the string's length and with any trailing spaces removed. For example, assume that you are using an embedded-key index that indexes on the first ten characters of the first name field. If the first name is `Jim`, then only `Jim` is stored, not `Jim` followed by seven spaces. The leading byte would specify the string length of `3`.

File Extents

A file *extent* is one or more contiguous disk blocks. When you create a file, you can specify the extent size (using either the `-extent_size` argument to the `create_file` command or the `s$create_extent_file` subroutine to create the file). A file that uses extents is called an *extent file*. Extent sizes must be a multiple of 8 blocks. (See the *VOS Commands Reference Manual (R098)* or the VOS Subroutines Manuals for more information.)

Based on the specified extent size, number of records, and record size, one or more extents are allocated when the file is created. The address of each extent, not the addresses of the individual blocks making up the extent is stored in the file entry of the directory (see the section "File Disk Storage"). Thus, the file entry can be much smaller for large files that use large extents. Furthermore, for non-extent files, the disk blocks that make up a file can be scattered around on a disk. This can slow down some I/O operations. However, all the blocks in an extent are contiguous, so I/O operations may be faster on an extent file.

Extents are only used for the initial allocation of a file. If the file grows beyond the size of the initially allocated extents, it grows by blocks (one or more blocks at a time, depending upon the value of allocation size (set by `set_file_allocation`). The address of each block allocated at this point is stored in the file entry. These disk blocks can be scattered around the disk.

For example, if you create a fixed file and specify an extent size of 1024 blocks, a record size of 256, and 32,000 records, two extents are allocated. (32,000 records, each 256 bytes long, requires 8,192,000 bytes, or 2000 blocks.) The file entry contains the addresses of these two extents. If the file grows beyond these two extents (beyond 2048 blocks), disk blocks are allocated one at a time and the address of each block is stored in the file entry.

For sequential and stream files, the number of records specifies the number of blocks to preallocate, since sequential and stream files do not have a fixed record size.

File I/O Operations

Before performing an I/O operation on any file, you must first attach a port to the file, then open the file. When you open the file, you must specify the type of I/O operation you are going to perform and the access method. (If you are using programming language I/O statements rather than the operating system subroutines, the operating system attaches the port for you when you open the file.)

The operating system supports the I/O types shown in Table 5-1.

Table 5-1. I/O Types *(Page 1 of 2)*

I/O Type	Description
input	The file is opened for reading only.
output	The file is opened for writing only. If a file with the specified name exists, it is truncated.

Table 5-1. I/O Types *(Page 2 of 2)*

I/O Type	Description
append	The file is opened for writing only. If a file with the specified name exists, any records you write are appended to the end.
update	The file is opened for reading or writing. You can read records, write new records, or rewrite records at existing positions.
dirty input	The file is opened for input. This differs from normal input in that you can read a file that has been write-locked by another process.
dirty notify	The file is opened for input. This is similar to dirty input, except you are notified if another process modifies the file.

For more information on the dirty input and dirty notify I/O types, see the section on "File Locking."

The current file *position* is the location in a file at which the next record operation will be performed. When you open the file, VOS sets the current position to the first record in the file, unless you specify an I/O type of append, in which case it sets the current position after the last record in the file.

Each port attached to a file has a *just-positioned switch*, which specifies whether the last operation on the file changed the file position but did not access the record. For example, `s$seq_position` changes the position but does nothing else, and thus sets the just-positioned switch. The just-positioned switch affects the next sequential read. For example, as explained under "Sequential Access," `s$seq_read` positions the file to the next record before reading the record if the just-positioned switch has not been set.

The *access mode* defines the way a record in the file is specified in an I/O operation. There are three methods for accessing data files: sequential, random (or direct), and indexed (keyed). Table 5-2 shows which methods are valid for which file types.

Table 5-2. File Access Methods

File	File Access Method
Organization	**Sequential**
Fixed	Yes
Sequential	Yes
Relative	Yes
Stream	Yes

All I/O types and access modes are available using VOS subroutine calls. These I/O types and access methods may also be available as part of the I/O syntax of the various programming languages (depending on the standard for that language). Table 5-3 summarizes the

subroutines used with each I/O type and access method. It also shows the effect of each of these subroutines on the file position.

Table 5-3. Subroutines Used for File I/O

Subroutine	Access Mode	Record Selection	Position Changed?
Input Subroutines: I/O Types: input and update			
`s$keyed_read`	indexed	key	yes
`s$rel_read`	random	record number	yes
`s$seq_read`	any	current record	yes
Output Subroutines: I/O Types: append, output, and update			
`s$keyed_write`	indexed	key	†
`s$rel_write`	random	record number	no
`s$seq_write`	any	end-of-file	no
`s$seq_write_partial`	any	end-of-file	no
Deletion Subroutines: I/O Types: update			
`s$keyed_delete`	indexed	key	no
`s$rel_delete`	random	record number	no
`s$seq_delete`	any	current record	no
Positioning Subroutines: I/O Types: input and update			
`s$keyed_position`	indexed	key	yes
`s$rel_position`	random,sequential	record number	yes
`s$seq_position`	sequential	‡	yes
Rewrite Subroutines: I/O Types: update			
`s$keyed_rewrite`	indexed	key	§
`s$rel_rewrite`	random	record number	no
`s$seq_rewrite`	any	current record	no

† Changes position depending upon whether the key was supplied. See the section “Indexed Access.”
‡ There are several ways to determine the position. See the VOS Subroutines Manuals for more information.
§ Changed unless the file was just positioned.

Sequential Access

Any file can be accessed sequentially. Sequential access allows movement through the file in a forward or backward direction. You can read records in a sequential file one after the other.

You can also position to the beginning or end of the file or forward or backward a specified number of records. When using stream files, it is also possible to go forward or backward a number of bytes. You **cannot** access a record in a file opened for sequential access by specifying a record number or an index key.

All of the programming languages support sequential access, and in most cases this is the default. For more details, see the language reference manuals for the specific language. The remainder of this section discusses sequential access in terms of the subroutine interface. See the VOS Subroutines Manuals for more details on the subroutines.

To open a file for sequential access you can use `s$seq_open` if it is a sequential file. This subroutine attaches a VOS port and then opens the file for sequential access. Alternately, you can use `s$attach_port` and `s$open` to attach the port and open the file.

To read a record, use `s$seq_read`. If the just-positioned switch is set and the file is positioned to a nondeleted record, `s$seq_read` reads that record. Otherwise it reads the first nondeleted record after the current position. If `s$seq_read` successfully reads a record, it positions the file to the record it last read and resets the just-positioned switch.

To change the file position, use `s$seq_position`. You can position to the beginning or end of the file or forward or backward a specified number of records. You can also use `s$seq_position` to set the just-positioned switch without changing the file position by specifying that it position forward or backward 0 records.

To write a record, use `s$seq_write` (or `s$seq_write_partial`). Where the record is written depends upon whether a record or deleted record index exists on the file. If there is a deleted record or record index on the file and space of the required size is available, the new record is written to this location. Otherwise, it is written to the end of the file. The current position in the file is not affected.

To rewrite a record, use `s$seq_rewrite`. This subroutine rewrites the current record (the record just read or positioned to). The rewritten record must be exactly the same size as the original record for a fixed, a sequential, or a stream file, and the rewritten record must not be larger than the maximum record size for a relative file.

To delete a record, use `s$seq_delete`. This subroutine deletes the current record. This does not change the current position of the file. You cannot delete a record in a stream file.

Sequential I/O can also be performed on a stream file using `s$read_raw` and `s$write_raw` (as well as `s$seq_read` and `s$seq_write`). These subroutines read and write at the current position in the file, ignoring line-feed characters. They read or write the number of characters specified and thus can read/write partial records or multiple records.

Random Access

Random (or direct) access allows the user to access records in a file randomly using the record number. Only relative and fixed files can be randomly accessed. With the exception of `s$rel_write`, all subroutines used with random access require that you supply a record number. Direct accessRandom access

To read a record, use `s$rel_read`. This subroutine returns the record and the record length. If the read is successful, the file is positioned to the record just read and the just-positioned switch is set.

To write a record, use `s$rel_write`. If you specify the number of a record that already exists in a relative file, an error results. In a fixed file, the record is overwritten, and no error message if returned. If you specify a record number as zero or negative, where the record is written depends upon whether a deleted record index exists for the file. If a deleted record or record index exists and deleted space is available, the record is written to the location of a deleted record. Otherwise, the record is written to the end of the file. The length of the record must be equal to the record size in a fixed file. In a relative file, the length can be less than or equal to the maximum record size.

To change the file position, use `s$rel_position`. You can position to:

- a specified record number
- a specified record number or the next undeleted record if that record has been deleted
- the next undeleted record after a specified record number.

To rewrite a record, use `s$rel_rewrite`. However, `s$rel_rewrite` cannot rewrite a deleted record in a relative file. It does not change the file position.

To delete a record, use `s$rel_delete`. As described above under "VOS File Formats," this has no effect on a fixed file without a record index, although any embedded-key indexes are updated to show that the record has been deleted.

Indexed Access

Indexed (or keyed) access can be performed only on files which have an associated embedded-key, separate-key, or embedded-separate-key index. Refer to "File Indexes" for more information on index types. Keyed accessIndexed access

When you open a file for indexed access, you must specify an index name. This index becomes the file's *primary index*. It is also initially the file's *current index*, which is the index used in the most recent successful keyed I/O operation. As such, the current index can be changed by subsequent keyed read or keyed position calls. If you open a file for indexed access and do not specify an index name (that is, specify a null string as the index name), `s$open` uses the default index name of `primary`. If no index named `primary` exists but some index exists on the file, the file is opened with no current and no primary index. If you specify the name of an index that does not exist, or if there are no indexes on the file, an error code is returned and the file is not opened.

To read a record, use `s$keyed_read`. If you do not specify an index, the current index is used. If you specify an index name and the read is successful, the specified index becomes the current index for subsequent I/O operations. If the index allows duplicate keys, the call to `s$keyed_read` only returns the first record with the specified key. To return subsequent records, use `s$seq_read`. You can also use `s$seq_read` to read the records of the file in index order.

To write a record, use `s$keyed_write`. Where the record is written depends upon whether a record or deleted record index exists on the file. If there is a deleted record or record index on the file and space of the required size is available, the new record is written to this location. Otherwise, it is written to the end of the file. This subroutine has both a record and a key argument. You need not specify a key for an embedded-key index. VOS extracts the key from the record. However, if the primary index is a separate-key index, you **must** specify the key. `s$keyed_write` updates the file's embedded-key and embedded-separate-key indexes, unless they have been set for no automatic update. If the file's primary index is a separate-key index, it also adds a key to this index. You must use `s$add_key` to update separate-key indexes other than the primary index.

You can also use `s$seq_write` to write a record to the file. Only the embedded and embedded-separate-key indexes will be updated. You must still use `s$add_key` to update any separate-key indexes.

If you supply the null string as a key, `s$keyed_write` does not change the current position in the file. If the given key is not the null string, and if `s$keyed_write` successfully writes a record, the subroutine changes the current index to the primary index, positions the file to the record it writes, and resets the port's just-positioned switch.

To rewrite a record, use `s$keyed_rewrite`. It selects the record to rewrite based on the file's primary index. The rules for index updating and file positioning are the same as for `s$keyed_write`. You can also use `s$seq_rewrite` if you are already at the position you want to rewrite.

To delete a record, use `s$keyed_delete`. This subroutine deletes a record using the file's primary index and given key value. It deletes the first instance of the key given, and deletes the key from the primary index and any embedded-key indexes. To delete the key for the same record from other separate-key indexes, use `s$delete_key`. You can also use `s$seq_delete` if you are already positioned at the record you want to delete.

To position to a record, use `s$keyed_position`. If you do not specify an index, the current index is used. If you specify an index name and the read is successful, the specified index becomes the current index for subsequent I/O operations. This subroutine has both a record argument and a key argument. You can supply either argument, depending upon the type of index. For embedded key indexes, if you specify a record key, it is used to locate the record. Otherwise, the key is extracted from the record in the record buffer argument. For separate key indexes, you must specify the key. You can specify a relation to be used in positioning to a record. See the description of `s$keyed_position` in the VOS Subroutines Manuals for more information.

You can use the sequential access I/O subroutines (`s$seq_read`, `s$seq_position`, and so forth) on a file opened for index access. In this case, the file is accessed sequentially according to the order of the records in the current index, not the order of records in the file.

Data Structures Involved in File I/O

The operating system uses several data structures when performing file I/O. These structures are summarized in Figure 5-6.

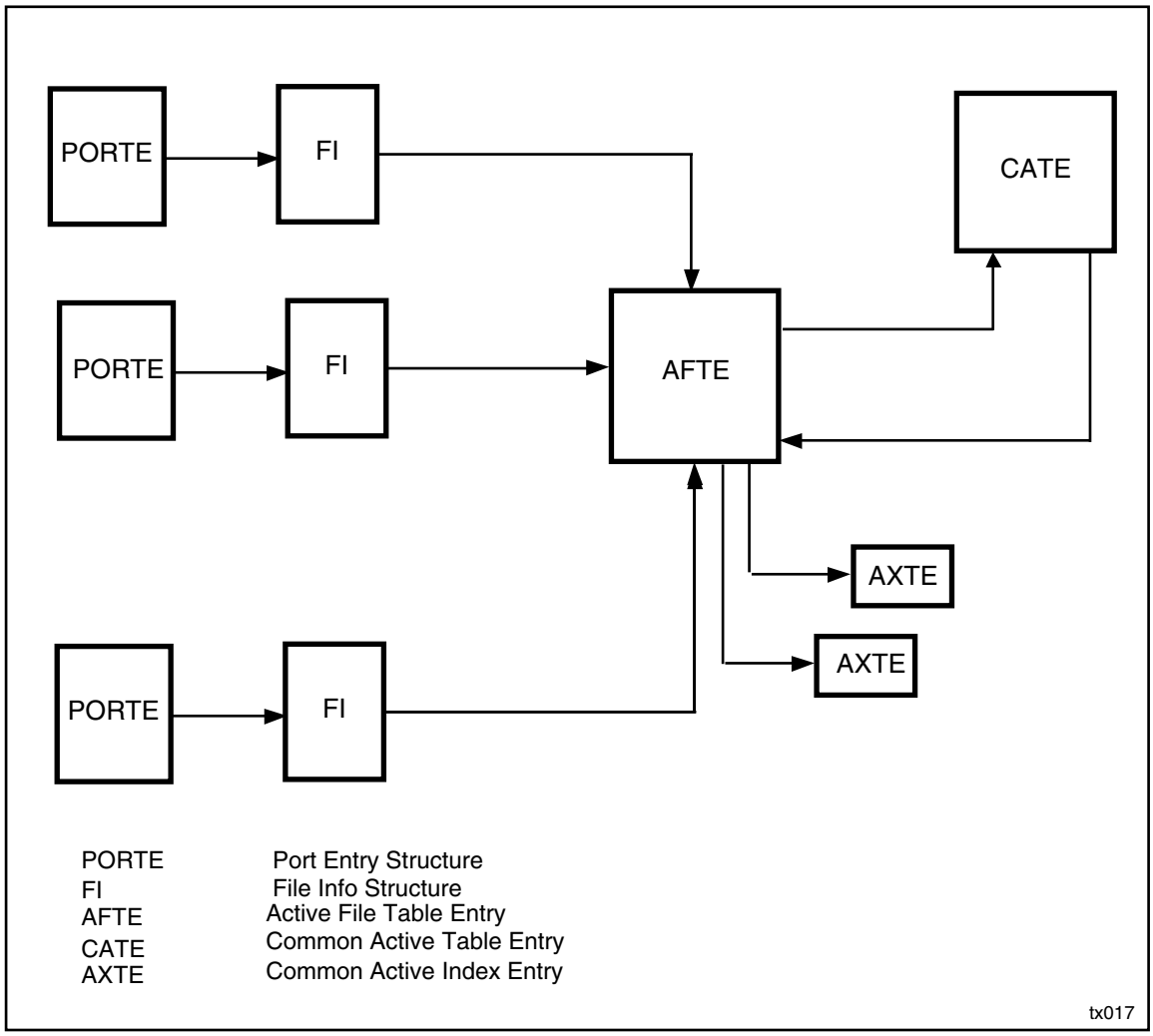

Figure 5-6. Data Structures Involved in File I/O

When an application attaches a port, information about the port is maintained in the port entry (PORTE) structure. The PORTE structure contains information about the port, about the file it is attached to, and about the file access modes and I/O type, if the file is open. PORTEPort entry structure

When the first process opens the file, a structure called the active file table entry (AFTE) is created. The AFTE maintains information relating to all processes using the file. It includes information on file type and size as well as the number of readers, writers, and so forth. The AFTE also contains a pointer to the common active table entry (CATE) which contains information about the disk addresses of the file blocks. AFTEActive file table entry CATECommon active table entry

For each process that opens the file, a second structure called the file information (FI) structure is created. The FI structure contains information about the state of the file, such as

which block number is being accessed, the current file position, the type of locking used, and a list of records that are locked. FIFile information structure

If the file has indexes, the AFTE also contains pointers to one active index table entry (AXTE) for each index. The AXTE includes information about the root block number of the index, the collating type, and the components (for embedded keys). AXTEActive index table entry

File Locking

Before accessing data, an application must lock the data, either at the file level, record level, or region level. When a process calls `s$open` to open a file, the locking procedures are determined for that process and operate on a per port basis. By carefully selecting the most appropriate file locking modes, concurrent use of files can be permitted without creating long wait times for some of these processes.

There are six levels of locking, which permit varying degrees of control over the files. Lock mode values and their names are shown in Table 5-4.

Table 5-4. Locking Types

TYPE	s$open LOCK VALUE
`implicit_locking`	4
`file locking`	
`set_lock_dont_wait` †	1
`wait_for_lock`	2
`dont_set_lock`	3
`record_locking`	5
`region locking`	6

† default locking mode

A file may have only one locking type in effect at any given time. For example, if a process opens a file with record locking, a second process cannot open that same file and request any other file locking mode. The second process could only use that file if it also specifies record locking.

The processes that have locked a file and the locking mode can be identified by issuing the command `who_locked` or by using the subroutine `s$get_lockers`.

The functional result of specifying a lock mode depends on other options you specify when opening the file. The other choices that effect the lock handling are the file type and the I/O type (`input`, `output`, `append`, `update`, `dirty input`, `dirty notify`). The effects of these combinations are presented in several tables which can be found in the VOS Subroutines Manuals where the `s$open` statement is described. For example, if the file type is relative, the lock mode is `wait_for_lock`, and the I/O type is `input`, the operating system waits until it can lock the file. If the I/O type is `output`, the operating system attempts

to lock the file. If it cannot be locked at that time, an error code indicating that the file is in use is returned to that process.

Locking modes are used differently for pipe files than for other file types. Because pipe files are used to communicate between processes, the lock modes of the companion processes must not conflict. The locking modes specified for pipe files control the synchronization of the processes **opening** the file. This contrasts with the control of **accessing** non-pipe files. See the description of `s$open` in the VOS Subroutines Manuals for more information.

Implicit Locking

In implicit locking, the whole file is locked, but only for the duration of an I/O operation. For example, if a record is being read, the file is locked for reading, but only during that read. Implicit locking allows multiple readers and multiple writers to share the opened file. If you open a file for implicit locking, any other user who attempts to open with a different locking mode receives an error message indicating that implicit locking must be used. Conversely, if you try to open a file for implicit locking when the file has already been opened by another process with some other locking mode specified, you will also receive an error.

If a file is opened for implicit locking, the operating system performs the following steps each time the process performs an I/O operation on the file:

1. locks the file
2. performs read, write, rewrite, position, or delete
3. unlocks the file.

Locking modes are generally specified when you open the file. However, you can override the open statement specification to limit processes' access to only the implicit mode of locking. To do so, use the `set_implicit_locking` command or the `s$set_implicit_locking` subroutine. In this case, regardless of what locking mode is specified in the open statement, implicit locking mode is used by default. When implicit locking is forced on a file, processes that attempt to open the file in non-implicit mode will **not** receive an error code when they call `s$open`, and the processes will use the file in the implicitly locked mode.

Explicit File Locking

Files may be explicitly locked in three modes: set-lock-don't-wait, wait-for-lock, and don't-lock. Any of these modes allows multiple readers **or** one writer when file locking is set.

In set-lock-don't-wait mode, the process opening the file tries to lock the file immediately. If it cannot, the call to `s$open` returns the error code `e$file_in_use`. If the I/O type is specified as `input`, the file is locked for reading and multiple readers can read from the file. For any other I/O type, the file is locked for writing.

In wait-for-lock mode, the process opening the file waits until it can lock the file, unless it is opening the file for output. In this case, `s$open` returns immediately with the error code `e$file_in_use`.

In don't-set-lock mode, the user opens the file without locking. Because files must be locked to be accessed, you must call `s$lock_file` before reading or writing to the file. If this call is successful, the file is locked. If not, it will wait a set amount of time (the user-specified lock-wait time), and try again to lock the file. (For more information, see the section "Waiting for Locks.") When the I/O completes, call `s$unlock_file` to release possession of the file. This locking type allows several operations to be performed on the file with the assurance that no one else has changed the file between updates.

Record Locking

If record locking is used, individual records are locked, not the entire file. This gives you exclusive use of some records, while allowing other users to access other records in the file. You must unlock the records when you finish with them. The only two valid I/O types used with record locking are `update` and `input`.

If the I/O type is `update`, any record that is read or written is locked automatically and the record must be explicitly unlocked using `s$seq_unlock_record` (if the record is the current record), `s$rel_unlock_record` (to unlock a record chosen by record number), `s$keyed_unlock_record` (to unlock a record chosen by key value), or `s$unlock_records` (to unlock multiple records). If the I/O type is `input`, any record that is not locked can be read, but no records can be locked.

Region Locking

Region locking is used to lock regions of stream files. You specify which bytes of the stream record are to be locked.

Four subroutines are used for region locking on stream files: `s$lock_region`, `s$unlock_region`, `s$get_region_lockers`, and `s$enforce_region_locks`.

There are two modes of region locking: advisory and mandatory. *Advisory locking* is in effect unless mandatory locking has been invoked by `s$enforce_region_locks`. When advisory locking is in effect, multiple processes can lock a region for reading at the same time, but only one process can lock a region for writing. It is essential that the locking status of the file be analyzed before issuing a lock. Otherwise, deadlock potential exists when two processes wait for each other to unlock their locked regions before continuing processing.

When *mandatory locking* is in effect, multiple processes cannot lock, read, or write to the locked area until that area is unlocked. If the operating system recognizes a deadlock potential, an error is returned. When mandatory locking is used, the application does not need to call `s$lock_region`. In this case, the first process which does call `s$lock_region` prevents all other processes from accessing that area.

Waiting for Locks

You can set the time limit on how long a process will wait for a lock. This limit is called the *lock-wait time*. The lock-wait time is, by default, 0 seconds, but it can be reset. The time can be determined or set for the module by a privileged process using the subroutines `s$get_default_lock_wait_time` or `s$set_default_lock_wait_time`, or using the commands `display_lock_wait_time` and `set_lock_wait_time`.

You can set the lock-wait time for an individual process or an individual program using the subroutines `s$get_lock_wait_time` and `s$set_lock_wait_time`, or the commands `set_process_lock_wait_time` and `display_process_lock_wait_time`.

The system uses lock-wait time set for the specific program, if one has been set. Otherwise, it uses the lock-wait time for the process. If none has been set, it uses the module's default lock-wait time.

Directories

A directory is a catalogue of the names and locations of files, links and other directories (generically referred to as *objects*). For convenience, you can put a set of objects that are logically related in one directory.

You can create your own directories and control the objects in them. There are two rules, however:

- A file can be in only one directory.
- The name of an object in a directory must be unique among the set of names of files, links, and subdirectories in that directory.

Different directories can contain objects that have the same name. The path names of these objects refer to different objects because they are in different directories. To specify an object unambiguously, you must specify the full path name of the directory as well as the name of the object (see the section "Path Names").

The attributes of an object are contained in the directory with the name of the object and its location. The collection of information about an object in a directory is called its entry. The attributes of an object that are contained in its entry include:

- its name
- the date and time it was created
- the date and time it was last used
- the date and time it was last modified
- the date and time it was last saved
- its access control list
- its default access control list, if it is a directory.

A directory can contain the names and attributes of other directories. As with any other object, a directory can be included in only one directory. This restriction imposes a hierarchical structure on the set of directories. In the hierarchy, one directory is immediately superior to another if it contains the other. A directory immediately contained in a directory is called a *subdirectory.*

The storage for a directory and all of its entries is contained on one logical disk. For each disk in the system, the operating system creates a directory to manage the directories on that disk. This topmost directory on a disk, called the *root parent directory*, is usually referred to by the name of the disk. All other directories on the disk are subordinate to this directory.

The unique chain of directories going from the topmost directory down to a lower directory is called the path name of the directory.

The system administrator controls the access to a top directory and its subdirectories. See the section “Access Control,” for more information.

When a file is to be used, it and its parent directories must be *activated.* Activation involves bringing the information from the disk blocks into the memory entries in the wired and paged heaps. All parent directories must be activated when a file is brought into memory. Information about active directories is maintained in the active directory table (ADT). There is one entry in the table (called an ADTE) for each active directory. ADTActive directory table ADTEActive directory table entries

If an active directory or its subdirectories are not used for a specified time, it times out and the space for its ADTE is freed. This time is established with the `set_tuning_parameters` command using the `-unused_dir_timeout` argument. The root parent directory stays active as long as the disk is mounted.

Current Directory and Home Directory

Whenever you are using a Stratus system, the operating system remembers the names of two directories for you, a current directory and a home directory. At times, your current directory will be your home directory; at other times, they will differ. For more information, see Chapter 3, “Processes and Interprocess Communication.”

System Directories and Group Directories

The *system directory* (`>system`) is found on the master disk on a module. Every processing module contains a master directory that contains a subdirectory named system. It contains the files and tables the operating system needs. It also contains subdirectories containing various programs, tools, and commands used by both the operating system and users, such as compilers, editors, and error message files. Usually a module has default library paths that include subdirectories of the system directory. (See Chapter 3 for information on library paths.)

As explained in Chapter 3, each user on a module is part of a group. A *group directory* contains the home directories of the members of the group. The name of the group directory is the same as the name of the group.

Path Names

The most important function of the directory hierarchy is to provide a way to uniquely but conveniently name any object in the I/O system. Any user on any processing module or system that can communicate with the module containing the object can then refer to the object.

The unique name of an object is derived from the object’s unique path in the I/O system. The unique name is called the *path name* of the object. A path name is constructed from the name of the object, the names of the directories in the path leading to the object, and the name of the system containing the root parent directory.

The path name of a file or directory is a combination of the following names:

1. the name of the system containing the object preceded by a percent sign (%).
2. the name of the disk containing the object preceded by a number sign (#)
3. the names of the directories in the path of the object, in order, each preceded by the greater-than sign (>)
4. **t**he name of the object preceded by the greater-than sign (>).

The symbol > is used to separate directories and files in the path name. Its use is similar to the use of `/` or `\` in other operating systems.

For example, suppose you have a system named `%s` containing a disk named `#d01`. (The module containing the disk is `%s#m1`.) The following is an example of a full path name for the file named `this_week`.

```
%s#d01>Administration>Jones>reports>this_week
```

The file is immediately contained in the directory `reports`, which is subordinate to the directory `Jones`. The home directory `Jones` is a subdirectory of the group directory `Administration` which is a subdirectory of the disk `#d01`.

Relative Path Names

The path names defined so far are full path names. The full path name of an object is unique because the path of an object is unique. The operating system can also interpret relative path names. A *relative path name* is a combination of object names and special symbols, like a full path name, that identifies an object in the directory hierarchy. A relative path name of the object generally does not contain all the directory names that are in the full path name. When you use a relative path name, the operating system determines the missing information about the object's location from the location of the current directory.

If the operating system reads a string that it expects to be a path name and the leading character is not a percent sign, it interprets the string as a relative path name.

The single character < can be used to refer to the parent directory of the current directory. For example, the command `change_current_dir <` moves you up one directory in the directory hierarchy. A single period (`.`) also refers to the current directory and two periods (`..`) refers to the parent directory. Thus, `change_current_dir ..` is the same as the `change_current_dir <`.

Links

An object can be in only one directory. However, a directory can contain one or more links to objects that are contained in other directories (or the same directory). A *link* is a directory entry that converts a reference to an object in the directory to another object, which usually is contained in another directory.

Any command that operates on files or directories can be given the path name of a link instead of the path name of a file or directory. In that case, the command will operate on the target file or target directory of the link.

Access Control

Access control is a mechanism that allows you to define which users can read and write files and directories. It is a way to protect data, and, at the same time, to selectively share the data with other users. However, the access control mechanism is designed so that you can completely ignore it once the default access list has been set up. Thereafter, files and directories automatically acquire the default access list.

File Access Rights

There are four types of access rights a user can have to a file. They are described as follows:

- *Null access* means a user has no access to the file.
- *Execute access* means a user can execute a program module or a command macro, but cannot read or write the file.
- *Read access* means a user can read the file or execute it if it is executable, but cannot write it.
- *Write access* means a user can execute, read, and write the file.

Directory Access Rights

There are three types of access rights a user can have to a directory. They are described as follows:

- *Null access* means a user has no access to the directory.
- *Status access* means a user can display information about the directory (using commands such as `list` and `display_file_status`), but cannot modify the directory by creating, deleting, or renaming objects.
- *Modify access* means a user has full access to the contents of the directory, including the ability to create, delete, and rename objects.

Access Control List Entries

Access to files and directories is defined by entries in system files called *access control lists* (ACLs). An entry in an access control list has two parts: ACLAccess control lists

- an *access right*, which specifies what kind of access the set of users has to a file or directory
- a *user name*, which specifies a set of one or more users having a specified type of access right to a file or directory.

User Names

A user name has the following form:

```
person_name.group_name
```

A user name can represent an individual user or multiple users. A user name that can represent one or more users is called a *generalized user name*. In a generalized user name, either component is an asterisk or both components are asterisks. There are three forms that a generalized user name can have, as follows:

- `person_name.*` specifies the set of all users with the same person name, regardless of group names.
- `*.group_name` specifies the set of all users in the same group, regardless of person names.
- `*.*` specifies the set of all users, regardless of person names or group names.

An individual's user name is said to *match* a generalized user name in the following circumstances:

- When the generalized user name is in the form `person_name.*`, the individual's user name must have the same person name.
- When the generalized user name is in the form `*.group_name`, the individual's user name must have the same group name.
- When the generalized user name has two asterisks, all user names are matches for it.

How Access Control List Entries Are Sorted

Entries in an access control list are sorted in order of increasingly inclusive user names. As a result, the order of generalized user names in an access control list is as follows:

- those with no asterisks
- those in the form `person_name.*`
- those in the form `*.group_name`
- the generalized user name `*.*`.

Types of Access Control Lists

The three kinds of lists that define access control are:

- File access control lists
- Directory access control lists
- Default access control lists.

These are described in the next three subsections. The fourth subsection describes how the operating system uses these lists to determine a user's access.

File Access Control Lists

A *file access control list* is associated with a file and contains an ordered set of entries that define the access rights of users to that file.

The operating system maintains a file access control list for every file in the system. The entries in a file access control list are created in the following ways:

- When you create a file, the operating system gives it an empty access control list.
- You can add, change, or remove entries from the access control lists associated with the files in any directory to which you have modify access.

Directory Access Control Lists

A *directory access control list* is associated with a directory and contains an ordered set of entries that define the access rights of users to that directory.

The operating system maintains a directory access control list for every directory in the system. The entries in a directory access control list are created in the following ways:

- When you create a directory, the operating system gives the directory a copy of the access control list of its containing directory.
- In any directory to which you have modify access, you can add, change, or remove entries from the access control lists associated with its subdirectories.

Default Access Control Lists

A *default access control list* (DACL) is associated with a directory and contains an ordered set of entries that define the access rights of users to **files** in that directory. DACLDefault access control lists

The operating system maintains a default access control list for every directory in the system. The entries in a default access control list are determined in the following ways:

- When you create a directory, the operating system gives it a copy of the default access control list of its containing directory.
- In any directory to which you have modify access, you can add, change, or remove entries from the default access control list associated with it.

How the System Determines Access

When a user tries to use a file or directory, the operating system checks entries in the appropriate access control list for a matching user name.

In an access control list, no user name can appear more than once. However, a user name can match more than one generalized user name. Further, each generalized user name matching an individual's user name can be paired with a different access right. Therefore, the operating system uses the following method to determine a user's access:

- searches the appropriate access control list (starting at the top) for a user name that matches the individual's user name
- uses the access right paired with the first matching user name.

For a file, this means that the operating system does the following:

1. searches the file access control list for a user name that matches the individual user's name
2. uses the access right paired with the first matching user name
3. searches the containing directory's default access control list if no match is found in the file access control list
4. uses the access right paired with the first matching user name in the containing directory's default access control list
5. if no match is found in either the file access control list or the containing directory's default access control list, the user has undefined access, which is effectively null access.

For a directory, the operating system does the following:

1. searches the directory access control list for a user name that matches the individual user's name
2. uses the access right paired with the first matching user name
3. if no match is found in the directory access control list, the user has undefined access, which is effectively null access.

The Cache Manager

The cache manager operates between the file system software and the disk system. It maintains a pool of buffers in wired memory that is referred to as the *cache*. The cache stores recently accessed file blocks.

All I/O to the file system goes through the cache manager. When a process does a read or a write, the I/O goes to the cache manager, which checks to see if the requested file block is in cache. If the file block is in cache, it is immediately returned to the application. This is called a cache *hit*. If not, the cache manager reads it from disk into the cache, then returns it to the

user. This is called a cache *miss*. Since many disk I/O operations are satisfied by the cache, the amount of actual disk I/O that has to be done is reduced. This also saves processor time and reduces file lock times.

Modified blocks are written to disk and the blocks are freed when the file is closed. Modified blocks are also written to disk (though not freed) when the application performs a runout operation (by calling `s$control` with the `runout` opcode).

Two values establish the limits for disk cache physical memory -- the maximum number of cache buffers and the minimum number of cache buffers. The system also calculates another value which is based on a percentage of the total available system memory. The true maximum is then the lesser of either the established maximum or the calculated value. Thus the system will not use more than is reasonable, unless it is forced to do so by having the minimum number of disk cache buffers set higher than the calculated value. This number of cache buffers varies up and down with system activity, but it is limited by the maximum and minimum values defined. The default value for the maximum number of physical cache buffers is 2048.

A user can force the system to use more or less physical memory for the cache by setting the minimum and maximum buffer size appropriately. Setting the maximum and the minimum to values higher than the calculated value forces the operating system to use at least the minimum physical memory for the cache buffer. Setting the maximum and minimum to values lower than the calculated value forces the operating system to vary the physical memory for the cache between the maximum and minimum values. Use the `analyze_system` request `dump_cache_header` to see the actual number of physical cache blocks the system is currently using.

Fast File I/O

Fast file I/O is a method in which, under certain circumstances, the file block you are currently working on is stored in your address space, enabling you to bypass the cache manager and speed up performance. On the first access to a new block, you read it from the disk, through the cache manager. All subsequent accesses to records in that block are made to the copy stored in your address space (in the PDR heap).

The criteria listed in Table 5-5 must be met for a file to use fast file I/O. In addition, there must be enough space in the process's PDR heap to allocate the buffer. When these criteria are met, fast file I/O occurs automatically. You need not do anything special to enable it.

Table 5-5. Fast File I/O Criteria *(Page 1 of 2)*

Criteria	Allowed Values
File Organization	Sequential, Fixed, Relative, or Stream
Access Mode	Sequential
I/O Type	Input, append, output
Lock Type	Set-Lock-Don't-Wait, Wait-for-Lock

Table 5-5. Fast File I/O Criteria *(Page 2 of 2)*

Criteria	Allowed Values
Maximum Record Size	2048 for files accessed over the network. 512 (fixed and relative files) and no limit (sequential and stream files) for files on the same module.
Indexes	None
Transaction Protection	None

The Disk System

The VOS disk subsystem supports duplexed disks. Disks are arranged in *logical volumes*. Logical volumes are made up of one or more duplexed disks (that is one or more pairs of physical disks). Figure 5-7 shows an example of a logical volume. In this figure, the logical volume is named `#d01`. It is made up of two *member* disks: `#d01.0` and `#d01.1`. Each member, in turn, is duplexed. The names of the physical disk include the side, either primary or secondary: `#d01.0.pri`, `#d01.0.sec`, and so forth. *Primary* and *secondary* are naming conventions only. The primary disk is *not* necessarily the disk from which reads are performed, nor does it have any other special function.

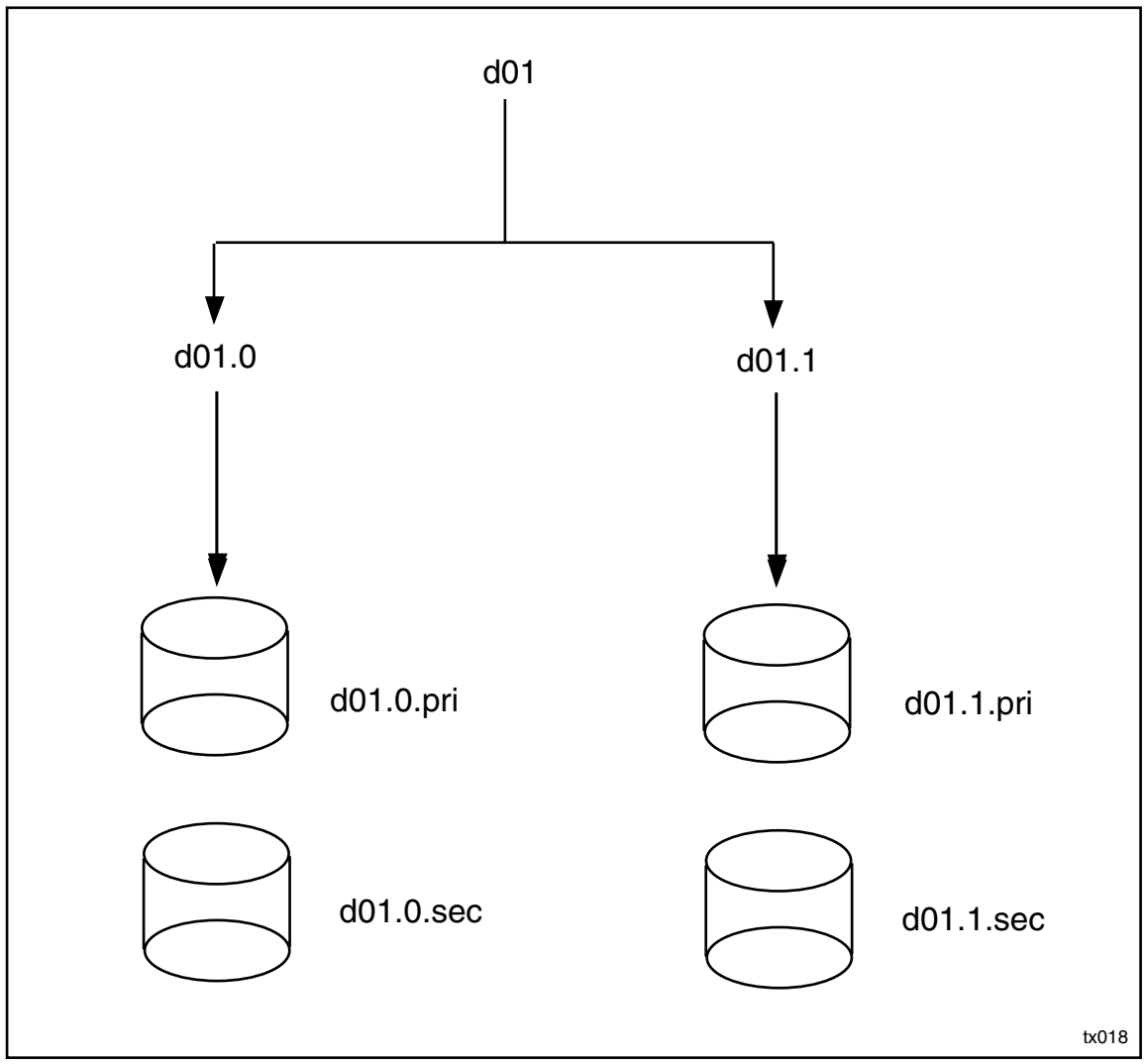

Figure 5-7. Disk Naming Conventions

Physical Disk Layout

The first block of any disk is the *disk label.* The label contains the disk name, the system and module names, various flags and times, the network address, and the address of the directory root parent for the disk.

The label is followed by the diagnostic, boot (if this is a boot disk), dump, paging, and file partitions.

The *diagnostic partition* occupies physical blocks 1 through 100 on a boot disk and blocks 1 through 10 on a non-boot disk. It contains information on bad blocks found on the disk. *Bad blocks* are areas of the disk that the operating system has marked as bad or unusable. Most disks have some bad blocks. For more on how bad blocks are handled, see the section below on “Bad Block Remapping.”

The *boot partition* holds current and past releases of the operating system. It begins in the blocks immediately following the diagnostic partition. These blocks are *indirect blocks*; that is, they hold pointers to other areas of the disk (in this case the file partition) rather than holding the information itself. The boot partition is made up of two blocks of pointers plus whatever space is needed from the area of the file partition.

The *dump partition* is used to hold dumps resulting from a system crash. It is arranged like the boot partition: two blocks of pointers to blocks taken from the file partition.

The *paging partition* is used by the page control software to hold temporary data pages. See Chapter 2 for more information on page control.

The *file partition* contains the storage area for all directories and files.

The *spare partition* and the *backup partition* are used for dynamic bad block remapping.

Bad Block Remapping

The operating system handles bad blocks by using *dynamic bad block remapping*. Dynamic bad block remapping uses two partitions on the disk: the spare partition and the backup partition. The *spare partition* is a pool of blocks the operating system uses to replace physical bad blocks on the disk. The *backup partition* is reserved for future use.

When a bad block is detected on a disk using dynamic bad block remapping, the operating system adds it to its list of bad blocks and remaps it to a block from the spare partition. (The physical bad block is replaced by a block in the spare partition.) Once the operating system uses a block from the spare partition, it does not recognize the bad block that was remapped to it and, therefore, sees a disk with no bad blocks. Duplex disks which use dynamic bad block remapping are logical mirrors of each other rather than physical mirrors.

Note that prior to Release 10 of the operating system, bad blocks could be handled by removing them from use on both partners. As of Release 10, however, all disks must use dynamic bad block remapping.

Disk Data Structures

LDTLogical disk table DDTDuplex disk table Three data structures are used to maintain information about disks: the logical disk table (LDT), the duplex disk table (DDT), and the disk data (DD) structure.

The LDT contains entries (called LDTEs) for each logical disk on the system. Each entry contains information about the name and location of the disk, the address of the root parent directory, and flags specifying whether the disk is local, whether it is mounted, and so forth. An LDTE has a reference to an entry in the DDT for each of its members.

Entries in the DDT (called DDTEs) contain information needed for duplexing, paging and file partition information, and information on the current status (such as how many requests are currently posted). A DDTE has a reference to an entry in the DD structure for each partner.

Entries in the DD structure (DDEs) contain information about the physical disk, such as the controller slot number, the drive number, the disk type, the current cylinder, and so forth. The entries also include flags indicating whether the disk is ready for I/O.

Disk Recovery

To prevent the inadvertent creation of different copies of data on the two members of a duplex disk pair, and to prevent loss of data, which can happen with automatic recovery, the operating system checks state variables at mount time to determine if the system can use the disk. State variables on the disk label indicate the last known paired state of the disk. The four possible states and their meanings are listed below.

- `null`, which means the disk is running under a previous version of the operating system.
- `duplexed`, which means the disks are duplex. No recovery is required.
- `recovering`, which means the disk is being recovered to.
- `running_alone`, which means the disk is running simplex or is being recovered from.

When checking state variables, the operating system deals differently with single disks than with a pair of disks. Both cases are described below.

1. Single disk. In this case, the system finds only one disk of a pair. If the last known state of the disk is the running alone state, the system uses the disk. If the last known state was the duplexed state and the disk had a partner that is no longer available, the system does not use the disk because the potential exists to create different copies of data.
2. Disk pair. In this case, the system finds both members of a pair. If the last known state of both members of the pair is the duplexed state, or if one of the pair is labeled as duplexed state and the other as running alone state, the system uses the disk. In all other cases, the system cannot determine which of the two disks is the most current, and the disks are not used.

If the system is unable to use a disk or disks, it prevents mounting the volume of that disk.

If the volume involved is the master volume (the volume that contains the master or boot disk), the system cannot be booted until the volume is complete. Operator intervention may be required to decide how to recover. See the *VOS System Administrator's Guide (R012)* for more information.

Chapter 6:
Programs and Program Execution

This chapter provides an overview of programming under VOS. It describes the steps you must follow in writing, compiling, and binding a program; programming language support; and system subroutines. It also describes stack frames, the calling sequence, memory allocation, error and condition handling, and the debugger.

Preparing and Executing Programs

To create a program to run under VOS, follow these steps:

1. Use a VOS editor to create an ASCII file containing the source code.
2. Use a compiler (such as C or PL/I) or the assembler to translate the source file into relocatable object code (an object module).
3. Use the binder to transform one or more object modules into an executable program module.
4. Use the debugger to test and debug the program.

Writing a Source Module

Using a VOS editor, write the program in one of the VOS programming languages: BASIC, C, COBOL, FORTRAN, Pascal, or PL/I. (You can also use 68000 assembly language.) The file you create is called a *source module*. The source module must have a suffix indicating the source language as shown in Table 6-1.

Table 6-1. File Suffixes for Source Modules

Programming Language	Suffix
Assembler	`.asm`
BASIC	`.basic`
C	`.c`
COBOL	`.cobol`
FORTRAN	`.fortran`
Pascal	`.pascal`
PL/I	`.pl1`

Compiling a Source Module

After you have created the source module, you must compile it into an object module. An *object module* is a fixed file (record length of 1024 bytes) created by the compiler. It contains the object (machine) code generated by the compiler. It is relocatable, that is, all addressing which is not position-independent is marked so that it can be resolved by the binder to absolute addresses. The compiler names the object module with the same name as the source module except the suffix is changed to `.obj`.

The VOS compilers share a common code generator. An object module generated by one compiler can call an object module produced by another. The calling sequence, at the machine level, is the same for all languages. For more information on the calling sequence, see the section “The Calling Sequence” later in this chapter.

Each object module has four sections: code, symbol table (symtab), static, and external static.

- The *code* section contains the machine code.
- The *symbol table* contains information relating symbols (such as variable names) to the addresses used in the program. It is primarily used in debugging the program. This section exists only if you specify `-table` or `-symbol_table` when you compile the program.
- The *static* section contains internal static data used by the program.
- The *external static* section contains global static data used by the program.

Binding a Set of Object Modules

The *binder* brings together a set of separately compiled object modules into an executable form called a *program module*. Like the object module, the program module is a fixed file, in this case with record length of 4096 bytes.

This process of combining object modules into a program module is called *binding* and is performed with the `bind` command. In this command, you must list the name of the primary object to be bound, as well as names of those object modules which are not in your object library (see Chapter 3). You can explicitly name the program module. Otherwise, the binder gives it the name of the first object module you list, replacing the `.obj` suffix by `.pm`. You can also specify detailed instructions concerning the bind (such as what object modules to bind, what libraries to search, and so forth) in a binder control file. Note that on some other systems binding is referred to as *linking*. On VOS, the term link has a different meaning. See Chapter 5, "The File System," for a description.

The binder resolves references to external procedures in the program module. It copies the language runtime routines from a system library. It copies your separately-compiled object modules into the program module. The locations of these modules, relative to the beginning of the program module, are available to the binder so the references can be resolved. The binder sets aside space for external static variables. Only references to the system subroutines and certain data kept by the operating system are not resolved in the program module. These references are resolved when the program is executed. Thus, a program module is not dependent on the version of the operating system under which it was bound.

Each program module has up to three types of sections plus a header and maps. User programs have only one section: the paged section. Operating system program modules may have two additional sections: the wired section and the initialization section. Each section in turn has four regions: code, symtab, static (also called unshared static), and external static (also called shared static), corresponding to the regions of the object modules being bound. When the program is loaded into memory, the code and symtab regions or memory can only be executed or read, not written to. The static and external static regions can be written to during execution. This determines which regions can be shared by multiple processes executing the same program (see "Sharing of Program Regions" in Chapter 2). Furthermore, the code and symtab regions of memory need not be written to the disk's paging partition if page control takes the pages containing these regions. Because they cannot be modified, they are still present in unmodified form in the program module file.

Debugging a Program

After you compile and bind a program, you can debug it using the debugger. There is also a multi-process debugger which can debug multiple programs or tasks or other processes. Both of these debuggers are described later in this chapter in the section "The VOS Debuggers".

VOS Programming Language Support

Six programming languages are available for programming under VOS.

- BASIC
- C
- COBOL
- FORTRAN
- Pascal
- PL/I

Programs can also be written in assembler.

These programming languages are described in the following language reference manuals.

- *VOS BASIC Language Manual (R011)*
- *VOS C Language Manual (R040)*
- *VOS COBOL Language Manual (R010)*
- *VOS FORTRAN Language Manual (R013)*
- *VOS Pascal Language Manual (R014)*
- *VOS PL/I Language Manual (R009)*

The assembler is described in the *VOS Hardware Reference Manual (R008)*.

Stack Frames And The Calling Sequence

When a program is executed, the operating system creates a stack frame for it. When any procedure is called, a stack frame is also created for it.

The stack frame contains the return address of the procedure as well as storage for automatic variables and parameters passed to the procedure. The layout of the stack frame is shown in Figure 6-1.

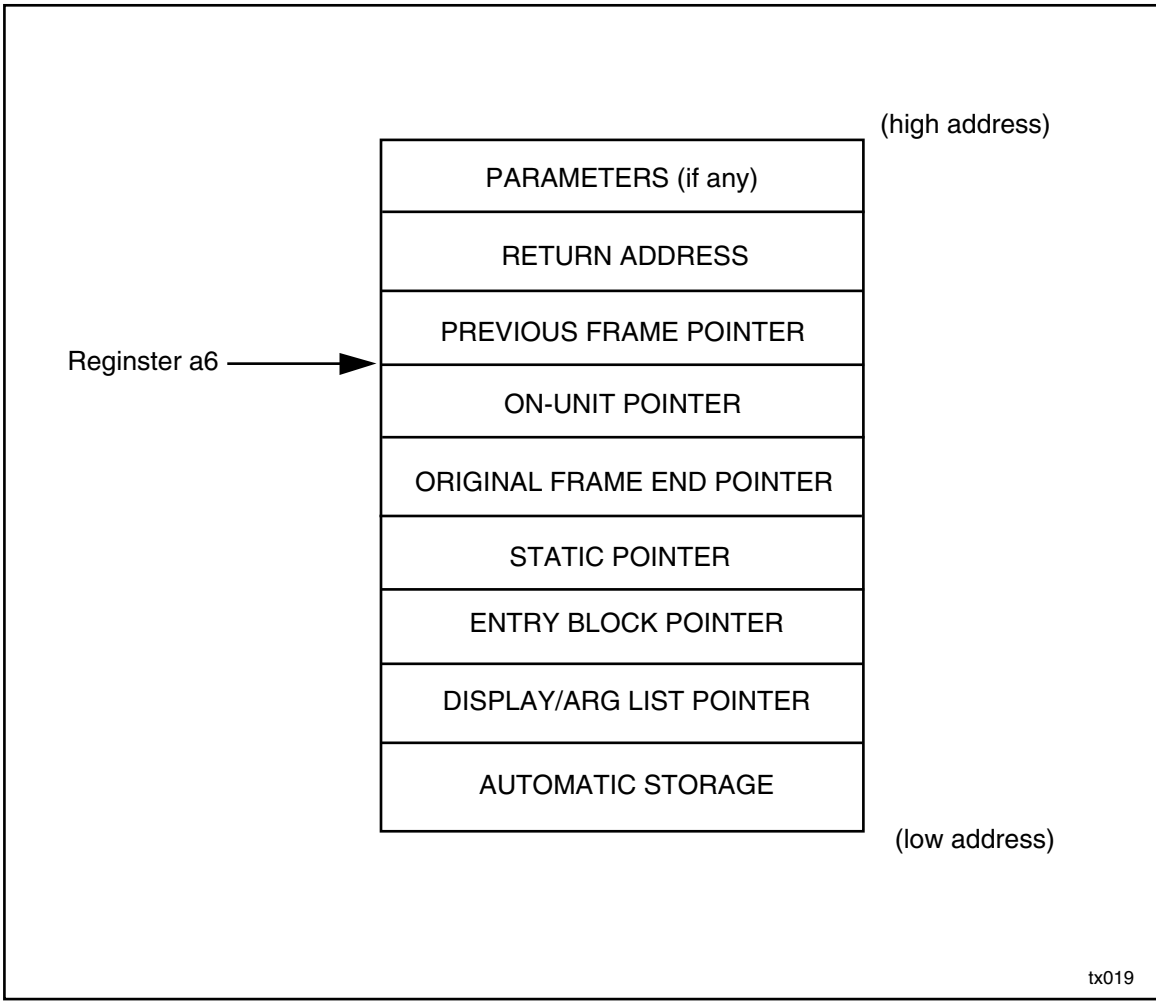

Figure 6-1. The Stack Frame

When a process calls a function or subroutine, the calling procedure performs the following steps:

- pushes pointers to all arguments on the stack (except when the program is in C, in which case the arguments themselves are pushed)
- gets the code and static pointers for the procedure to be called
- jumps to the first executable instruction of the called procedure.

The called procedure then performs the following steps:

- saves the current beginning-of-frame pointer
- pushes a stack frame of the required size
- loads a pointer to the entry block

- saves the original frame-end, static, and entry-block pointers on the stack
- executes the procedure
- pops the stack by putting the beginning of frame pointer into the stack frame pointer
- pops previous-frame pointer from the stack and puts in the beginning of frame pointer
- returns.

After the called procedure has returned, the calling procedure performs the following steps:

- reloads its static pointer from its stack frame
- pops arguments off the stack.

The *VOS Hardware Reference Manual (R008)* shows samples of the assembler code that performs the above steps.

You need to be aware of the stack frame format and the calling sequence only if you are programming in assembler. For the higher-level languages, the operating system handles parameters, stack frames, registers, and so forth for you. In addition, the assembler has pseudo-ops that often let you ignore the details of the stack frame layout.

Memory Allocation

A process can use both static and dynamic storage. *Static storage* is storage allocated by a compiler. The amount of static storage allocated to a program is known before the program runs. A program can use static storage by declaring a variable to be a static variable. The implementation varies from language to language; consult the reference manual for the appropriate language.

Storage can also be allocated during program execution. Two types of storage are allocated when a program executes: automatic storage and dynamic storage.

Automatic storage is allocated on the stack. Variables stored on the stack are called *automatic* variables. They are allocated automatically whenever the procedure that declares them is activated. These variables are part of the procedure's stack frame. Stack frames are described in the section "Stack Frames and the Calling Sequence." By default (for languages other than FORTRAN and COBOL), variables declared inside a procedure are automatic variables. For example, in the following code sample, `counter` and `buffer` are automatic variables:

```
function_name()
{
int  counter;
char buffer[80];

     .
     .
     .
}
```

Dynamic storage is allocated in the user heap. Space is allocated in the heap when a program makes an explicit call to a memory allocation routine, such as the system subroutine `s$allocate`, the PL/I `allocate` built-in function, or the C `malloc` function. The program specifies the amount of memory needed. The program is also responsible for freeing this storage (using, for example, `s$free` or `free`). In the following example, `buffer` is a dynamic variable:

```
function_name()
{
char      *buffer;

buffer = (char *) malloc(80);

}
```

Error and Condition Handling

The operating system supports error and condition handling by programs. This is described in the next two sections.

Error Handling

The VOS subroutines report errors by returning error codes. Each subroutine that can detect errors has an `error_code` argument. If no errors occur when the subroutine is executed, a value of 0 is returned in this argument. A non-zero value indicates an error. The calling program should examine the returned value to determine whether an error occurred and if so, the nature of the error.

All VOS error codes are two-byte integers. A program can access these variables by declaring the appropriate static external variables. For example, a C program could declare the variable `e$end_of_file` as an external global variable, as follows:

```
extern short     e$end_of_file;

main()
{
     /* body of the program  */
}
```

In PL/I, the declaration would be as follows:

```
declare e$end_of_file fixed binary(15) static external;
```

When the binder encounters an external static variable whose name begins with `e$`, it checks to see if it is the name of an operating system error code. If so, it initializes this variable with the appropriate value (in this case 1025). The program could then use the following syntax in checking the error:

```
if (error_code == e$end_of_file)
```

This method saves you from needing to determine the actual numbers associated with an error code.

The subroutine `s$error` can be used to report an error to the user. If the program passes it an error code, it displays the associated error message to the process's default output port. For more information, see the VOS Subroutines Manuals.

All messages are contained in message files, not in the program itself. Messages can thus be changed without the need of recompiling and rebinding the program that references the file. For example, when the operating system is used in a non-English speaking country, the message file is replaced with one containing messages in the appropriate language.

Condition Handling

The operating system also has a more general condition signalling facility. You can use system subroutines to manage condition handling in a program. This facility provides the same capabilities as the condition handling mechanism in PL/I (which uses the `on` statement).

The subroutine `s$enable_condition` enables a condition handler in the current stack frame of a running program. A condition handler is associated with the stack frame in which it is enabled. When a stack frame is discarded (for example, when a procedure returns), the condition is *reverted*; that is, the handler is no longer invoked. The subroutine `s$revert_condition` also reverts a condition.

When the operating system detects a condition, it activates a condition handler for that condition if you have enabled one. A program can also signal a condition using the subroutine `s$signal_condition`. A program must signal any user-defined conditions since the operating system does not signal these.

When a condition occurs, the operating system searches your process's stack, starting at the current frame and working backward for the most recent frame in which the condition handler is enabled. If it finds a handler, it calls it. If not, it calls the default handler for the condition.

A condition handler can use the subroutine `s$find_condition_info` to obtain information about the signalled condition. It can use the subroutine `s$continue_to_signal` to resignal the condition in the preceding stack frame.

The operating system's default condition handler is called whenever a condition is signaled and you have not enabled a specific handler for that condition. For most conditions, the default handler writes an error message on the process's terminal output port, and suspends the program.

The following table lists the conditions recognized by the operating system. The first column gives the value for the `condition_number` argument. The third column describes the circumstances under which the operating system signals the condition.

(Page 1 of 2)

Condition Number	**Condition Name**	**Explanation**
1	`error`	The operating system signals the `error` condition when an uncategorized fatal error or exception occurs.
2	`undefinedfile`	This condition is signaled only in VOS PL/I. See the *VOS PL/I Language Manual (R009)*.
3	`break`	The operating system signals the `break` condition when the process receives a break request, for example, when a CTRL BREAK request is issued.
4	`endpage`	This condition is signaled only in VOS PL/I. See the *VOS PL/I Language Manual (R009)*.
5	`endfile`	This condition is signaled only in VOS PL/I. See the *VOS PL/I Language Manual (R009)*.
6	`key`	This condition is signaled only in VOS PL/I. See the *VOS PL/I Language Manual (R009)*.
7	`pause`	The operating system signals the `pause` condition when a VOS COBOL `stop` statement is executed.
8	`underflow`	The operating system signals the `underflow` condition when a language support routine detects a floating-point underflow.
9	`overflow`	The operating system signals the `overflow` condition when a language support routine detects a floating-point overflow.
10	`zerodivide`	The operating system signals the `zerodivide` condition when a language support routine detects a division by 0.
11	`fixedoverflow`	The operating system signals the `fixedoverflow` condition when a language support routine detects a fixed-point overflow.
12	*`user_defined`*	The operating system never signals user-defined conditions.
13	`warning`	The operating system signals the `warning` condition when an uncategorized nonfatal error or exception occurs.
14	`anyother`	The operating system never signals the `anyother` condition.
16	`alarmtimer`	The operating system signals the `alarmtimer` condition when the period of time specified in `s$set_alarm_timer` has elapsed.

(Page 2 of 2)

Condition Number	Condition Name	Explanation
18	`reenter`	The operating system signals the `reenter` condition when the `re-enter` command is issued at break level.
24	`cleanup`	This condition is signaled only in VOS PL/I, when the containing stack frame is popped as the result of a nonlocal goto.

The VOS Debuggers

The debugger is a symbolic debugger that allows you to do any of the following:

- start running a program
- set breakpoints in the program
- see the values of program variables and the contents of stack frames
- set the values of variables
- sstep through the program
- call procedures, subroutines, or VOS C functions with arguments.

The debugger can be entered by typing `debug` from command level or break level. From command level, you must specify the program to be debugged.

You can debug a program in either source mode or machine mode.

The source modes of the debugger correspond to the high-level languages VOS COBOL, VOS BASIC, VOS FORTRAN, VOS Pascal, VOS C, and VOS PL/I. You can debug a program written in one of these languages in the corresponding debugger mode.

The mode determines how the debugger interprets your requests and how it displays information about the program you are debugging. For example, when the debugger displays the data type of a variable, the form is in the language associated with the mode.

To debug a program in a source mode, you must have compiled the program with a symbol table. The symbol table contains information about the names of variables and their locations in the compiled program. Use the compiler's `-table` or `-production_table` option to incorporate a symbol table into the program module.

You can debug any program when the debugger is in machine mode. Machine mode is also called the object mode of the debugger. In this mode, refer to code and data values by address instead of by name, and you can examine the contents of processor registers.

For more information on the debugger, see *VOS Commands Reference Manual (R098)*.

Multi-Process Debugging

The debugger only allows you to debug the program your process is executing. To debug other programs or more than one program at a time, use the multi-process debugger. It is particularly useful when you want to debug a process that does not usually have a terminal associated with it, such as a server process, or when you want to debug a screen-oriented

program. In the latter case, you can debug the program from a separate terminal, and thus not interfere with the screens you are debugging.

Invoke the multi-process debugger with the command `mp_debug`. The set of processes that are debugged in a multi-process debugging session is called the debug process set; it can include processes from anywhere in your network.

When you invoke the `mp_debug` command, you intercept the debugging input and output requests for other processes. Initially, there are no processes in the debug process set. Processes can be added to or removed from the debug process set with `mp_debug` requests. The processes that you include are called slave processes because they are under the control of the multi-process debugger. Using the `mp_debug` requests, you can list these processes, suspend them, and restart them. You can also create a new process, call the debugger for it, and then start the process, all from within the multi-process debugger.

For more information on the multi-process debugger, see *VOS Commands Reference Manual (R098)*.

Obtaining Profiles of Programs

The `profile` command provides information about the performance and statement execution of a program.

To obtain a profile of a program, perform the following steps:

- Compile the program with either the `-profile` or `-cpu_profile` arguments.
- Bind the program.
- Execute the program.

When the program executes, a file with the suffix `.profile` is produced. This file is used by the `profile` command to generate a file with the `.plist` suffix. The `.plist` file contains performance information about the most recent execution of the program.

When you compile a source module with the `-profile` option, the compiler inserts code to count the number of times each source statement is executed when the program runs. When you compile a source module with the `-cpu_profile` option, the compiler inserts code to obtain the amount of CPU time spent executing each statement, the number of page faults taken for each statement, and the number of times each statement executes. The compiler also marks the object module so that the operating system produces a profile file every time it runs a program containing the object module.

A profile file contains performance information about all the object modules compiled with the `-profile` or `-cpu_profile` options and bound together in the program. The name of the profile file is the name of the program module with the `.pm` suffix replaced by the suffix `.profile`. When the program is run, the operating system puts the profile file in your current directory, overwriting any existing profile file of an earlier execution. When you issue a `profile` command, the program module must be in the same location in the directory hierarchy as when executed and it must be the same version (not rebound).

The `.plist` file contains a section for each object module compiled with the `-profile` or `-cpu_profile` option, and bound in the program. Within each section, there is one line for

each executable statement in the source module giving performance information about the statement (although data for a call statement includes the subroutine only if the subroutine is not itself profiled). If the source module was compiled with the `-profile` option, the information is the source module statement number and the number of times the statement was executed during the program's execution (frequency). If the source module was compiled with the `-cpu_profile` option, the information is the same module statement number, its frequency, the CPU time (in milliseconds) spent executing the statement, and the number of page faults taken while the statement was being executed.

To combine profile information from multiple executions of the same program in one profile, use the `add_profile` command to create a sum file. Note that each execution of the program overwrites the profile file created by the previous execution, so you must run `add_profile` between executions of the program. When you are done, run `profile` on the sum file.

For more information on the the `profile` and `add_profile` commands, see *VOS Commands Reference Manual (R098)*.

National Language Support

The COBOL, C, Pascal and PL/I compilers support international character sets. This chapter describes how these compilers provide international character set support. See Chapter 2 for more information on how the operating system supports international character sets. NLSNational language support

Canonical Form and Normal Form

As described in Chapter 2, all strings can be written in canonical form, in which every right graphic character is preceded by a single-shift character. If a string is in canonical form, many of the language string handling functions work correctly. For example:

```
i = index(str1, str2);
```

This sets `i` to the byte position in `str1` of the first occurrence of `str2`, as long as both strings are in canonical form.

Each compiler also supports the `shift` function, which converts any string to a canonical string. For example:

```
canon_str = shift(str);
```

This statement converts the contents of `str` to its canonical form and assigns that value to `canon_str`.

The compilers also provide a built-in function, `unshift`, which can be used to convert any valid NLS string to a common string. (See Chapter 2 for a description of common strings.)

For example:

```
common_str = unshift(str);
```

This statement converts the contents of `str` to common form (a fully single-shifted string excluding Latin alphabet No. 1 single shifts) and assigns it to `common_str`. If a string

containing locking-shifts is to be written to a terminal or text file, then this built-in function can be used to convert that string as required.

NLS System Subroutines

Two system subroutines, which can be called from any programming language, support manipulation of NLS strings by programs. The `s$convert_charset` subroutine converts a source string with a given default character set to a target string with a potentially different default character set. The `s$validate_string` subroutine checks a source string for consistency as an NLS character sequence. For an explanation of these and other system subroutines that provide international character set support, see the VOS Subroutines Manuals.

The Forms Management System

The operating system Forms Management System (FMS) is a set of tools, system software, and programming language extensions that give your programs a consistent interface for the entry and display of data on a video display terminal. The interface is designed to be efficient for applications in which the user performs transactions that require the entry of several specific pieces of information.

FMS has three major components.

- The Forms Editor allows you to create and modify forms.
- The `screen` statements are extensions to the VOS programming languages that allow you to invoke a form within a program.
- The Forms Processor is the system software that is invoked by the `screen` statements. The Forms Processor manages form displays.

In a typical forms application, the three components work together as follows:

1. Using the Forms Editor, you design a form. The Forms Editor stores the form description as a form object module that can be referenced by a program.
2. Within a program, you reference the form in `screen` statements.
3. When you compile your program, the compiler translates the `screen` statements to object code that calls the Forms Processor run-time software.
4. When you bind your program, the binder links your program object module with the form object module and the Forms Processor software.
5. When control reaches a `screen` statement during program execution, the Forms Processor is invoked to manage the form. For example, when an `input screen` statement is executed, the Forms Processor displays the form. The Forms Processor then allows the user to move the cursor and type values within the form. When the user submits the form, the Forms Processor validates the input values and loads them into program variables specified within the `screen` statement. Control then returns to the program and execution continues with the statement following the `screen` statement.

For more information on the Forms Management System, see the VOS Forms Management System Manuals.

Chapter 7:
Networking Support

The operating system provides support for local networks and for wide area networks.

Local networks connect several modules into a single system. This support is provided by the StrataLINK local network. Local networks, connecting both Stratus and non-Stratus modules, can also be configured using other local area network products such as Ethernet. StrataLINK is described in more detail in the following sections. Ethernet is described in the *VOS Ethernet Protocol Support (R128)* and in the TCP/IP manuals.

Multiple systems are tied together primarily by the StrataNET wide-area network. X.25 and X.29 support is also provided.

This chapter provides an overview of the StrataLINK local network and the StrataNET wide area network. For more information, see the manuals on these products.

Possible Configurations

Modules can be tied together in several ways, depending on their locations and whether you want a single system or multiple systems. Four different local and wide-area network configurations can be created using StrataLINK and StrataNET hardware and software. These configurations are:

- single system
- multisystem
- single cluster
- multicluster.

A *single-system* network is created when two or more modules are connected using StrataLINK and defined as a single entity. Modules connected into a single system function as if they were a single machine, from the user's point of view. A maximum of 32 modules can be connected to form a single system. This configuration is described in greater detail in the manual *VOS StrataLINK Administrator's Guide (R192)*.

A *multisystem* network is created when individual Stratus systems that are geographically remote are connected using StrataNET. These systems can be composed of either a single module or multiple modules. This configuration is described in greater detail in the *VOS Communications Software: X.25 and StrataNET Administration (R091)*. Figure 7-1 illustrates a multisystem network.

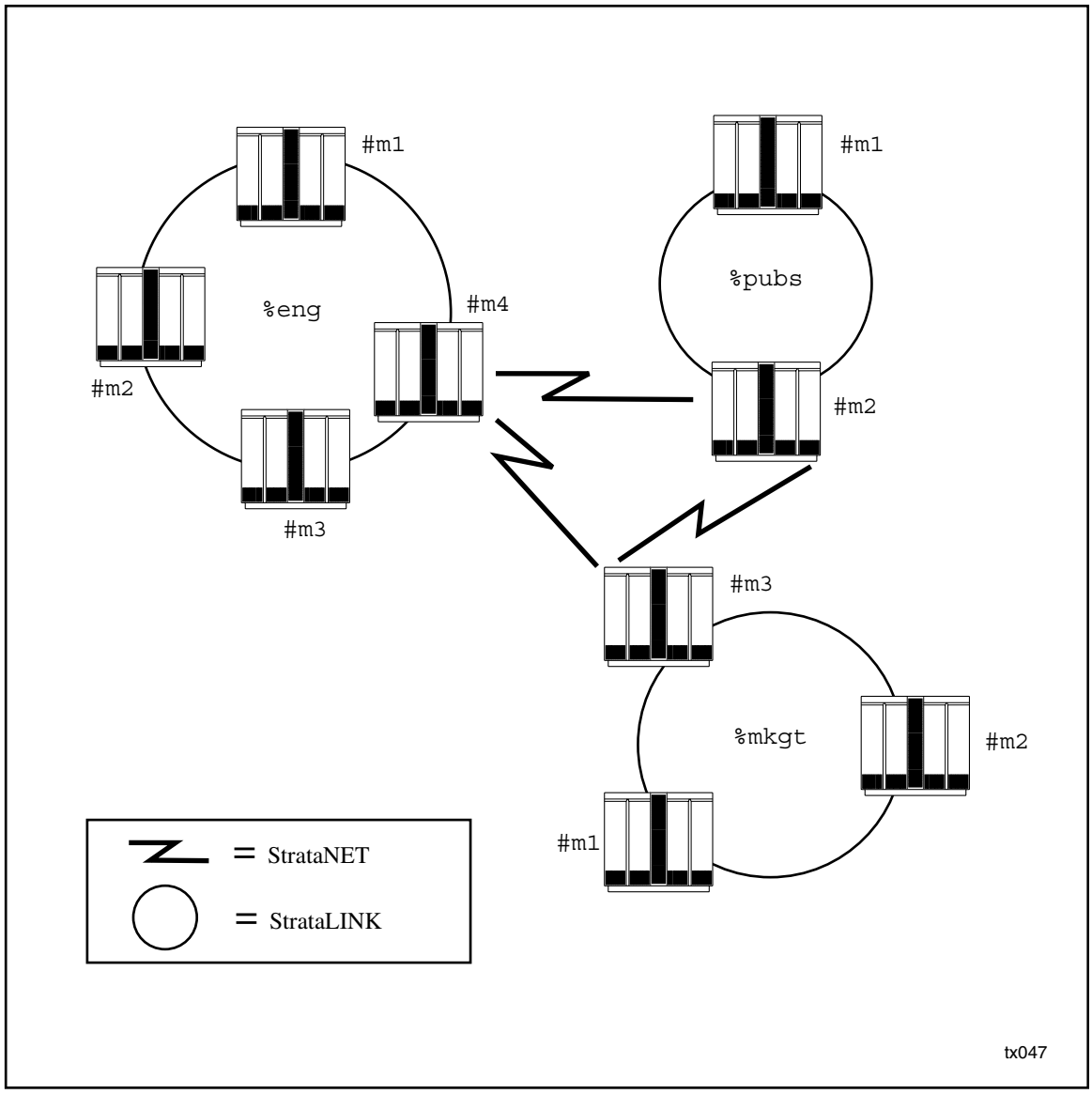

Figure 7-1. Multisystem Network

A *single-cluster* network is created when two or more Stratus systems located at the same site (the same building or neighboring buildings) are connected using a backbone ring. A *backbone ring* is simply a StrataLINK ring that provides communication between separate systems. Each system (either a multiple- or a single-module system) is connected to the backbone ring by means of a bridge module (see Figure 7-2). The maximum number of systems that can be connected is 32. Since each system can consist of 32 modules, a single-cluster network can connect a maximum of 10 modules. This configuration is described in greater detail in the *VOS StrataLINK Administrator's Guide (R192)*.

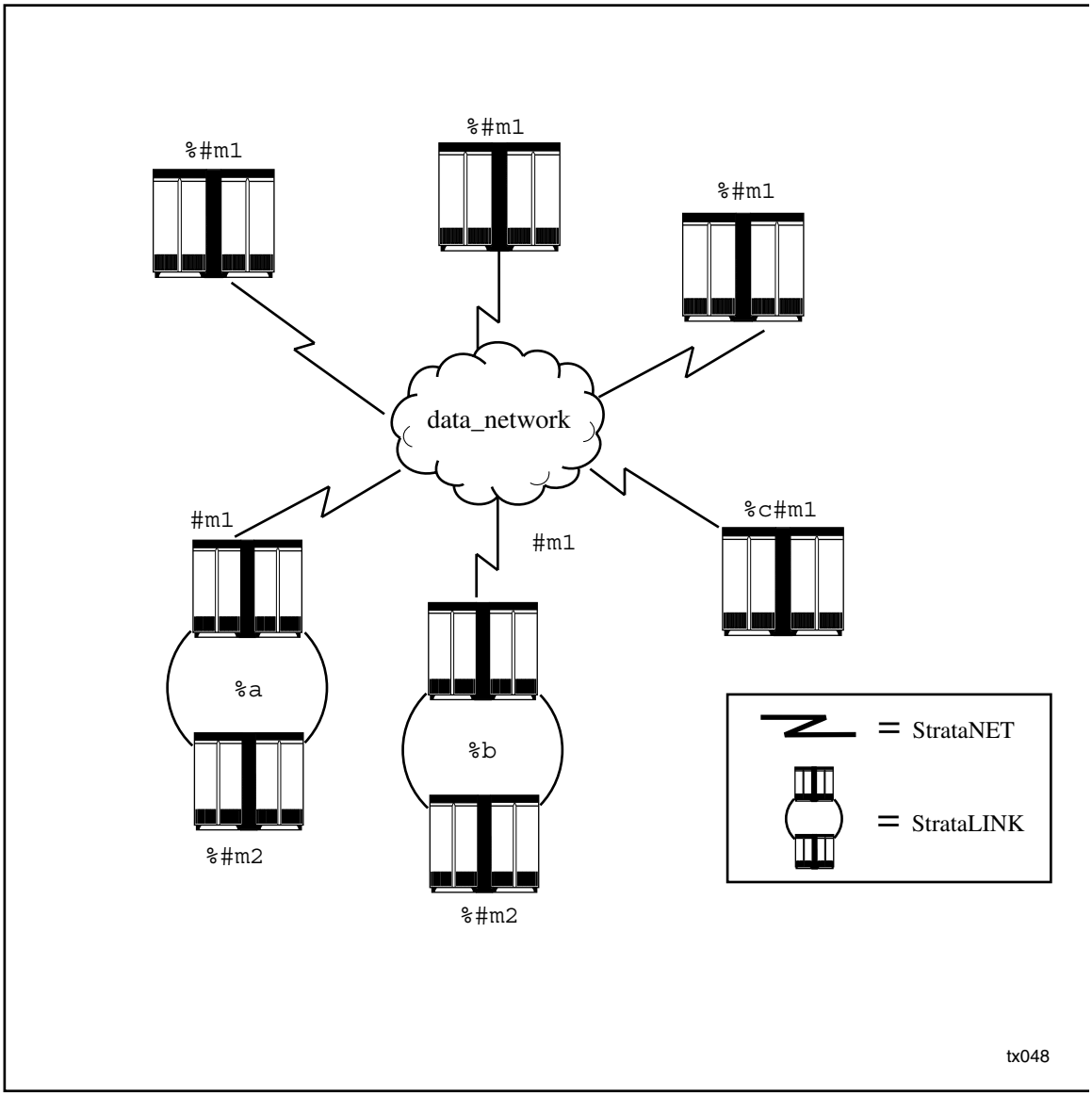

Figure 7-2. Single-Cluster Network

A *multicluster* network is created when a single cluster is connected to a remote system(s) and/or cluster(s) using StrataNET. This configuration is also described in the *VOS StrataLINK Administrator's Guide (R192)* manual.

The StrataLINK Local Network

The StrataLINK local network provides a 12.5 MHz (1.4 MByte/sec) data path between modules, tying multiple modules into a single system. It uses a collision-detection scheme, and consists of coaxial cable lines configured as a ring, but it is routed through a hub and thus

appears physically as a star layout. Up to 32 modules can be tied together on a single StrataLINK. More modules can be added to a system by configuring one or more subrings.

The StrataLINK local network allows for horizontal growth: when the capacity of a single processing module is exceeded, additional modules can be added nondisruptively, without any impact to end users. The interconnected modules offer a single system image to the user, who needs to do nothing different whether using a single or multimodule system.

The StrataLINK Software

The StrataLINK software enables users and application programs to transparently access other modules in a system. Users simply specify resources on another module the same way they would on their local module. The StrataLINK software handles all the network operations.

Communication between modules occurs using messages that are transmitted as one or more packets. Packets are made up of a header containing destination and source address fields, up to 4096 bytes of data, and a reply field to acknowledge receipt of the message. They flow along the link in one direction and circulate until they return to the sender, which checks for an acknowledgement from the receiver and then purges the message. Broadcast messages are supported. The physical message passing from module to module, including automatic retry on error, takes place at the hardware level, with no CPU overhead. The delay at each module is only two bits, or 160 nanoseconds. The links are duplexed for fault tolerance and operate independently. Traffic is spread between both links, which provide a peer-to-peer, any-to-any communications path between the operating systems of the interconnected processing modules.

Link Server Process

Stratus network communication always involves two processes: a client and a server. A *client* is a process (generally a user process) that requests service from a module other than the one on which it is executing. A *server* is a system process whose purpose is to receive and respond to requests from clients.

Servers that operate within local networks are *link servers*; they exchange messages with clients via the StrataLINK network. Each module provides one or more link servers to serve clients in other modules of the same system (or other systems, if you are using StrataLINK to connect multiple systems).

One or more link servers on each module in the system listen for incoming requests. In most cases, several link server processes are started when the module is booted.

Each link server listens to a socket for incoming requests. A *socket* is an address on a module that is used for network communication.

How StrataLINK Software Handles Requests

The operating system determines if a request for service from a process is local. If it is not, the operating system builds a message and forwards it to another module, where the link server processes it. Figure 7-3 and 7-4 provide an overview of how this process works. The following paragraphs provide more detail.

As shown in Figure 7-3, a user or application requests service using an operating system subroutine by either calling the subroutine directly or issuing a command that calls the subroutine. This subroutine call in turn is passed to the kernel. The operating system on the local module checks to see if that subroutine is requesting local resources. If so, it performs the requested action. Otherwise, the subroutine call is converted to a network call (the `s$` call is translated into an `n$` call). This network call builds a message consisting of one or more packets for the local net driver. The net driver takes each network packet and sends it over the link to the remote module's socket input queue. The `n$` call, and hence the application, waits for the server to return a reply.

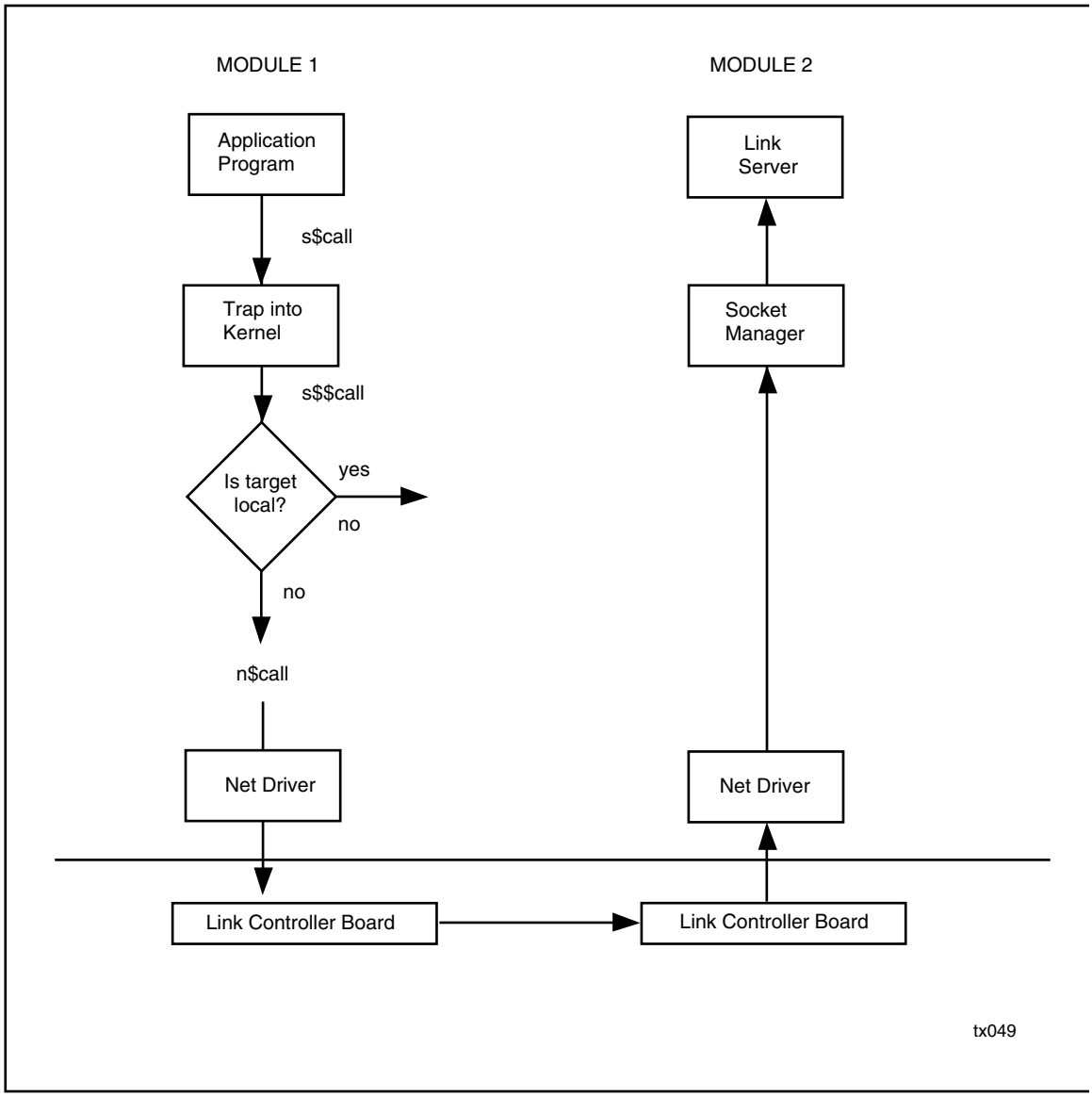

Figure 7-3. StrataLINK Software

On the remote module, the socket manager listens for incoming requests. When one arrives, it passes the request to the link server, which adopts the identity of the client and issues a call to perform the desired service. It then builds and returns a reply to the local module (as shown in Figure 7-4).

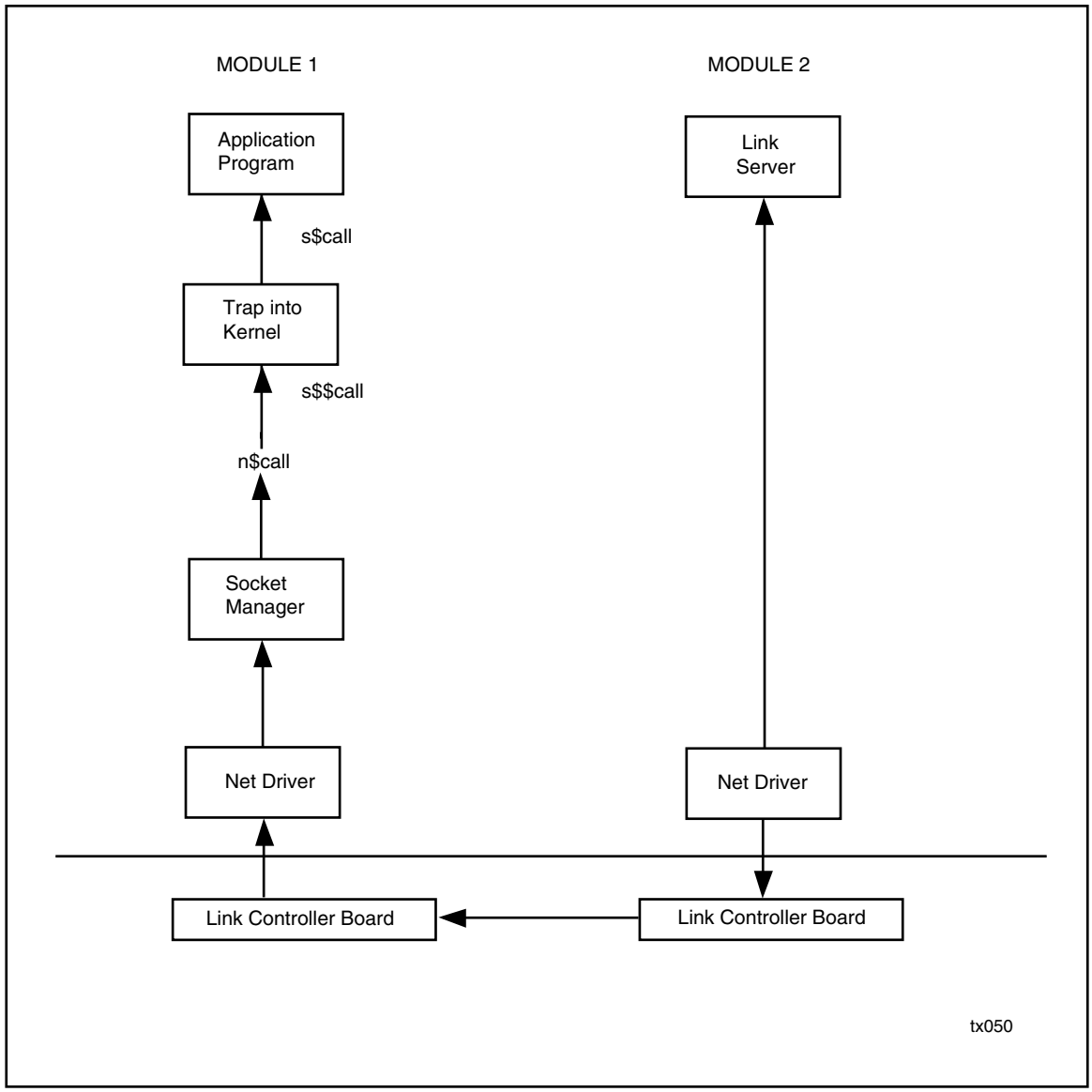

Figure 7-4. StrataLINK Software Message Return Path

The Network Watchdog Process

The *network watchdog* is a system process that periodically performs the following functions:

- checks to see if other modules in the network have either lost or regained communication with the module the network watchdog is running on
- cleans up file, device, and event attachments that have become inoperative because of communications failures or normal module shutdowns.

A network watchdog process runs on each module in the system. It is started when the module is booted.

Cluster Networks

As described previously, multiple systems in close physical proximity to one another can be connected using the StrataLINK local network. This configuration is described in the section "Possible Configurations," earlier in this chapter.

The details of how the StrataLINK local network works are the same for cluster configurations as for the single system configuration described in the previous sections. Additional link servers are used on the bridge modules to pass and receive information between systems in the cluster. Generally, a different link server handles each of the following major paths of communication within a cluster:

- communication between the modules in the local system
- communication between local-system modules and a remote system through the local bridge module and out over the backbone ring (this link server is not necessary if the local system is a single module)
- communication between a remote system and the local system received and forwarded by the local bridge module
- communication between the systems in the cluster and remote systems accessed via a wide area network connection (this link server is only necessary if the cluster has a wide area network connection).

The StrataNET Wide Area Network

StrataNET is the Stratus wide area network facility, used to connect systems. It uses the the Virtual Circuit Facility and X.25 software (see *VOS Communications Software: X.25 and X.29 Programming (R028)*) to enable transparent system calls among multiple Stratus systems. Using the StrataNET wide area network, the operating system can transparently access remote resources (files, programs, devices). Thus, a user can simply request a resource by name (such as the path name of a file or program), and the operating system uses StrataNET to respond to those requests involving resources located on other systems. Unlike the StrataLINK software, StrataNET software is not entirely contained by the operating system. It uses network clients and servers and X.25 gateways to handle communications.

Multiple systems are tied together by *gateway* modules. Messages to other systems are sent by the StrataLINK local network to the gateway module, and then by the StrataNET wide-area network to the remote system.

When an application makes an operating system subroutine (`s$`) call, the operating system checks to see if the resources requested are on the local module. If not, it checks to see if they are on the local system, in which case it operates as described for the StrataLINK local network. If the resource needed (a file, for example) is on another system, it uses the StrataNET wide-area network to access the file.

On each module with a gateway to another system, two system processes handle StrataNET communications:

- The *network client* formats packets and sends requests over the network.
- The *network server* listens for and processes incoming requests.

For example, suppose an application on one system, `%sys1#m1`, references a file on `%sys2#m1` (see Figure 7-5). The `s$` call that makes this reference is translated into an `n$` call and is sent to `%sys1#m2`, since the gateway connecting the systems is on `#m2`. On `%sys1#m2`, the network client receives the request, formats it, and sends the packets to the listening network server on `%sys2#m2`. The link server sends the request to `#m1`, and the acknowledgment and reply follow the reverse route back.

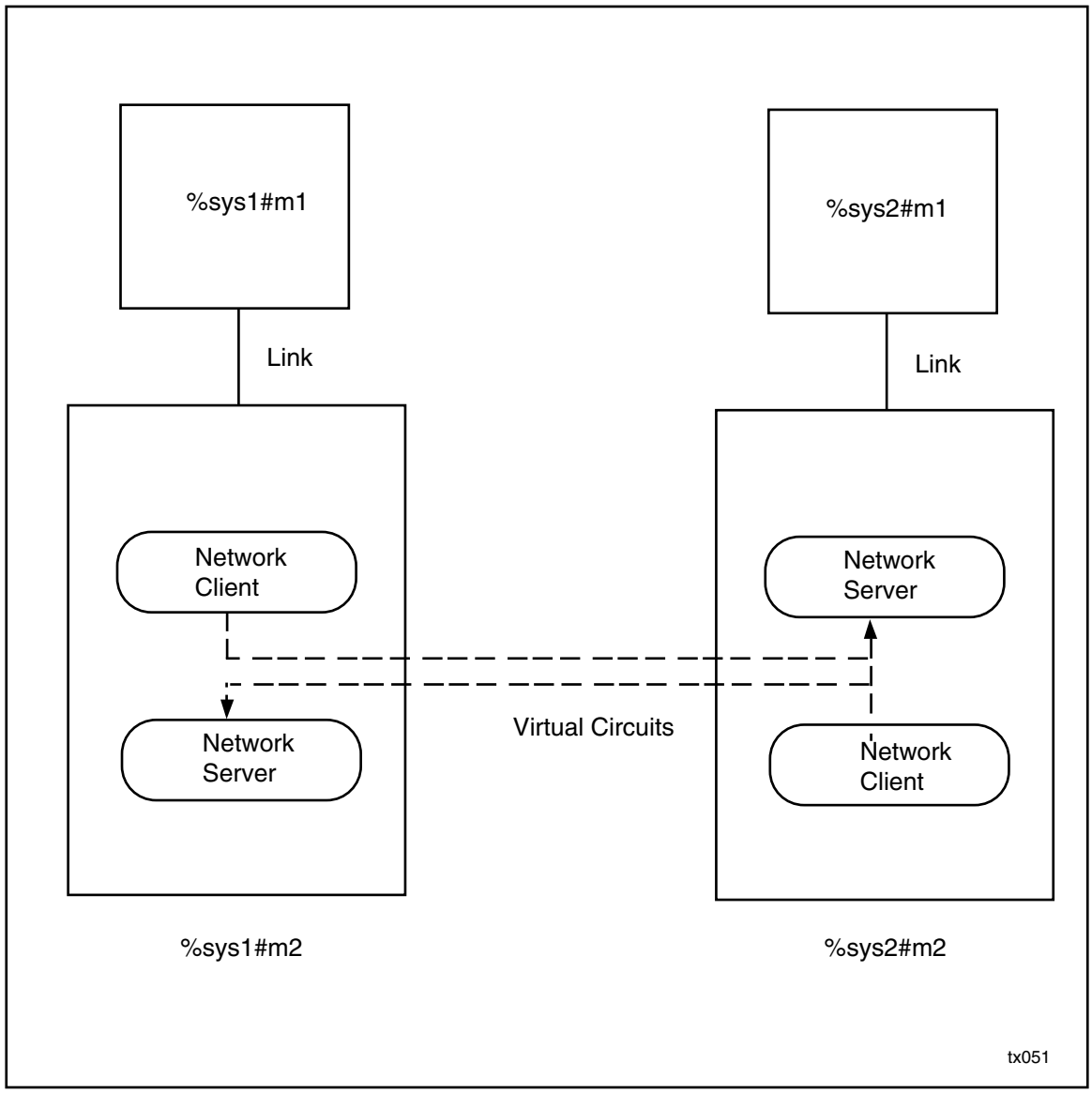

Figure 7-5. The StrataNET Wide Area Network

Chapter 8:
Tape Processing

The operating system supports the use of tapes at command level and from programs. Both of these methods are described in detail in the *VOS Tape Processing User's Guide and Programmer's Reference (R052)*. This chapter provides a brief overview of the operating system tape processing facility.

Using Tapes from Command Level

There are two ways to use tapes at command level:

- reading or writing ANSI, IBM, or unlabeled tapes using the `read_tape` and `write_tape` commands
- reading or writing to operating system save tapes using the `save` and `restore` commands.

Tapes and Tape Devices

The operating system supports two types of tapes:

- Reel-to-reel tapes
- Cartridge tapes

Reel-to-reel tapes usually come in lengths of 1200 or 2400 feet. Each reel comes with a plastic ring, the *write ring*, that can be inserted in the slot around the hub. The write ring must be in the slot before you can write to the tape.

Cartridge tapes are somewhat smaller than standard VHS video tapes.

Information on reel-to-reel tapes is stored magnetically in parallel tracks, with the one bit of each byte stored parallel to the other bits. There are usually nine such tracks. The *density* of a tape is measured in bits per inch (bpi); that is, the number of bits of information per inch in a single linear track.

On cartridge tapes, information is stored serially; the bits in a byte are stored one after the other. There are parallel tracks, but the bit in one track is unrelated to the bit in the tracks parallel to it.

The amount of information that can be stored on a reel-to-reel tape depends upon the block size, density, and the tape blocking factor. A formula for calculating the number of tape blocks on a tape is given in *VOS Tape Processing User's Guide and Programmer's*

Reference (R052). For cartridge tapes, the capacity depends upon the block size and blocking factor, since each drive only writes at one density.

Tapes are processed in logical groupings called *tape volumes*. One volume can contain a single file, multiple files, or a portion of a file (which would be continued on another volume). In most cases, one volume is contained on one reel, and the terms can be used interchangeably.

A *block* is a physical grouping of characters that is written or read as a unit. A *record* is a logical grouping of related information. A record can, for example, correspond to a line of data in a file. Records can be of varying length, fixed length, or undefined length. The number of records per block is called the *blocking factor.*

The *file format* specifies the type of records (varying, fixed, or undefined length) in a file. It also specifies whether blocks can contain more than one record (in which case the file is said to have a *blocked record format*) and whether records can begin in one block and end in another (a *spanned record format*).

Tape labels are special records appearing at the beginning of a volume of tape and at the beginning and end of files on the volume. The labels provide details about the formats and other attributes of files on the tape.

A record of zero length is called a *tape mark*. A tape mark acts as a delimiter between labels and files. The end of the volume is indicated by an *end-of-volume mark*. This mark follows the last data on the tape. Note that the end-of-volume mark is not necessarily at the end of the tape. However, the operating system ignores data after the end-of-volume mark.

The Steps in Reading and Writing a Tape

To read or write a tape, follow these steps.

1. Attach a port to the tape device using the `attach_port` command.
2. Physically load the tape into the tape drive.
3. Mount the tape by issuing the `mount_tape` command. When you *mount* a tape you describe the tape and how you are going to use it. The operating system verifies that the tape matches any specified parameters and positions the tape to the beginning of the first file.
4. Read or write the tape. To *read* a tape means to copy a file or files from the tape to disk. To *write* a tape means to copy a file or files from disk to tape. To read or write the tape, use `read_tape` and `write_tape`, or `save` and `restore`. Use `read_tape` to read a tape written using `write_tape`. Use `restore` to read a tape written using `save`.
5. When you have finished reading and writing, dismount the tape (using `dismount_tape`) and detach the port (using `detach_port`).

If you are using a tape whose parameters differ from the operating system defaults, you may have to use `set_tape_defaults` before mounting the tape. If you want to read a specific file from the tape or write to a certain location on the tape, you must first position the tape by using the `position_tape` command. You can find out what files are already on the tape

using the `list_tape` command. For tapes written using `save`, you can determine the contents of the tape using the `restore` command with the `-no_restore`, `-list`, and `-same_place` arguments (see the *VOS System Administrator's Guide (R012)*).

As noted in Step 4, you can use either `read_tape` or `restore` (or `restore_object`) to read a file or files from a tape. Likewise, you can use either `write_tape` or `save` (or `save_object`) to write files to a tape. The `read_tape` and `write_tape` commands can be used any time you are transferring unindexed files between disk and tape. However, these commands transfer only files; they do not transfer a file's indexes or access control list. They also cannot copy other objects such as directories and links. To transfer any of these objects to or from tape, you must use `save` and `restore`.

Using Tapes from a Program

Reading files on tapes is similar to reading files sequentially on disk. However, there are some peculiarities:

- Tape files cannot be accessed in random or indexed fashions.
- The operating system treats tapes with label formats that it does not support as unlabeled. When reading such tapes, the program must understand the format of the tape, including the format of the labels (if any), files, blocks, and records. It then can break the data into records if multiple records have been packed into a block, and remove any characters that were added to the records to pad them to the minimum record length when the file was written.

To process a file from a tape, you must follow these steps:

- Attach a port to the tape drive using `s$attach_port`.
- Mount the tape and change any defaults that have to be changed using `s$control` with the appropriate opcodes.
- If need be, position the tape, again using `s$control`.
- Open the current file, using `s$open`.
- Read or write the tape, using `s$seq_read` or `s$seq_write`.
- Close the file, using `s$close`.
- If you have more files to process, repeat the previous three steps.
- Dismount the tape, using `s$control`.
- Detach the port, using `s$detach_port`.

For more information on using tapes from within an application program, refer to the Programmer's Reference section of the *VOS Tape Processing User's Guide and Programmer's Reference (R052)*.

Glossary

abbreviation

A character string in a command line that the operating system replaces before executing a command. For the operating system to perform this replacement, a user must specify a file named `abbreviations` (with the `use_abbreviations` command) containing unique abbreviations and their expansions. See also **all directive**, **first directive**, **replacement directive**, and **subsequent directive**.

abbreviation parameter

A variable in the output string of a first directive that is replaced by one or more input arguments or argument values on the command line.

abbreviations file

A text file containing abbreviation directives.

abbreviations table

A table created from an abbreviations file by the `use_abbreviations` command. The operating system refers to a user's abbreviations table to determine the replacement for an abbreviation.

abort

1. In TPF, in processing of two-way queues, to discontinue processing of a message without sending a reply.

2. In transaction protection, to ensure that any transaction files involved in a transaction will remain exactly as they were before the transaction was started. This is done by calling `s$abort_transaction`.

access

To read from or write to a file or device. See also **access mode**, **access right**, and **access type**.

access code

A code used in access control lists and default access control lists to show access rights. access control

The mechanism that the operating system uses to determines a user's access rights to files and directories.

access control list (ACL)

A list that the operating system uses to determine a user's access rights to a particular file or directory. An ACL is a list of entries, each of which shows an access code and a user name. The following is an example of an ACL for a directory.

```
m    Jones.sales
s    *.sales
n    *.*
```

The following is an example of an ACL for a file.

```
w    Jones.sales
r    *.sales
n    *.*
```

See also **default access control list (DACL)**.

access mode

The method that the I/O system uses to access records for reading or writing. The access modes are sequential, random, and indexed.

access right

A designation that determines the operations a user is permitted to perform on a file or directory. Access rights to a file are null, execute, read, and write. Access rights to a directory are null, status, and modify.

access type

A type of I/O operation performed on a file or device. Access types are input, output, append, update, dirty read, and dirty notify.

access violation

Any attempt by a user to gain unauthorized access to a file, a directory, or a system, or to issue a privileged command or subroutine without being privileged. Also, any failure by a user to supply the correct password.

ACL

See **access control list**.

active directory

The state of a directory when it is assigned an active directory table entry (ADTE). The operating system activates a directory when information from the directory is put in main storage the first time a process operates on the directory. This permits the operating system to efficiently handle subsequent operations on the directory.

When a directory has not been used for a period of time, the operating system automatically deactivates it, thus freeing up the space in main storage. You can also explicitly deactivate a directory with a program call to the VOS system subroutine `s$deactivate_dir`.

When a directory is active, all of its superior directories are active.

active file

The state of a file when it is assigned an active file table entry (AFTE). The AFTE contains the file map and other data to enable the file system to manage the file. If a file is active, its containing directory and all of its superior directories are active.

active queue

The current print queue and all print queues that have been started with the `start_queue` request to the `spooler_admin` command and that have the same characteristics as the current queue.

adapter

See **I/O adapter**.

address

The location of an area of storage. An address is a four-byte value.

address space

See **virtual address space**.

alias

An alternate (usually shorter) form of a person name that can be used with the `login` command.

aligned

Allocated on a particular storage boundary. In VOS PL/I, data can be word-aligned, byte-aligned, or bit-aligned. If you specify the `aligned` attribute for a character or bit string, the string is word-aligned. In VOS Pascal, data can be aligned on a multiple of a byte boundary, a byte boundary, or on a bit boundary.

all directive

A type of replacement directive in an abbreviations file that tells the operating system to replace an input string with an output string wherever the specified input string occurs in the command. See also **abbreviation**, **first directive**, **replacement directive**, and **subsequent directive**.

allocate

In those programming languages that support dynamic memory allocation, to set aside an area of storage for a particular purpose.

American National Standard Institute (ANSI)

A group that promotes standards for computer languages and devices. An ANSI terminal is a terminal that conforms to those standards.

American Standard Code for Information Interchange (ASCII)

In the Stratus internal character coding system, the half of the 8-bit code page with code values in the range 00-7F (hexadecimal), representing the American Standard Code for Information Interchange.

ANSI

See **American National Standard Institute**.

argument

A character string that specifies how a command, request, subroutine, or function is to be executed.

argument list

In VOS PL/I, a list of arguments to be passed to a called procedure. The argument list corresponds to a parameter list in the called procedure.

ascending key

A key whose values are used to order data, starting with the lowest value of the key up to the highest value, in accordance with the rules for comparing data items.

ASCII

See **American Standard Code for Information Interchange**.

ASCII string

A character string that contains only ASCII data. It therefore contains no shift characters and no characters from the right-hand graphic sets. However, it may contain generic input or generic output sequences.

assembler

The VOS assembly-language translator. The assembler takes a text file of assembly-language statements as input and generates an object module as output.

asynchronous communications

Data transmission where the time interval between the transmission of characters can vary from one character to the next. The beginning and end of each character transmitted is determined by means of start and stop bits preceding and following each character.

attach

1. To associate a port with a file or device, creating the port, if necessary, and generating a port ID. The port ID can then be used to refer to the file or device.

2. To associate an event with a process.

automatic storage

In programming languages, storage that is allocated within a stack frame at the time it is pushed onto the stack. If you do not specify a storage class for a variable that does not appear in a parameter list, automatic is the default.

backbone ring

A StrataLINK cable used to connect individual systems into a cluster configuration. Each system is connected to the backbone ring by means of a bridge module. (Note that these systems may be single- or multiple-module systems.) See also **ring**.

bad block

A block that the operating system has identified as unusable. The operating system keeps a list of bad blocks in the disk label.

batch process

A noninteractive process that executes a command or command macro. When you request a batch process, the operating system puts the batch request into a specified queue and starts a batch process to execute the command whenever resources become available. The batch process runs independently of the process that issued the batch request.

batch processor

A VOS facility that creates processes to execute commands for users independently of their interactive processes.

batch queue

A file that holds batch requests while they await processing.

batch request

One or more commands that are executed in a batch process.

binary

Base 2; a base designation for arithmetic data.

binary file

A file in which a data item is stored in its internal form, as opposed to its character form. In the VOSprogramming languages, structures, records, arrays, strings, arithmetic values, logical values, and characters can be stored in binary files. See also **text file**.

bind

To combine a set of one or more independently compiled object modules into a program module. Binding compacts the code and resolves symbolic references to external programs and variables that are shared by object modules in the set and in the object library. See also **library**.

bind map

A file, produced by the binder, containing information about a program module. A bind map is only produced if you specify the `-map` option of the `bind` command.

bind time

The time at which the binder is invoked to combine one or more user object modules and VOS support routines into a program module.

binder

The program that combines a set of independently compiled object modules into a program module. The binder is invoked with the `bind` command.

binder control file

A text file containing directives for the binder.

bit

1. The smallest unit of internal computer storage. A bit can have one of two values: `'1'b` or `'0'b`.

2. A bit string of length one.

block

Part of a file or directory in secondary storage, usually on a disk or magnetic tape. A disk block holds 4096 bytes. It is the smallest addressable unit in disk secondary storage and is the unit of transfer of data between main storage and secondary storage. File and directory blocks are read into the buffer pool before being accessed by the system.

blocked

A record format in which each block in a tape file can contain more than one record or record segment. See also **unblocked**.

board

An electrical panel containing circuits.

boot

To load a copy of the operating system into main storage so that it can execute on the module; to start the operation of a system. The bootload sequence consists of several stages, including hardware test, system initialization, and network connection.

boot disk

Each module has at least 1 boot disk which is always member number 0. The boot disk contains a copy of the operating system that is needed to boot the module.

boot partition

A partition that can contain a copy of the operating system. Only a boot disk has boot partitions.

boot tape

A tape that contains a copy of the operating system plus a complete copy of the contents of the boot disk. It is created when the `dump_disk` command executes on a boot disk.

boot volume

A logical volume that contains the boot disk.

bootload

See **boot**.

break

1. A signal (or to send a signal) that interrupts a program being executed and places the process executing the program at break level.

2. To bring the communications line temporarily to a low voltage condition in order to request attention from the running program or the operating system.

break characters

Input characters that cause an interrupt and terminal event notification when a terminal is in break table mode. Break characters are used to delimit lines (records and packets) coming in from a terminal as raw input.

breakpoint

A statement or instruction in a program being debugged at which the debugger stops execution of the program in order to examine the state of the program (for example, to examine variables or the call stack). Breakpoints are set with the debugger.

bridge

1. A module used to connect a system to a backbone ring.

2. In communications, a physical device that interconnects networks using the same link-level protocol.

bridge module

1. A module used to connect a system to a backbone ring.

2. A module in a remote system containing the hardware and software used to communicate with the HUB in a Remote Service Network.

buffer

1. For a terminal, a temporary data storage location in the terminal's memory. The buffer can be used to compensate for differences in transmission rates or temporarily store characters until the computer can accept them.

2. A temporary storage area for input or output data.

buffer pool

A noncontiguous region of main storage in which file and directory blocks are allocated. The operating system performs all file and directory I/O using the buffer pool.

built-in functions

Functions that are predefined within the VOS PL/I language.

bus

A network topology in which all stations attach to a single transmission medium. All stations have equal access to the carrier and can monitor it for the presence or absence of signals.

busy

In TPF, a message that is being processed in a queue is termed busy.

by-reference

A method of argument-passing in which the called procedure receives the address of the actual storage of a program variable.

by-value

A method of argument-passing in which the called procedure receives the address of a temporary storage location rather than the address of a variable's storage.

byte

1. Eight bits of data. An unsigned byte variable can contain integer values in the range 0 to 255; a signed byte variable can contain integer values in the range -128 to 127.
2. The unit of storage consisting of eight contiguous bits.

CAC

See **Customer Assistance Center**.

cache

In disk I/O, an area of memory where information read from disk is stored. It is used to reduce the amount of disk I/O necessary for file access.

cache hit

When a process performs a read or write operation and the requested file block is in the cache.

cache miss

When a process performs a read or write operation and the requested file block is **not** in the cache.

call

To activate a program block; usually by means of a `call` statement. **Note:** Only subroutines can be activated by a `call` statement.

called procedure

A procedure activated by the `call` statement or by a function reference.

caller

See **calling process** and **calling procedure**.

calling procedure

A procedure containing the `call` statement or function reference that activates another procedure.

calling process

The process that invoked the program currently being executed.

canonical string

A character string with no default character set and no locking- shift characters. All non-ASCII characters are therefore preceded by a single-shift character. It may contain generic input or output sequences.

channel

A C-Series hardware-addressable connection point to which a device (such as a terminal or printer) can be connected via a cable. The channel is associated with a particular connector, which is in turn associated with a particular electrical interface (such as RS-232-C). There can be up to 32 channels on a communications controller (or duplex pair of controllers). The channel should be clearly differentiated from the device connected to the channel. (The device may be a terminal, a printer, another computer, or another channel on the same computer.)

character

1. A symbol, such as a letter of the alphabet or a numeral, or a control signal, such as a carriage return or a backspace. Characters are represented in electronic media by character codes.

2. A character code.

character code

1. A numeric value used to represent a character according to a specified system. For example, the ASCII (character) code for `A` is `41` (hexadecimal). In the VOS internal character coding system, the bit representation of a character code can occupy one or two bytes, depending on the character set.

2. A particular system used to assign numeric values to characters (for example, the American Standard Code for Information Interchange (ASCII)).

character string

An ordered set of characters from one or more character sets. The length of a character string depends on the size of each character (one or more bytes, depending on the character set), the number of characters in the string, and the use of single-shift and locking-shift characters within the string. Character strings are evaluated from left to right.

clean up

In TPF, to close and detach a task's terminal device and prepare the task to be reinitialized.

clear a device

To release a device and make it ready for another I/O operation. Clearing a device is similar to closing a file.

close

To disconnect from an operating system file or device.

cluster

A network configuration in which two or more systems are connected via a StrataLINK backbone ring, forming a local network. Each system (either a multiple-module system, also known as a subring, or a single-module system) is connected to the backbone ring by means of a bridge module.

code

1. Machine instructions generated by the compiler.
2. The contents of a source module.
3. To write (in) a source module.

code page

An assignment of a character to each full set of code values that can be represented by a specified number of bits; that is, a complete set of code points. For example, an 8-bit code page consists of 256 distinct code points; a 14-bit code page consists of 16,384 code points.

code point

The assignment of a character to one of the set of code values that can be represented by a specified number of bits. See also **code page**.

code region

The portion of a standard program module that contains the actual instruction sequences that represent the program.

collating sequence

An ordered series of symbols, usually including lower- and upper-case letters, digits, punctuation symbols, etc.; usually for purposes of sorting, merging, or comparing.

command

A program invoked from command level, either interactively or as a statement in a command macro.

command function

A function that can be invoked when a process is at command level. An example of a VOS command function is `(time)`, which the operating system replaces with the current time. A command function must always be enclosed in parentheses.

command level

The state at which you can issue commands to the command processor or run programs.

command library

A directory that the operating system searches for command macros and program modules.

command line

A set of one or more commands, separated by semicolons. Pressing one of the following keys terminates a command line: RETURN, ENTER, DISPLAY_FORM (and, on V101 terminals only, EGI, ESI, EMI).

command line storage areas

Two storage areas, called the insert saved buffer and the insert default buffer, that save text you enter at command level, break level, and prompt level. The insert saved buffer saves text you enter with the RETURN, DISPLAY_FORM, or ENTER key. The insert default buffer saves text you enter with the RETURN or DISPLAY_FORM keys. You retrieve text from the insert saved buffer by pressing INSERT_SAVED, and from the insert default buffer by pressing INSERT_DEFAULT.

command macro

A text file that is a user-written program, invoked in the same way that you invoke a command. A command macro is composed of calls to VOS commands, calls to user programs and commands, and command macro statements. The command processor reads and executes lines from the macro file until one of the following occurs: it reaches the end of the file, it reaches the macro statement `&return`, or some program terminates in such a way that the file is abandoned.

The name of a command macro must end with the suffix `.cm`.

command macro statement

A VOS command that you can issue only in command macros. The name of a macro statement is a keyword beginning with an ampersand (`&`).

command name

The name of a program or command macro that can be called/invoked as a command.

command processor

The part of the operating system that accepts and executes commands. The command processor replaces abbreviations and evaluates command functions in a command before loading the program specified in the command.

command string

A command name and any command arguments and/or macro parameters you specify.

comment

Documentary information included in source code that is ignored by the compiler; it has no effect on the execution of the program.

common string

A string that can contain characters from any right graphic character set and that has a default character set of Latin alphabet No. 1. A single-shift character precedes each right graphic character except for Latin alphabet No. 1. A common string contains no locking-shift characters, but can contain generic input or generic output sequences.

communications controller

A main chassis controller board that connects terminals, printers, and synchronous devices to a module via line adapters. See also **I/O processor**.

communications line

The physical medium connecting one location to another for the purpose of transmitting or receiving data.

communications line adapter

See **line adapter**.

compile time

The time at which a compiler is invoked to translate a source module into an object module (program).

compiler

A program that translates a source module (source code) into machine code. The generated machine code is stored in an object module.

compiler directive

Statements that are not directly compiled, but cause the compiler to perform specific actions.

concatenate

To join end-to-end.

condition

An exceptional occurrence during the execution of a program. In VOS PL/I, a program can be set up to detect a condition by using the `on` statement. Execution of an `on` statement establishes an on-unit as a routine to be executed when a specified condition occurs. The other VOS languages use service subroutines, such as `s$enable_condition`, to work with conditions.

condition handler

A routine, defined by a program, that is invoked to respond to a condition signaled during the execution of the program. In VOS PL/I, an on-unit.

configuration table

One of the table files that the operating system uses to identify the elements of a system or a network. For example, the file `devices.table` contains information about each device present in the system, the file `disks.table` contains information about each of the disks present in a system, and the file `modules.table` contains information about each module in the system.

constant

In programming languages, A value that cannot be changed.

continue

In TPF, to allow a paused task to continue waiting or running.

control character

A character in the VOS internal character coding system that can perform a specified control function. There are two sets of control characters: ASCII and Stratus-specific. The ASCII control characters are represented by the hexadecimal character codes 00 to 1F. The Stratus-specific control characters are represented by the hexadecimal character codes 80 to 9F. Unlike graphic characters, control characters are non-printing characters. See also **control function** and **graphic character**.

control function

An action that affects the recording, processing, transmission, or interpretation of data. A control function has a coded representation consisting of a 1-byte bit combination in the VOS internal character coding system. See also **control character**.

control key

A special key found on many terminals. When it is held down, certain keys perform control functions. The control key is denoted by the caret symbol (`^`); for example, `^D` means “hold the control key down and press `D`." The key face is usually labeled CTRL.

conversion

The process of transforming a value from one data type to another.

counter

A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

current directory

The directory currently associated with your process. The operating system uses your current directory as the default directory when you do not specifically name the directory containing an object that you want the operating system to find. For example, if you supply a relative path name in a command, the operating system uses the current directory as the reference point from which to locate the object in the directory hierarchy.

When you log in, your current directory is set to your home directory. You can change your current directory with the `change_current_dir` command or the `s$change_current_dir` subroutine.

current index

The file index that was used for the most recent keyed access to that file.

current module

The module associated with a process; the module on which the process is executing. You cannot change the current module of a process.

current position

1. For files other than pipe files, the file's current position is the location in a file at which the next record operation will be performed.

   Pipe files have two current positions. These are: (1) the position in the file at which data will be written by the next output operation and (2) the position in the file at which data will be read by the next input operation. The two current positions of a pipe file are the same for all processes attached to the file.

2. In a record file, the current record; in a stream file, the current line and column.

current process

The process at which you are working while performing the procedure being described.

current record

The record to which the current file position is set.

current system

The Stratus system associated with a process; the system containing the process's current module.

cursor

A marker on the display, commonly a blinking rectangular block or underline, that shows where the next character typed will appear.

Customer Assistance Center (CAC)

The central point in a Remote Service Network. The CAC is often referred to as the hub.

cycle field

In a form, a field that has a specific list of possible values. The user cannot type in a cycle field but can change the displayed value by using the CYCLE key and other keys.

cycle list

The list of possible values for a cycle field.

DACL

See **default access control list**.

data type

The collective attributes of a value, variable, or object that determine the operations that can be performed on that value, variable, or object and the scheme by which the value, variable, or object is stored.

deadlock

A situation in which a lock contention cannot be resolved.

debug

To correct errors (bugs) in a program.

debugger

A VOS tool used as an aid in finding program errors.

decimal

Base 10; a base designation for arithmetic data.

declaration

A statement that introduces a name and defines its attributes.

default

The value or attribute used when a necessary value or attribute is omitted.

default access control list (DACL)

A list that the operating system uses to determine the access rights of a user to the files within a particular directory. The operating system looks at the DACL if the user's access cannot be determined by looking at the file's ACL. (See **access control list**

(ACL).) The use of DACLs allows you to set up default access for all of the files within a directory.

A DACL is a list of entries, with each entry showing an access code and a user name. An example of a DACL follows.

```
w    Jones.sales
r    *.sales
n    *.*
```

default boot partition

The boot partition that contains the copy of the operating system used during automatic startup. This partition is identified by a special field in the disk label.

default character set

In NLS strings or in text files, the supplementary graphic character set that, in the absence of single- or locking-shift characters, is represented by the character codes in the range A0-FF (hexadecimal). The default character set for a file can be set by the `create_file`, `set_text_file`, or `emacs` commands. Some of the NLS built-in functions in VOS programming languages permit specification of the default character set as an argument. If a default character set is not specified, the default is usually assumed to be Latin alphabet No. 1.

default priority

The priority that an interactive process started by a particular user will have if the user does not specify a different priority.

default priority level

The priority assigned to your initial interactive process unless you specify a different priority.

default value

The value that the operating system uses if a specific value is not supplied.

defined storage

In VOS PL/I, storage that is not allocated separately but is overlaid on storage that has already been allocated. A variable has the defined storage class only if you give the `defined` attribute in its declaration.

definition

In C, a definition provides a complete description of the data and allocates storage.

detach

1. To disassociate a port from a file or device.

2. To disassociate an event from a process.

device

Any hardware component that can be referenced and used by the system or users of the system and that is defined in the device configuration table. Terminals, printers, tape drives, and communications lines are devices. The term *logical device* indicates that a device can be a software entity; for example, a virtual terminal is a logical device.

device configuration table

See *devices table*.

devices table

A file with the name `devices.table` that contains information about each device present in a system. See also **configuration table**.

`devices.tin` **file**

The table input file that contains definitions of all the devices, except disks, attached to a system. These devices can be I/O adapters, line adapters, terminals, printers, or tape drives. The VOS operating system uses this file to create the devices table. See also **devices table**.

diagnostic

A message from the compiler warning of a possible error.

diagnostic partition

A region of a disk that the operating system uses to test that disk. Every disk has a diagnostic partition.

direct access

A method of referring to records by giving their ordinal position within a file. See also **keyed sequential access** and **sequential access**.

direct queue

In TPF, a fast, memory-resident, one- or two-way queue. The direct queue has a depth limitation of one message per port attached, a message length limitation of 3072 bytes, and cannot be transaction protected.

directory

A segment of disk storage containing files, links, and subdirectories and having its own access limitations.

directory access control list

An access control list that is associated with a directory and contains an ordered set of entries that defines the access rights of users to that directory.

directory hierarchy

The structure of the set of directories subordinate to a disk.

disk directory

The top directory on a disk. In a path name, the disk directory name is prefixed by a number sign (#).

In UNIX, the disk directory is called the **root**.

disk label

Information placed in the first sector or sectors of a disk that identifies the contents and name of the disk.

disk name

The name of a disk. Disk names are specified by the system administrator in the disks configuration table. (See **configuration table**.) The path name of a disk has two components: (1) the name of the system, prefixed by a percent sign, and (2) the name of the disk, prefixed by a number sign (for example, `% s1# d01)`.

Note that module names and device names have the same form as disk names.

dismount

To close all files on a logical volume while leaving the disk contents in a consistent state. The operating system will recognize the logical volume, but the file system cannot use a dismounted logical volume.

display attribute

A visible characteristic of data on the screen (for example color, highlighting, underlining, or blinking).

distributed

Refers to applications that are spread among multiple processes and/or processors, and possibly multiple machines, that share the load of an application.

driver

A program that transmits information between input and output devices and the operating system. Typically, a driver controls how instructions are sent to and information received from the input or output device.

dump

See **dump image**.

dump image

All modified disk blocks of the operating system and selected disk blocks of processes that were running at the time of a module failure.

dump partition

A partition that is available to hold a dump image in the event of module failure. The dump partition is a holding place for dump images until they can be copied into files.

Dump partitions exist only on master disks; each master disk has one dump partition.

duplex disk

Two physical disks that are used as a member disk, and are known to the system and to users by the same member disk name. These disks are known as partners of the duplex disk.

dynamic task

In TPF, a task that can be created or deleted at run time.

EBCDIC

See **Extended Binary Coded Decimal Interchange Code**.

entry data

In VOS PL/I, entry point constants, entry point variables, and entry point values.

entry point

1. A statement in a program at which execution of the program can begin. An entry point can be the target of a call statement and it can be specified in a binder control file.

2. A location within a procedure where execution can begin when the procedure is activated. In VOS PL/I, each `procedure` and `entry` statement is an entry point.

error code

An arithmetic value (usually, a two-byte integer) indicating what, if any, error has occurred; usually, a VOS status code. An error code argument is often included in subroutines. See also **status code**.

error message

A character string that is associated with an error code.

Ethernet

A local area network based on the specifications published by Digital Equipment Corp., Xerox, and Intel. It is a baseband communications system employing a bus topology. IEEE 802.3 defines Carrier Sense Multiple Access/Collision Detection (CSMA/CD) as the access control method for Ethernet. Ethernet can have a maximum length of 1500 meters.

event

1. In a Stratus system, a data structure, associated with a file or device, that is used by processes to communicate with one another. When one process notices that some action has occurred that is of potential interest to one or more processes, the first process notifies the event. This notification causes the operating system to inform all processes that have been waiting for the action. (These processes are said to be *waiting on the event*.) Each time an event is notified, its event count is incremented.

2. An occurrence of significance to a task; for example, the completion of an asynchronous operation, such as an input/output operation.

event attachment

The association of an event with a process.

event count

An integer value that is associated with an event. Each time the event is notified, the event count is incremented.

event notification

The modification of an event's data structure, which causes the operating system to notify any processes waiting on the event.

event status

An integer value, associated with an event, that can be used to tell waiting processes the reason for the event's notification.

executable files

The executable files, located in the `>system>wordperfect>commands` directory, contain the software program code.

executable image

The object created from a program module by the loader. The modifiable part of the executable image does not reside in the file hierarchy, but rather in temporary paging storage that is shared by many processes. Parts of an executable image can be swapped in and out of main storage.

executable program

See **executable image**.

executable statement

A source code statement for which the compiler generates machine code.

execute

To process an executable statement at run time.

To run a program.

execute access

A type of file access that means a user can execute a program module or a command macro, but cannot read or write the file.

execution time

See **run time**.

expansion cabinet

A cabinet that can be added to one of the modules. An expansion cabinet can contain a communications chassis, StrataHUBs, tape drives, or disk drives.

Extended Binary Coded Decimal Interchange Code (EBCDIC)

A coded character set consisting of 8-bit coded characters.

extent-based file

A file comprised of one or more extents, which are sets of contiguous blocks on disk. For this type of file, the file map records the address of each extent rather than the address of each block, as in traditional (non-extent) VOS files. This enables creating larger files. File types that can be extent-based include fixed, sequential, stream, and relative files.

external

1. Usable throughout a program; visible to all blocks of a program. See also **internal**.
2. Not nested.

external definition

In C, a definition of an object whose scope extends outside of one or more functions. An object defined with the keyword `static` is external to functions within the file it is defined. An object defined with the keyword `extern` is external to all functions in the program.

external variables

Variables that can be shared by more than one program.

false

Unambiguously incorrect; the opposite of true; evaluating to a bit-string value of `'0'b` or to a value of 0.

fault tolerant

A system containing duplexed hardware components that operate in lock-step simultaneously. Processing continues even if a component fails.

fence

A portion of a user's virtual address space between the stack and the heap.

file

A set of records or bytes stored on disk or tape as a unit. A disk file has a path name that identifies it as a unique entity in the system's directory hierarchy. Attributes of a disk file, such as its size and when it was created, are maintained in the directory containing the file.

file block

See **block**.

file index

An ordered list of keys associated with a file. Each key has an associated descriptor that usually locates that key's record. A file index has a name. A file can have more than one index.

There are four types of file indexes.

- Separate-key index, in which the value of each key is independent of the value of its associated record
- Embedded-key index, in which each key is the concatenation of substrings of a record
- Embedded-separate-key index, which is an embedded-key index to which separate keys can be added and deleted
- Item index, in which each entry is a four-byte value that has an associated separate key

file organization

The manner in which data in a VOS file is arranged. The operating system supports four file organizations.

- `fixed`
- `relative`
- `sequential`
- `stream`.

Note that VOS Pascal does not support the stream file organization.

file partition

A partition that provides permanent storage for files, directories, and indexes. All disk space not occupied by other partitions is given to the file partition. Every disk has a file partition.

file system

The part of the operating system that manages files in the directory hierarchy.

In UNIX, a file system is associated with an entire logical device.

first directive

A type of replacement directive in an abbreviations file that tells the operating system to replace an input string with an output string when the specified input string occurs first in a command string. See also **abbreviation**, **all directive**, **replacement directive**, and **subsequent directive**.

fixed file

A file with a fixed organization. In a fixed file, the records all have the same size. Each record is stored in a disk or tape region holding a number of bytes that is the same for all the records in the file. See also **relative file**, **sequential file**, and **stream file**.

fixed organization

A VOS file organization in which data are stored in records of equal length. See also **relative organization**, **sequential organization**, and **stream organization**.

fixed-point number

An arithmetic value having a predetermined and inflexible number of digits to the right of the radix point.

floating-point number

An arithmetic value consisting of a mantissa and an exponent.

Forms Management System(FMS)

A set of tools, system software, and programming language extensions that gives application programs a consistent interface for the entry and display of data on a video display terminal.

format

For disks, to format a disk is to destroy its contents, then record control information on it so that it can be addressed by the operating system. (Note: All disks delivered from Stratus are already formatted.)

full path name

For a file, directory, or link, a name that is composed of the name of the system, the name of the disk, the names of the directories that contain the object, and finally the name of the file, directory, or link.

For a device, a name that is composed of the name of the system and the name of the device.

A full path name refers to only one object; an object has exactly one full path name. (However, many links can refer to the same object.)

gateway

Under the operating system, a gateway is a process running on a Stratus module that serves as the interface between an external network and the system containing the module. Each gateway runs a synchronous channel that converts VOS virtual circuits to X.25 virtual circuits. A module can contain up to 64 gateways, with one gateway for each X.25 communications line.

generic sequence introducer (GSI)

In the Stratus asynchronous communications, the first character in every generic input and generic output sequence: the ASCII character ESC (hexadecimal code 1B).

graphic character

A symbol, such as a letter of the alphabet, a numeral, or a punctuation mark, as opposed to a control character. In the VOS internal character coding system, graphic characters are represented by codes in the ranges 21-7E and A0-FF (hexadecimal).

group

A set of users whose directories are contained in the same group directory.

group directory

A set of user directories, combined for the purposes of allocating system resources and simplifying access control. A group directory is a directory immediately subordinate to a disk. It has the same name as the group and contains the user directories of the members of the group.

group name

The name that identifies the group directory to which a person registered to use the system belongs. A group name is one of the two parts of a user name.

heap

A collection of randomly accessible storage associated with a process and available for allocation. See also **user heap**.

hexadecimal

Base 16; a base designation for arithmetic data.

home directory

A directory that is your current directory when you log in and that the operating system uses as the default directory in which to place certain per-user files.

A directory can be specified as your home directory in the registration databases. If the databases do not specify a home directory, your default home directory is a subdirectory of the group directory that you logged into, and is named with your person name.

host

In the context of networking, any processor attached to and accessed from a network.

I/O

Input and output.

I/O adapter

A K-Series adapter (card) that is connected to an I/O processor pair. There are four types of I/O adapters:

- communications
- tape
- disk
- terminator.

I/O processor

A K-Series controller board that manages communications I/O adapters, tape I/O adapters, disk I/O adapters, and Ethernet I/O adapters. It resides in the main chassis of an equipment cabinet.

implicit locking

A VOS locking mode in which the operating system does not lock the file or record for either reading or writing when it opens the file, but rather locks it for the appropriate access type each time a process performs an I/O operation on the file or record.

inactive directory

A directory on which no directory operation has recently been performed or that has been explicitly deactivated by a program call to the VOS system subroutine `s$deactivate_dir`.

include file

1. A file that the compiler includes in the source module used by the compilation process. The name of the include file must be specified in a language-specific directive within the source module.

2. A text file containing language statements, compile-time statements, or both, that the compiler inserts into the source module in place of an `%include` compile-time statement. For example, VOS PL/I include files have the suffix `.incl.pl1`.

include library

A directory that the operating system searches for include files.

index

1. In relation to a file, an ordered list of keys associated with the file. See also **file index**.

2. In relation to an array, a number used to specify an element of the array.

3. In VOS PL/I, in relation to a `do` statement, a variable used in controlling the execution of the do-group.

4. In TPF, in relation to a file, an ordered list of keys associated with the file. See also **file index**.

index key

A file index value associated with a particular record.

index order

The order in which the keys of an indexed file are sorted. When you use the VOS `create_index` command, you can specify the collation scheme and direction in which the keys are to be sorted.

indexed access

A method of referring to records in a file by their index-key value. See also **random access** and **sequential access**.

indirection

In C, the unary `*` operator results in indirection; the operand must yield the address of an object, which the indirection operator references.

insert default buffer

A command line storage area that saves text you input with the RETURN key or the DISPLAY_FORM keys.

insert saved buffer

A command line storage area that saves text you input with the RETURN, DISPLAY_FORM, or ENTER keys.

integer

A whole number; an arithmetic value with no fractional part.

interactive process

A process started with the `login` command. A user working with an interactive process is engaged in a dialogue with the operating system, issuing commands and receiving responses, normally from a CRT terminal.

internal

Usable only within the block in which it is declared and in blocks contained within that block. See also **external**.

internal command

A command that is built into the operating system. For a list of VOS internal commands, give the `help` command with the argument `-type internal`.

K

When referring to memory or storage capacity, 2 to the 10th; 1024 in decimal notation.

keep module

A file that allows for future debugging by retaining an executable image of a program after the break signal is given or after a fault occurs. If the program running is in an interactive process, the operating system produces the keep module only if you choose the `keep` option. If the program running is in a batch process, the operating system automatically produces the keep module.

All keep modules have the suffix `.kp`. The name of the keep module is the same as the program name, but with the suffix `.kp` instead of `.pm`.

kernel

The part of the operating system that performs all privileged system operations. The command processor and the debugger are included in the kernel.

key

An identifier used to locate records in an indexed file. Keys are stored in a file index where they are ordered according to a definite collating sequence. There are two types of keys: an embedded key, which gets its value from the data in the field in the record, and a separate key, which gets its value independently of the record's literal value.

keyed file

A file in which an index has been created, or a file of records that can be accessed directly.

keyed sequential access

A method of referring to records in a file by their index-key value; if no key value is given, records in a keyed sequential file are accessed in index order. See also **direct access** and **sequential access**.

label

1. For an internal command, the name of a command argument. In a command macro, an optional element of a parameter descriptor that provides a descriptive term for documenting the macro.

2. A statement label.

3. A special record appearing at the beginning of a volume of tape and at the beginning and end of files on the volume. Labels provide details about the formats of files on the tape.

Latin alphabet No. 1 character set

A character set located in the right-hand side of the VOS internal character coding system in the range A0x to FFx. At Stratus, the left-hand side is occupied by ASCII.

left control character

An ASCII control character. See also **control character**.

left graphic character

A character located in the range 21x to 7Ex in the VOS internal character coding system. Left graphic characters compose the ASCII character set.

library

One or more directories in which the operating system looks for objects of a particular type. There are four types of libraries defined by the operating system:

- include libraries, in which the compilers search for include files
- object libraries, in which the binder searches for object modules

- command libraries, in which the command processor searches for commands
- message libraries, in which the operating system searches for message files associated with individual `.pm` files.

One of each of these libraries is available in the `>system` directory of each module for all processes running on the module. In addition, you can define your own libraries.

library paths

The set of directories associated with a particular library. Each process has one set each of include library paths, object library paths, command library paths, and message library paths.

line adapter

A small board containing electrical interfaces that communications devices can be plugged into.

link

1. An object in a directory that directs all references to itself to a file, directory, or another link. Like many other objects, a link has a path name that identifies it as a unique entity in the system directory hierarchy.

2. A cable that connects one location to another for the purpose of transmitting information.

3. See **bind**.

link servers

Servers that operate within local networks to exchange messages with clients via the StrataLINK local network.

load control facility

A VOS facility that monitors the processing load on a module and initiates action to increase or decrease the load.

loaded image

See **executable image**.

loaded program

See **executable image**.

loader

The part of the command processor that converts a program module (created by the binder) to an executable image. The loader relocates many pointers in the static region, allocates and initializes the heap and stack, resolves external references to kernel entry points, and manages the sharing of programs and data.

local area network (LAN)

See **local network**.

local network

A high-speed network that connects computers and other data processing components located in a limited geographical area (typically, the same building or neighboring buildings) in order to share resources. For example, StrataLINK is the Stratus local network facility that connects modules within a system and/or connects systems located at the same site, and provides transparent use of the operating system among Stratus modules and systems. See also **cluster**, **ring**, **system**, and **wide area network**.

A local network is also called a local area network (LAN).

lock

1. A system data structure associated with a file, file record, or device that can be set to restrict the use of the object; the restriction remains in effect until the lock is reset.

2. The act of setting a lock. Setting a lock is called locking the lock; resetting it is called unlocking the lock.

lock wait time

In TPF, the maximum time that a task or process will wait to acquire an implicit lock during any I/O operation.

locking-shift character

A user-transparent character in an array or string, indicating that the remaining characters in the key or record are from a character set other than the default character set.

logical disk

See **logical volume**.

logical volume

A grouping of up to ten member disks. With multi-member logical volumes, you can create files which are larger than the capacity of a single physical disk.

logical volume name

The name of the group of up to 10 member disks; the name of the top directory in the logical volume's directory hierarchy.

login terminal

A terminal from which a valid user can execute, or interrupt or stop execution of VOS commands, command macros, or program modules.

login virtual terminal

Under the operating system, a logical device that is used to handle VOS X.25 output to an actual (remote) terminal. Remote terminals are assigned to a login virtual terminal, which causes the standard VOS pre-login sequence to be displayed at the actual terminal device. Virtual terminals provide an interface that insulates VOS terminal I/O from network I/O. See also **slave virtual terminal**.

long-haul network

See **wide area network**.

long word

Four bytes of data aligned on a two-byte boundary. The integers that can fit in a long word range from 0 to 4294967295 and from -2147483648 to 2147483647.

loop

A series of statements, like a do-group, that are executed repeatedly because some statement within the series, or some loop controller, transfers control back to the beginning of the series.

lower bound

The lowest subscript for a particular dimension of an array.

macro event

A means of building a two-dimensional array of events. When an event occurs, notification is issued not only for the event that occurs, but also for the macro event.

macro parameter

A named variable that is declared within the parameter declaration section of a command macro.

macro processor

The part of the operating system software that reads macro files and processes macro statements. The macro processor does not process command lines and input lines that it encounters in macros; instead, it returns these lines to the command processor. The operating system starts up the macro processor when a command macro is issued.

macro variable

A named variable in a command macro that is declared and given its initial value in one of the macro assignment statements, `&set` or `&set_string`.

master disk

The disk from which the module was booted.

master volume

The logical volume which contains the module's master disk.

maximum priority level

The highest priority level that you can assign to any of your processes.

member disk

A duplex disk pair or a non-duplex disk which is a member of a logical volume. Each member disk is known by a member number, between 0 and 9 inclusive, which describes its location within the logical volume.

member disk name

The name of the logical volume of which the disk is a member, with the disk's member number appended as a suffix.

member number

A number from 0 to 9 that is assigned to a member disk when the disk is initialized.

message

A line of text that the operating system displays to provide you with information. Messages are one type of status code. Each message has an associated code and an associated name that begins with the prefix `m$`.

modem

A device that converts data from a form that is compatible with data processing (digital) to a form that is compatible with transmission facilities (analog), and vice versa.

modify access

A type of directory access that means a user has full access to the contents of the directory, including the ability to create, delete, and rename objects.

module

A single Stratus computer. A module is the smallest hardware unit of a system capable of executing a user's process.

module ID

See **module name**.

module name

The name of a module. Module names are specified on the module's disk label and by the system administrator in the modules configuration table. (See **configuration table**.) The path name of a module has two components: (1) the name of the system, prefixed by a percent sign, and (2) the name of the module, prefixed by a number sign (for example, `% s1# m5)`.

Note that device names and disk names have the same form as module names.

module star name

A name that contains one or more asterisks or consists solely of an asterisk, used to specify a set of modules in a system. An asterisk can be in any position in the name, and each asterisk represents zero or more characters. A module star name can be used alone or as part of a full module path name; however, in a full module path name, there can be no asterisks in the system name. When a module star name consists solely of an asterisk, it represents all modules in the current system. See also **star name**.

`module_start_up.cm` **file**

A command file that the operating system reads when starting up a module.

modules table

A file with the name `modules.table` that contains information about each module present in a system. See also **configuration table**.

monitor task

In TPF, a task, executing either of the subroutines `s$monitor` or `s$monitor_full`, which accepts interactive requests to control tasks or display information about them.

monitor terminal

A terminal that displays information about the module to which it is connected.

mount

To bring a disk into the mounted state.

multicluster

A network configuration in which a cluster is connected to the remote system(s) and/or cluster(s) via the StrataNET wide-area network.

network

A communications facility that connects two or more points. The Stratus network structure can consist of both local networks and long-haul networks.

network client

A process within a StrataNET network that passes requests for service to remote systems and receives replies from the remote systems.

network configuration

The arrangement or organization of the components in a network, including both the hardware and software needed to implement the configuration.

network server

A process within a StrataNET network that receives requests for service from network clients in remote systems and sends replies to the remote systems.

network watchdog

A process that executes on every module to oversee the functioning of both local and wide area networks. The network watchdog periodically checks to see if other modules and/or systems have lost or regained communication with its own system. It also cleans up file, device, and event attachments that have become inoperative due to communications failures.

network watchdog process

A system process that executes on every module to oversee the functioning of both local and wide area networks. It checks periodically to see if other modules in the network have either lost or regained communication with its own module.

newline character

A nonprinting character that you insert in a file by pressing the RETURN key; an ASCII LF (linefeed) character whose hexadecimal code is 0A.

next record

In a sequential access file, the record that follows the current record in storage; in a keyed sequential access file, the record that follows the current record in index order.

notify

To increment the event count of an event. When a process notifies an event, the operating system informs all of the processes that are waiting on the event.

null access

A type of file or directory access that means a user has no access to the file or directory.

null pointer value

A specific pointer value that does not address any storage. **Note:** an operating system null pointer has a value of 1.

null string

A character or bit string with zero length.

null value

In C, the value zero (0). This is a useful convention for pointers. A pointer containing the null value never locates a valid object.

object

1. In C, a region of storage that a program can examine and modify. In general, such an object is called an "lvalue".

2. Any data structure or device in the system that you can refer to by a name or some other identifier. For example, all of the following are objects: directories, files, links, systems, modules, devices, groups, persons, ports, queues, locks, file indexes, and file records.

3. In VOS PL/I, a named program entity (for example, a variable, file constant, entry point, statement label, format, or procedure).

object library

A series of directories that the operating system searches for object modules.

object module

A file produced by a compiler that contains the machine-code version of one or more procedures; it usually contains symbolic references to external variables and programs. To execute the program, an object module must be processed by the binder to produce a program module, and then loaded by the loader.

object name

A character string identifying an object. The maximum length of an object name is 32 characters. Examples of objects that have object names are files, directories, and links.

on-unit

In VOS PL/I, the statement, begin block, or the token `system` that appears within an `on` statement. The on-unit is executed only if the condition specified in the `on` statement is signaled.

opcode

In the operating system, an operation code; a 2-byte integer value passed to `s$control`. Each opcode directs `s$control` to execute a different device control operation.

open

To prepare a file or device for a particular type of access. The file or device must be attached to a port in the process before opening.

optional argument

An argument for which the operating system does not need a value to execute the command.

organization

The way records in a file are stored on a disk. The four types of organizations are sequential, relative, fixed, and stream.

output argument

A subroutine argument that can be altered by the called procedure. Output arguments must be passed by-reference.

owner

The author of a program module.

page

1. A unit of virtual storage containing 4096 bytes. A page is the smallest unit of storage that the operating system moves between main storage and secondary storage on a disk.

2. A series of lines, delimited by form-feed characters, that are written to a print file.

page fault

A program notification that occurs when an the program tries to access a virtual page that is not in main memory.

paging

Dividing a virtual address space into pages and managing pages (reading in from the disk to main storage and writing out to disk, when necessary) when a process refers to virtual addresses in the pages. Paging is invisible to users' programs.

paging partition

A partition that the operating system uses for temporary storage of a process. A boot disk must have a paging partition; it is optional on any other disk.

parameter

A value upon which a procedure operates. The actual value is supplied when the procedure is called.

parameter declaration

A line in a command macro that appears between the macro statements `&begin_parameters` and `&end_parameters`.

partition

A region of disk dedicated to a specific purpose. There are five types of partition: boot, diagnostic, dump, file, and paging.

password

A sequence of characters that a user can be required to supply when logging in. The characters in a password do not appear on the screen when they are typed in response to a prompt for a password.

path

An attribute of an object. The object's path is the sequence of objects (system, disk, directory) that are superior to the object in the directory hierarchy.

path name

A unique name that identifies a device or locates an object in the directory hierarchy. See also **full path name** and **relative path name**.

person name

The name that identifies a person to the system. When logging in, you must supply either the person name or the alias by which the system can identify you.

A person name is one of the two parts of a user name.

pipe file

A file that is used to connect processes, so that one process produces data for another process. A file is designated a pipe file with the command `set_pipe_file`.

PL/I

The full standard PL/I programming language, a subset of that language, or an implementation of either. In this manual, the term PL/I generally refers to VOS PL/I, an implementation of Subset G.

pointer

1. In COBOL, an elementary data item defined with the `usage is pointer` clause and used to reference a Linkage-Section data item that is not directly or indirectly specified in the `using` list of the Procedure Division header or of an `entry` statement.

2. In VOS PL/I, a storage address or a variable declared with the `pointer` attribute (the value of such a variable is a storage address); addresses are stored as four-byte integers.

3. The address of a storage location that contains an object of a particular type. In C there are pointer *variables* and pointer *constants*.

4. In Pascal, a pointer is a variable that either holds the address of another variable or has the `nil` value. If a pointer has the `nil` value, it does not point to another variable.

pop

To remove a stack frame from the top of the stack. This occurs when the block activation associated with the stack frame terminates. See also **push**.

port

1. A data structure, identified by a name or ID, that you attach to a file or device for the purpose of accessing the file or device. When an executing program refers to a file or device, the operating system uses the port having the same name or ID.

   A port is created when it is attached; a port is destroyed when it is detached.

2. A system data structure that can be attached to a file or device for the purpose of accessing data from the file or device.

   In the VOS programming langauges, a port is created automatically whenever you open a file.

port attachment

The creation of a port for the purpose of accessing a file or device.

port ID

A two-byte integer used to identify a port.

port identifier

A system-generated number corresponding to a port.

port name

The character-string name of a port.

position

The location of a record or character in a file.

positional argument

In command line form, an argument that does not have a keyword.

pre-login process

A process that the Overseer starts for a terminal for which login processes are enabled. From a pre-login process, you can issue only a few commands (such as `login`, `help`, and `list_users`).

precision

The total number of significant digits in an arithmetic value and, in the case of fixed-point decimal data, the scaling factor of the value.

primary index

A file's index that was defined in a previous keyed access to the file. When no other index is defined for the current keyed operation on the file, the operating system uses the primary index.

primary task

In a tasking program, the task with a task ID of 1.

priority level

A number ranging from 0 (the lowest) to 9 (the highest) that determines the precedence of a process relative to other processes for the allocation of CPU time.

privileged

An attribute of users that allows them to use certain commands, requests, and subroutines. To be privileged, users must be defined as such in the registration databases and must specify the `-privileged` argument when logging in.

privileged process

An attribute of users that allows them to use certain commands, requests, and subroutines. You can be privileged or not depending on your status as defined in the registration databases and on how you logged in.

procedure call

A jump to an executable procedure, the execution of the code, and then a return using the saved address.

process

The sequence of states of the hardware and software during the execution of a user's programs. When you log in, the operating system creates a process for you to control the execution of your programs. Your process can create other processes at your request. A process is always in one of three states: running, waiting, or ready.

process directory

A directory associated with a process that contains temporary information needed by the process. The process directories in each module are in the directory `process_dir_dir`, which is a subdirectory of the `(master_disk)` directory.

process lock wait time

See **lock wait time**.

process state

One of the three states--running, waiting, or ready--that a process can be in.

process terminal

The terminal of the primary task (Task 1).

program

One or more procedures, from one or more source modules, that together perform a task.

program entry

See **entry point**.

program module

A file containing an executable form of a program. The program module consists of one or more object modules (compiled source programs) bound together. The program module always has the suffix `.pm`.

program name

The full or relative path name that identifies a program module, optionally omitting the `.pm` suffix.

program termination

A program terminates when it calls `s$stop_program` (a VOS PL/I program terminates when it executes a `stop` statement), or when it finishes normal execution and returns to the command processor. Upon termination, the ports (files) used by the program are closed (and detached, if appropriate) and any locks locked by the process are unlocked.

programmer-defined condition

In VOS PL/I, an exceptional condition declared in a declare statement with the `condition` attribute and signaled only by the `signal` statement.

prompt

A message to the terminal user requesting further information before processing can continue.

push

To add a stack frame to the top of the stack. Each stack frame is associated with the activation of a block. A push occurs whenever a new block is activated. See also **pop**.

queue

1. A mechanism to record jobs that are waiting to be processed. Jobs can be given priorities within a queue.

2. A logical collection of messages awaiting transmission or processing.

queue depth

The number of messages in a queue.

queue priority

A number ranging from 0 (the lowest) to 9 (the highest) that determines where in the queue a newly issued batch request will be inserted relative to existing requests.

random access

An access mode in which the program-specified value of a key identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

raw I/O

In the operating system, the transfer of input and output characters between the user's buffer space and a device without any translation. The system subroutines `s$read_raw` and `s$write_raw` perform raw I/O.

read access

A type of file access that means a user can read the file or execute it if it is executable, but cannot write it.

read lock

In TPF, a lock that allows other tasks or processes to set a read lock on a given object but prevents them from setting a write lock. It allows a reader to ensure that an object will not be modified while it is being read.

receive

In TPF, to start processing a message by marking it as busy.

record

The data structure that the operating system uses to manage data in a file. For files other than stream files, a record is the smallest unit of data that the operating system I/O routines can access when performing I/O operations on files or devices. For stream files, a record consists of unstructured data. See also **stream file**.

record locking

A feature that protects two users against updating the same data at the same time.

recover

For a physical disk, to bring one of the partners of a duplex disk up to date with its partner.

recursion

The act of invoking a procedure or function from itself. This type of procedure or function that can invoke itself is said to be recursive.

registration databases

The databases `user_registration.sysdb` and `change_password_sysdb`, which together identify the users who are registered to use a system and the system resources available to each user. A system administrator uses the `registration_admin` command to add, delete, or change information in these databases, including person names, passwords, aliases, names of groups individual users are registered in, and the privileged or non-privileged status of individual users.

relative file

A file with a relative organization. In a relative file, the records can have varying sizes. Each record is stored in a disk or tape region holding a number of bytes that is the same for all the records in the file. See also **fixed file**, **sequential file**, and **stream file**.

relative organization

A VOS file organization in which data are stored as varying-length records in fixed-length fields. See also **fixed organization**, **sequential organization**, and **stream organization**.

relative path name

A name that identifies a device or an object in the directory hierarchy without specifying its full path name.

Remote Service Network

The remote network that connects any Stratus system to the Customer Assistance Center through a modem and the telephone lines. The RSN will automatically report hardware failures to the Customer Assistance Center.

replacement directive

A line of text in an abbreviations file that contains a string to be replaced (the input string) and a string to replace it with (the output string). See also **abbreviation**, **all directive**, **first directive**, and **subsequent directive**.

requester

1. In a system operator subsystem, a person who operates an interactive requester process.

2. In TPF, a process or task that sends messages to a queue and, for two-way queues, waits for replies.

requester process

A process within the system operator subsystem that issues a tape mount request on a module served by an operator process.

required argument

A command argument for which you must specify a value.

return

To terminate a procedure and transfer control back to the calling block.

revert

In VOS PL/I, to disestablish an on-unit; to undo an `on` statement.

right control character

A control character with a hexadecimal character code between 80 and 9F. See also **control character**.

right graphic character

A character located in the range A0x to FFx in the VOS internal character coding system. Right graphic characters compose the Latin alphabet No. 1, katakana, kanji, and hangul character sets.

right graphic character set

A character set located in the range A0x to FFx in the VOS internal character coding system. Latin alphabet No. 1, katakana kanji, and hangul are examples of right graphic character sets.

ring

The physical arrangement of network components in which the components are connected in a circular configuration and data is transmitted in one direction around the circle. For example, a multiple-module Stratus system (that is, a local network in which the StrataLINK network connects two or more modules into a system) is a ring. See also **backbone ring**.

root

The top of a directory hierarchy on a particular device. Also, the privileged user under UNIX.

run

To execute a program.

run time

The time at which a program module is invoked and executed.

salvage

To salvage a disk is to examine the disk for consistency and make it consistent where necessary.

scope

1. The region of a program in which a particular declared name is known.
2. An attribute of a declared name: `internal` or `external`.

secondary partner

One of the physical disks in a duplex disk pair. The secondary partner has the suffix `.sec` as part of its full name.

sector

The unit of allocation for disk packs. A sector can contain one page or one file block (4096 bytes).

send

In TPF, to add a new message to a queue.

sequential access

Accessing records in a file in the order in which they are stored. See also **direct access** and **keyed sequential access**.

sequential file

1. A sequentially organized file that contains records of varying sizes stored in a disk or tape region holding approximately the same number of bytes as the record. Thus, the record storage regions in a sequential file vary from record to record.

   Sequential files can only be accessed on a record basis, using `s$seq_read`, which reads the next record from the file. See also **fixed file**, **relative file**, and **stream file**.

sequential organization

A VOS file organization in which data are stored in varying-length records; each record is preceded and followed by two bytes representing its length. See also **fixed organization**, **relative organization**, and **stream organization**.

server

1. A system process responsible for receiving and responding to requests from clients.
2. In TPF, a process or task which waits for messages in a queue, receives them, processes them, and, for two-way queues, sends replies to them.

shared programs

Programs that can be shared simultaneously by several processes. All programs can be shared, unless users have set up access control or locking mechanisms to deny access to other users.

shared variable

1. A static external variable that can be shared simultaneously by several object modules. Shared variables are allocated in the virtual address space of one or more processes.

2. In a tasking environment, a variable allocated only once for all tasks. Unshared variables are allocated on a per-task basis.

shift character

See **single-shift character** and **locking-shift character**.

signal

In VOS PL/I, to invoke the on-unit for a particular condition.

simple string

A string in which Latin alphabet No. 1 characters are the only right graphic characters. A simple string contains no shift characters, but can contain generic input or generic output sequences. There is a one-to-one correspondence between bytes and characters.

single-shift character

A user-transparent character in a key or record, indicating that the next character is from a character set other than the default character set.

slave virtual terminal

Under the operating system, a virtual terminal that is tightly coupled with an application program. Incoming X.29 calls that specify a slave ID are assigned to a slave virtual terminal; an application program typically is waiting to open the slave virtual terminal. All application I/O under the operating system is directed to the slave virtual terminal. See also **login virtual terminal**.

source module

A text file (single source program) containing language statements, compile-time statements, and comments that can be compiled to produce an object module.

spanned

A record format in which each record in a tape file may begin in one block and end in another. See also **unspanned**.

spooler

A process that is associated with a printer. Each printer can have only one spooler associated with it. A spooler is created by the `login` request to the `spooler_admin` command.

stack

An area of storage consisting of an ordered series of stack frames associated with the execution of a program.

stack frame

An area of storage on the stack associated with an activation of a procedure, begin block, or on-unit.

star name

A name that contains one or more asterisks or consists solely of an asterisk. A star name can be used to specify a set of objects. Star names function in the following manner.

- An asterisk can be in any position in a star name.
- In a path name, a star name can be in the final name position only.
- When the operating system matches non-star names to a star name, each asterisk represents zero or more characters.
- A name cannot contain consecutive asterisks; there must always be an intervening character.

Some names with asterisks function differently; see **module star name** and **user star name**.

start

In TPF, to prepare a task to run, designating the program entry point at which it is to begin execution.

started process

A process that runs independently of and concurrently with your interactive process.

startup

The procedure that powers on and boots a module.

startup command macro

A command macro named `start_up.cm` that is located in a user's home directory. When the user logs in, the operating system runs the command macro before placing the user's process at command level or placing the process in the subsystem that is specified in the user's registration entry or named in the `login` command's `-subsystem` argument.

static data

Per-process temporary data used by a program. External static data can be shared among several object modules (in which case it is called a shared variable); internal static data can be referenced only by the program that declared it.

static pointer

A biased pointer to the static storage for a program. In VOS PL/I, every entry value includes a static pointer.

static region

The region of an object module that contains external references and internal static data.

static storage

Storage allocated within the program module prior to program execution. Variables have the static storage class only if you specifically give the `static` or `external` attribute in the variable declaration.

static task

A task that is defined at bind time and remains defined for the duration of the program run.

status access

A type of directory access that means a user can display information about the directory, but cannot modify the directory by creating, deleting, or renaming objects.

status code

A two-byte integer, with an associated name, indicating the success, failure, or other status of an operation. A zero status code generally indicates a successful operation.

In the operating system, there are four types of status codes:

- error codes, whose names begin with the prefix `e$`
- message codes, whose names begin with the prefix `m$`
- query codes, whose names begin with the prefix `q$`
- response codes, whose names begin with the prefix `r$`.

storage class

In VOS PL/I, an attribute indicating when and where storage is allocated for an object. VOS PL/I supports five storage classes: `automatic`, `based`, `defined`, `static`, and parameter.

StrataLINK local area network

The Stratus local network facility used to connect modules within a system and/or connect systems located at the same site (the same building or neighboring buildings) in order to provide transparent use of the operating system among Stratus modules and systems. StrataLINK operation requires the installation of special link hardware and cables and the creation of configuration tables and certain software processes (such as link servers).

StrataNET wide area network

The Stratus wide area network facility used to connect geographically remote systems in order to provide transparent use of the operating system. StrataNET operation requires the purchase of X.25 StrataNET software, the installation of a synchronous line adapter, and the creation of configuration tables and certain software processes (such as network clients and servers).

stream file

1. A sequentially organized file that contains records of varying sizes stored in a disk or tape region holding approximately the same number of bytes as the record. Thus, the record storage regions in a stream file vary from record to record.

   Stream files are accessible on either a record or byte basis. This allows the user to read the next record from a stream file, or read a specified number of bytes from a record while ignoring the file's record structure. See also **fixed file**, **relative file**, and **sequential file**.

stream organization

A VOS file organization in which stream files with varying length records are stored in a disk or tape region holding approximately the same number of bytes as the record. The record storage regions vary from record to record, and may be accessed on a record or byte basis.

When stream files are used to store text, each record contains one line of text. See also **fixed organization**, **relative organization**, and **sequential organization**.

string

A character string or bit string. See also **character string**.

string data

Character-string data or bit-string data.

structure

1. In PL/I, a collection of hierarchically arranged variables.

2. In C, an object that comprises an ordered sequence of named members, possibly of different types.

subprocess

An additional user process begun while the former process is suspended but not terminated. The new subprocess "inherits" several attributes from the former process, such as priority and access privileges, and the current directory. Each successive process that you start is a subprocess of the preceding process. Therefore, you must log out of each new subprocess to return to your original process.

subroutine

1. A sequence of statements that can be invoked as a set at one or more points in a program to execute a specific operation.

2. In VOS PL/I, a procedure that can be invoked by a `call` statement from within another procedure. The `procedure` statement and all `entry` statements of a subroutine must not include the `returns` option. Any `return` statement in a subroutine must not include a return value.

subscript

An arithmetic value used to specify a particular element or elements of an array.

subsequent directive

A type of replacement directive in an abbreviations file that tells the operating system to replace an input string with an output string when the specified input string occurs at any point in the command string after the first word. See also **abbreviation**, **all directive**, **first directive**, and **replacement directive**.

subsystem

A VOS facility that enters a command loop in which you can give directives or requests that have functions unique to the subsystem. The most common subsystems are analyze-system, system-operator, and the debugger.

suffix

A character string that begins with a period and is appended to an object name to indicate the type of the object.

supplementary graphic character set

In the VOS internal character coding system, one of the graphic character sets that can be represented by the character codes in the range A0-FF (hexadecimal); for example, the non-ASCII half of Latin alphabet No. 1. The determination of which character set is represented by these codes in a particular instance depends on the default and on the presence of single- or locking-shift character codes preceding the graphic character code in question. See also **default character set**.

symbol directive

A type of replacement directive in an abbreviations file that tells the operating system to replace one character in the command string by another.

symbol table

1. The portion of a program module that contains data for debugging the program. The symbol table allows the debugger to convert the names of user-defined variables to locations of data or instructions. For object modules, the symbol table is placed in the symtab region.

2. A construct that the compiler creates in the symtab region of an object module to facilitate symbolic debugging. The symbol table allows the debugger to convert user-defined variable names to locations of data or instructions. The compiler creates a symbol table only if the `-table` or `-production_table` argument of a compile command is specified.

symtab

See **symbol table**.

symtab region

The portion of an object module that contains the symbol table.

synchronous communications

Transmission of data at a fixed rate, with the transmitter and receiver synchronized by means of synchronization characters located at the beginning of each message or block of data.

system

Either a single module or a group of up to 32 modules located at the same site (the same building or neighboring buildings), connected in a local network by means of the StrataLINK local network, and defined as a single logical entity. See also **ring**.

system name

The unique name assigned to each system in a network. System names are specified on a module's disk label and by the system administrator in the systems and backbone systems configuration tables. See also **configuration table**.

table

A named collection of related information that is stored in a database. See also **symbol table** and **table file**.

table file

A file that the operating system creates by means of the `create_table` command. All of the records in a table file have the same format and are subdivided into fixed-length fields. The name of a table file always ends in the suffix `.table`.

tape mark

A special mark on a tape that acts as a delimiter between labels and files. A record of zero length serves as a tape mark. On unlabeled tapes, two consecutive tape marks indicate the end of a volume.

target of a link

The immediate target of a link is the file, directory, or link specified by the link's path name. The ultimate target of a link is the file or directory that is the target of the last link in a chain.

task

In TPF, a process subdivision that shares many of the characteristics of the process itself but shares some of the process resources with other tasks. The same section or different sections of program code may execute in each task, with different file and device attachments. Because some process resources are shared among tasks, it may be more efficient to have a given application program executing in multiple tasks of the same process rather than in multiple processes.

task data region (TDR)

A data structure maintained only for tasking programs. It is located in the user heap, and contains information about a task such as state, priority, terminal port ID, CPU time consumed, and stack length.

task identifier

In TPF, the integer assigned by the binder or by the operating system to identify a particular task within a process.

task priority

In TPF, an integer task attribute in the range 0 to 255 inclusive which affects the order in which tasks are scheduled to run and the order in which events are notified in a process. Tasks with higher-numbered priorities normally run sooner, and events associated with those tasks are notified sooner.

task scheduling

The order in which tasks are selected for execution when the number of ready tasks exceeds available processors.

task state

In TPF, one of eight possible task states: uninitialized, initialized, ready, running, waiting, paused waiting, paused ready, and stopped.

task switch

In TPF, the passing of control of a process from one task to another. This occurs when the task currently executing must wait on some event, is paused or stopped, or completes a segment of processing and explicitly calls `s$reschedule_task`. The task stops running and the next task is chosen according to task priority from those ready to run.

text file

1. In general usage, a file of characters. A text file can contain graphic characters and control characters from the Stratus internal character set. See also **binary file**.

2. In the context of NLS functionality, a file whose default character-set attribute is set to other than none. This type of file can contain characters from a standard character set. Additionally, if the shift-mode attribute is set to other than none, the file can contain characters from more than one standard character set.

thread

A path of execution with its own unique processor state and stack. Similar to a task.

time slice

An interval of CPU time that the operating system allocates to a process.

timeout message

In the system operator subsystem, a notification that the operator has not responded to a command or query.

token

An identifier, literal constant, punctuation symbol, comment, or compile-time statement (in VOS PL/I) or compile-time directive (in VOS Pascal).

transaction

1. A sequence of operations that are performed as a unit. Typically, a transaction involves updating one or more pieces of data in a database.

2. In TPF, a collection of I/O operations that occur after a call to `s$start_transaction` or `s$start_priority_transaction` and before a call to `s$commit_transaction` or `s$abort_transaction`. See also **transaction file**.

3. In a relational database management system, a sequence of statements that a database treats as a single entity. A transaction can be `read_only`, in which case a user only has permission to read the data, `write_only`, in which case a user only has permission to write to the data, or `read_write`, in which case a user can read the data and write to it. A transaction is also known as a logical work unit.

transaction file

A file that can be accessed only after a call to `s$start_transaction` or `s$start_priority_transaction` and before a call to `s$commit_transaction` or `s$abort_transaction`. Any changes that are made to a transaction file will not be left partially completed, regardless of the state of the hardware or software. Either all of the changes will be made or none of them be made.

transaction log

In TPF, a log file used by the operating system to record I/O operations on transaction files. Requested changes to transaction files are written to this log file during a transaction and are not copied to the transaction file until the transaction is committed.

transaction protection

In TPF, ensuring the integrity of a series of I/O operations by including them in a transaction.

true

Unambiguously correct or valid; evaluating to the bit-string value `'1'b` in PL/I or to a non-zero value in C.

two-way queue

In TPF, a queue that requires a server to reply to a message received from a requester.

unblocked

A record format in which each block in a tape file contains only one record or record segment. See also **blocked**.

unlock

See **lock**.

unspanned

A record format in which each record in a tape file ends in the same block in which it begins. See also **spanned**.

user heap

A portion of a user's virtual address space in which the operating system can allocate storage for the user's programs. The user heap is a free storage region located after the executable image in the user's region of virtual storage. It grows up towards the user stack, which grows down from the highest virtual address.

user name

An identifier that is composed of a person name and a group name.

user process

See **process**.

user stack

The portion of a user's virtual address space that is used for the user's program procedure call stack.

user star name

A user name containing one or two asterisks that is used to specify a set of users. When a user attempts to use a file or directory to which an access control list applies, the operating system checks the user's access by matching user star names on the list to actual users and groups.

Either component of a user star name (the person name or the group name) can be an asterisk, or both components can be asterisks. An asterisk as the first component matches all person names; an asterisk as the second component matches all group names. In arguments that accept user star names, if only a person name (or only a single asterisk) is given, the operating system appends `.*` to the name.

value

A measurable, describable, storable quantity that is associated with a constant, variable, or expression.

variable

1. A data item whose value may be changed by the execution of the program. In COBOL, a variable used in a numeric expression must be a numeric or a numeric-edited elementary item.

2. A declared object that can be assigned a value.

varying-length character string

A character string that can have any length from zero up to some stated maximum.

virtual address space

A set of addresses to which a process can refer. The operating system gives each user a virtual address space of 4096 pages; the first 2048 pages belong to the kernel. Excluding the addresses used by the operating system, the size of a user's virtual address space is either two megabytes or eight megabytes.

virtual circuit

A bidirectional communications path that provides for full-duplex data transmission independent of a fixed physical circuit. In X.25, virtual circuits exist at the packet level to enable multiple users to share a single X.25 data link simultaneously. The packet layer manages virtual circuits as separate, logical channels. It multiplexes virtual circuit data transparently onto the data link, using virtual circuit IDs (logical channel numbers), which also enables it to demultiplex incoming virtual circuit data.

Virtual Circuit Facility

The VOS software that provides the user interface to VOS virtual circuits and to X.25. It includes the virtual circuit subroutines that allow users (application programs) to establish X.25 and/or VOS virtual circuits on a per-call basis. The DTE/DCE operations of X.25 are handled entirely by the Virtual Circuit Facility.

Virtual Operating System

See **VOS operating system**.

virtual terminal

Under the operating system, a logical device used to support the host side of the X.29 protocol. Virtual terminals operate in a manner similar to real asynchronous terminals, and thus serve to insulate the operating system from the I/O requirements of X.25 and X.29. Virtual terminals can be defined either as slave terminals or login terminals. See also **slave virtual terminal** and **login virtual terminal**.

volume

A logical grouping of information on a tape. In most cases, one volume is contained on one reel, and the terms can be used interchangeably. One volume can contain a single file, multiple files, or a portion of a file.

VOS internal character coding system

The system used internally for encoding character data on Stratus systems. This system, based on the international standard ISO-2022-1986, allows encoding the multiple character sets needed for National Language Support.

See also **ASCII**, **character code**, **graphic character**, and **supplementary graphic character set**.

VOS operating system

The virtual operating system of a Stratus computer.

wait mode

A port setting that requires a task or process to wait until a requested I/O operation completes before it can continue running.

wait time

The maximum number of seconds that a process or a task will wait to lock a file or record during any I/O operation.

wide area network

A network that connects computers and other data processing components that are geographically remote. For example, the StrataNET wide area network is the Stratus wide area network facility that provides transparent use of the operating system among Stratus systems. See also **local network**.

word

Two bytes of data aligned on a two-byte boundary. An unsigned word variable can be assigned integer values in the range 0 to 65535; a signed word variable can be assigned integer values in the range -32768 to 32767.

word-aligned

Having storage that begins on a word boundary.

write access

A type of file access that means a user can execute, read, and write the file.

write lock

A lock that prevents all other tasks and processes from setting either a read or a write lock on a given object. It allows a writer to ensure that an object will not be read while being modified and that multiple tasks or processes are not simultaneously modifying the same object.

Index

D

L

M

N

O

P

T

U

V

W